1.1.1. Flex 和 Bison 介绍

(1) Flex 介绍

1975 年 Mike Lesk 和实习生 Eric Schmidt 设计并实现了一个词法分析器 lex（lexical analyzer generator），其中大部分的实现工作是由 Schmidt 完成的。lex 既可以独立使用也可以与 Johnson 的 yacc 配合使用。虽然 lex 运行比较慢并且也不太稳定，但是应用还是非常广泛。Schmidt 后来担任了 Google 的 CEO。大约 1987 年左右，Lawrence Berkeley 实验室的 Vern Paxson 将使用 ratfor 语言（一种 Fortran 扩展语言）编写的一个 lex 版本翻译成了 C 版本，并重新命名为 flex（Fast Lexical Analyzer Generator）。这个版本比 AT&T 版本的 lex 更快更稳定，并使用了 Berkeley 开源协议，因此取代了原来的 lex。Flex 以前是 SourceForge 的一个开源项目，目前托管在 github 上，可以从 https://github.com/westes/flex 下载。

在 Ubuntu 系统直接安装 flex 的命令为：

apt install flex

flex 采用扩展的正规式作为单词规则的描述方式，一个 flex 的输入文件基本上是由一些正规式以及匹配到单词后的处理动作构成的。Flex 生成的词法分析器读入输入，匹配所有的正规式并激活相应的处理动作。如下所示为一个简单的词法分析器的输入文件：

```
/* recognize tokens for the calculator and print them out */

%%
"+"     { printf("PLUS\n");       }
"-"     { printf("MINUS\n");      }
"*"     { printf("TIMES\n");      }
"/"     { printf("DIVIDE\n");     }
"|"     { printf("ABS\n");        }
[0-9]+  { printf("NUMBER %s\n", yytext); }
\n      { printf("NEWLINE\n"); }
[ \t]   {                        }
.       { printf("Mystery character %s\n", yytext); }
%%
```

设上述文件存储为 calc.l，则用下面的命令生成对应的词法分析程序:

flex calc.l

然后对生成的程序进行编译:

gcc lex.yy.c -lfl

(2) Bison 介绍

yacc 是 1975 到 1978 年间由 Bell 实验室的 Stephen C. Johnson 设计并实现的一个语法分析器自动生成工具。在 yacc 实现期间，许多人都在实现语法分析器，Yacc 是“yet another compiler compiler”的缩写，以 D. E. Knuth 坚实的分析工作作为理论基础，具有非常好的稳定性，在 Unix 类系统中使用非常广泛。但是由于当时的开源协议，在学术界和 Bell 系统以外使用 Yacc 非常受限。大约在 1985 年，加州伯克利大学的一个研究生 Bob Corbett 重新实现了 yacc,并采用了改进的分析算法,这个实现版本后来发展成为了 Berkeley Yacc。由于这个版本比 Bell 实验室的 yacc 要快很多,并且使用了更加灵活的 Berkeley 开源协议，因此很快变为最流行的一个 yacc 版本。自由软件基金会的 Richard Stallman 在 GNU 项目中使用了 Corbett 的工作，并对其进行了改写，增加了很多新的特性并最终发展为现在的 Bison。Bison 目前由 FSF 维护并且使用了 GNU 开源协议**错误!未找到引用源。**。

Bison 能够将输入的上下文无关文法转换为 LALR(1)语法分析器，另外也可以生成 LALR(1)和 LR(1)分析器，Bison 向后与 yacc 兼容，所有 yacc 能处理的文法不用修改就可以使用 Bison 生成相应的语法分析器，目前 Bison 已经支持 Java 版语法分析器的生成。

如下所示为逆波兰式计算对应的 Bison 输入文法，在此基础上添加 yylex, yyerror 和 main 函数之后就可以运行 Bison 生成完成的 C 文件。

Ubuntu 下安装 Bison 的命令为:

apt install bison

运行 Bison 的命令为:

gcc -o calc ./calc.tab.c -lm

ls 命令可以发现已经生成的可执行文件 calc，运行程序并输入逆波兰式：

*3 7 + 3 4 5 *+-*

可以看到输出的结果 “-13”。

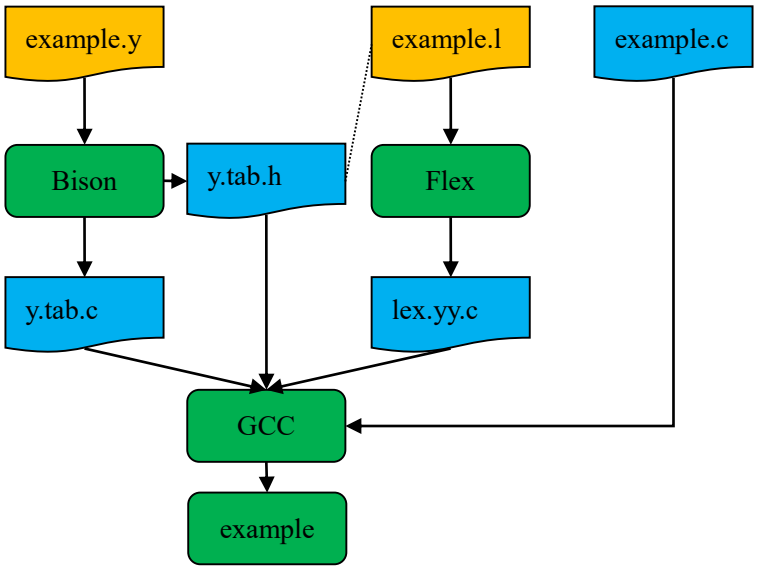
Bison 产生式中的每个符号都有值，目标符号（产生式左边符号）的值在代码中用 “$$” 表示，而产生式右边的符号从左向右分别用 “$1”、“$2” 等表示。终结符对应的值是词法分析器返回的 Token 的值，也就是 llval 中的值，其类型依赖于词法分析器的返回 Token 类型，而非终结符的值则是通过产生式对应的动作代码中设定或者计算的。在这个例子中，factor、term 和 exp 等符号的值代表了对应表达式的值。

```
/* Reverse Polish Notation calculator. */

%{
   #include <stdio.h>
   #include <math.h>
   int yylex (void);
   void yyerror (char const *);
%}

%define api.value.type {double}
%token NUM

%% /* Grammar rules and actions follow. */

input:
   %empty
| input line
;

line:
   '\n'
| exp '\n'          { printf ("%.10g\n", $1); }
;

exp:
   NUM
| exp exp '+'          { $$ = $1 + $2;          }
| exp exp '-'          { $$ = $1 - $2;          }
| exp exp '*'          { $$ = $1 * $2;          }
| exp exp '/'          { $$ = $1 / $2;          }
| exp exp '^'          { $$ = pow ($1, $2); }
| exp 'n'              { $$ = -$1;                }
;
%%
```

```
#include <ctype.h>

int
yylex (void)
{
   int c = getchar ();
   /* Skip white space. */
   while (c == ' ' || c == '\t')
      c = getchar ();
   /* Process numbers. */
   if (c == '.' || isdigit (c))
      {
         ungetc (c, stdin);
         scanf ("%lf", &yylval);
         return NUM;
      }
   /* Return end-of-input. */
   else if (c == EOF)
      return 0;
   /* Return a single char. */
   else
      return c;
}

/* Called by yyparse on error. */
void
yyerror (char const *s)
{
   fprintf (stderr, "%s\n", s);
}

int
main (void)
{
   return yyparse ();
}
```

（3）Flex 和 Bison 配合使用

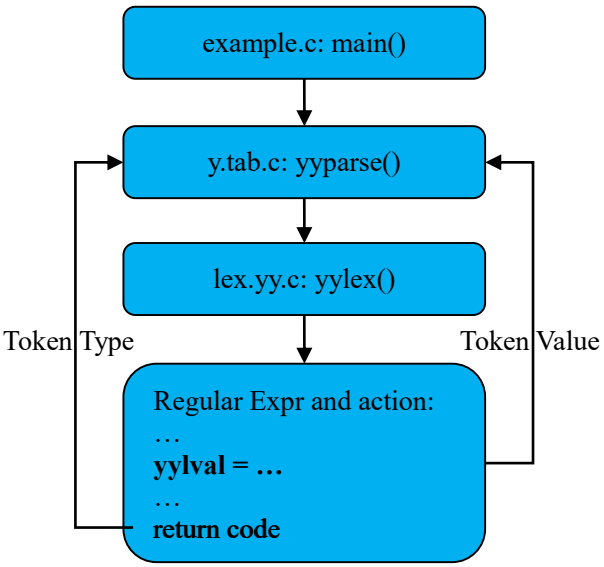
词法分析器的作用是识别单词并以 token 的形式返回给语法分析器，因此要让二者自然衔接并配合工作就需要定义二者的交互接口和共享数据结构。下图给出了相关文件之间的关系，example.l 和 example.y 分别是 Flex 和 Bison 的输入。首先 example.y 经过 Bison 处理后生成了 y.tab.h 和 y.tab.c，其中，y.tab.h 主要包括了文法终结符，即属性字的定义，也就是词法分析器返回的单词的类型。example.l 需要引用将该头文件，当时别到某个单词时，按照 y.tab.l 中的定义返回。除了 Flex 和 Bison 生成的词法分析器和语法分析器，用户还需要编写包括 mian 和出错处理程序的 example.c，并一起使用 GCC 编译生成最终的 example 这个程序。



如下所示为这几个文件中相关函数和代码的执行流程，example.c 中的 main 函数调用 y.tab.c 中的 yyparse 函数对输入代码进行语法分析，yyparse 多次调用 lex.yy.c 中的 yylex 返回一个终结符 token，每调用一次只返回一个 token，包括 token 类型和内码值两部分内容，其中如果 yylval 是 lex 生成的词法分析器中定义的一个全局变量，其类型为数值型。当然除了 yylval 之外，lex 中还定义了其他全局变量作为词法分析器和语法分析器之间交换数据的公共缓冲区，具体包括：

- char * yytext：指向单词的文本字符串，以'\0'结尾；

- int ylen: 单词字符串的长度，不包括最后的'\0';
- yylval: 表示单词内码值的全局变量，类型可以在 Bison 中定义。



以前面后缀表达式的例子为例来说明 Flex 和 Bison 配合使用的方法。Bison 生成的词法分析器默认会调用 yylex 函数返回下一个 token，而 Flex 生成的词法分析器则通过 yylex 完成上述功能，另外使用 Bison 时需要使用如下的命令:

```
bison –d calc.y
```

该命令生成一个头文件 calc.tab.h，这个头文件里面定义了包括所有 token 的 enum，我们需要修改 calc.l，当匹配到相应的单词的时候，需要按照预定义的枚举值返回 token 类型，并将 token 的值则存储在 yyval 或者 yytext 中。

如下为修改过的 calc.l 和 calc.y 文件:

```
%{
   #include "calc.tab.h"
%}

%%
"+"         { return ADD;         }
"-"         { return SUB;         }
"*"         { return MUL;         }
"/"         { return DIV;         }
"^"         { return POW;         }
[0-9]+      { yylval = atoi(yytext); return NUM; }
\n          { return EOL;         }
[ \t]       {                     }
.           { printf("Mystery character %s\n", yytext); }
%%
```

```
/* Reverse Polish Notation calculator. */

%{
   #include <stdio.h>
   #include <math.h>
   int yylex (void);
   void yyerror (char const *);
%}

%define api.value.type {double}
%token NUM
%token ADD SUB MUL DIV POW NEG
%token EOL

%% /* Grammar rules and actions follow. */

input:
   %empty
| input line
;

line:
   EOL
| exp EOL            { printf ("%.10g\n", $1);       }
;

exp:
   NUM
| exp exp ADD        { $$ = $1 + $2;                 }
| exp exp SUB        { $$ = $1 - $2;                 }
| exp exp MUL        { $$ = $1 * $2;                 }
| exp exp DIV        { $$ = $1 / $2;                 }
| exp exp POW        { $$ = pow ($1, $2);            }   /* Exponentiation */
| exp NEG            { $$ = -$1;                     }   /* Unary minus    */
;
%%

#include <ctype.h>

/* Called by yyparse on error. */
void
yyerror (char const *s){
   fprintf (stderr, "%s\n", s);
}

int main (void){
   return yyparse ();
}
```

编译运行的命令为:

```
bison -d calc.y

flex calc.l

gcc -o calc lex.yy.c calc.tab.c -lfl -lm
```

1.1.2. 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;

（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。

1.1.3. 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自动机，并使用 Java、C\C++或者 Python 其中任何一种语言，采用程序中心法或者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。

学生可以选择编码实现词法分析器，也可以选择使用 Flex 自动生成词法分析器。需要注意的是，Flex 生成的是 C 为实现语言的词法分析器，如果需要生成 Java 为实现语言的词法分析器，可以尝试 JFlex 或者 ANTLR。

由于框架是基于 Java 语言实现的，并且提供了相应的示例程序，建议学生使用 Java 语言在示例的基础上完成词法分析器。

1.1.4. 实验过程与方法

该实验以 C 语言作为源语言，构建 C 语言的词法分析器，对于给定的测试程序，输出属性字符流。词法分析器的构建按照 C 语言的词法规则进行。C 语言的发展经历了不同的阶段，早期按照 C99 标准进行编程和编译器的实现，2011 年又对 C 语言规范进行了修订，形成了 C11（又称 C1X)。下面以 C11 为基准，对 C 语言的词法规则进行简要的描述。

C 语言的关键字包括如下单词:

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***	

C 语言标识符的定义如下:

identifier → *identifier-nondigit*
| *identifier identifier-nondigit*
| *identifier digit*

identifier-nondigit → *nodigit*
| *universal-character-name*
| *other implementation-define characters*

nondigit → _ | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r*
| *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z* | *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I*
| *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* | *U* | *V* | *W* | *X* | *Y*
| *Z*

digit → *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

C 语言整型常量的定义如下:

integer-constant→ *decimal-constant integer-suffix*
| *octal-constant integer-suffix*
| *hexadecimal-constant integer-suffix*

decimal-constant → *nonzero-digit* | *decimal-constant digit*

octal-constant → *0* | *octal-constant octal-digit*

hexadecimal-constant → *hexadecimal-prefix hexadecimal-digit*
| *hexadecimal-constant hexadecimal-digit*

hexadecimal-prefix → *0x* | *0X*

nonzero-digit → *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

octal-digit → *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7*

hexadecimal-digit → *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* |*9* | *a* | *b* | *c* | *d* | *e* | *f* |
A | *B* | *C* | *D* | *E* | *F*

integer-suffix → *unsigned-suffix long-suffix*
| *unsigned suffix long-long suffix*
| *long-suffix unsigned-suffix*
| *long-long-suffix unsigned-suffix*

unsigned-suffix → *u* | *U*

long-suffix → *l* | *L*

long-long-suffix → *ll* | *LL*

C 语言浮点型常量定义如下:

floating-constant→ decimal-floating-constant
| hexadecimal-floating-constant
decimal-floating-constant → fractional-constant exponent-part floating-suffix
| digit-sequence exponent-part floating-suffix
hexadecimal-floating-constant → hexadecimal-prefix hexadecimal-fraction-constant
binary-exponent-part floating-suffix
| hexadecimal-prefix hexadecimal-digit-sequence
binary-exponent-part floating-suffix
fractional-constant → digit-sequence . digit-sequence | digit-sequence .
exponent-part → e sign digit-sequence | E sign digit-sequence
sign → + | -
digit-sequence → digit | digit-sequence digit
hexadecimal-fractional-constant → hexadecimal-digit-sequence . hexadecimal-digit-sequence
| hexadecimal-digit-sequence .
binary-exponent-part → p sign digit-sequence | P sigh digit-sequence
hexadecimal-digit-sequence → hexadecimal-digit
| hexadecimal-digit-sequence hexadecimal-digit
floating-suffix → f | l | F | L

C 语言字符常量定义如下:

character-constant→ ' c-char-sequence ' | L' c-char-sequence ' | u' c-char-sequence '
| U' c-char-sequence '
c-char-sequence → c-char | c-char-sequence c-char
c-char → any member of the source character set except the single-quote ', backslash \, or new-line character
| escape-sequence
escape-sequence → simple-escape-sequence | octal-escape-sequence
| hexadecimal-escape-sequence | universal-character-name
simple-escape-sequence → \' | \" | \? | \\ | \a | \b | \f | \n | \r | \t | \v

C 语言字符串字面量定义如下:

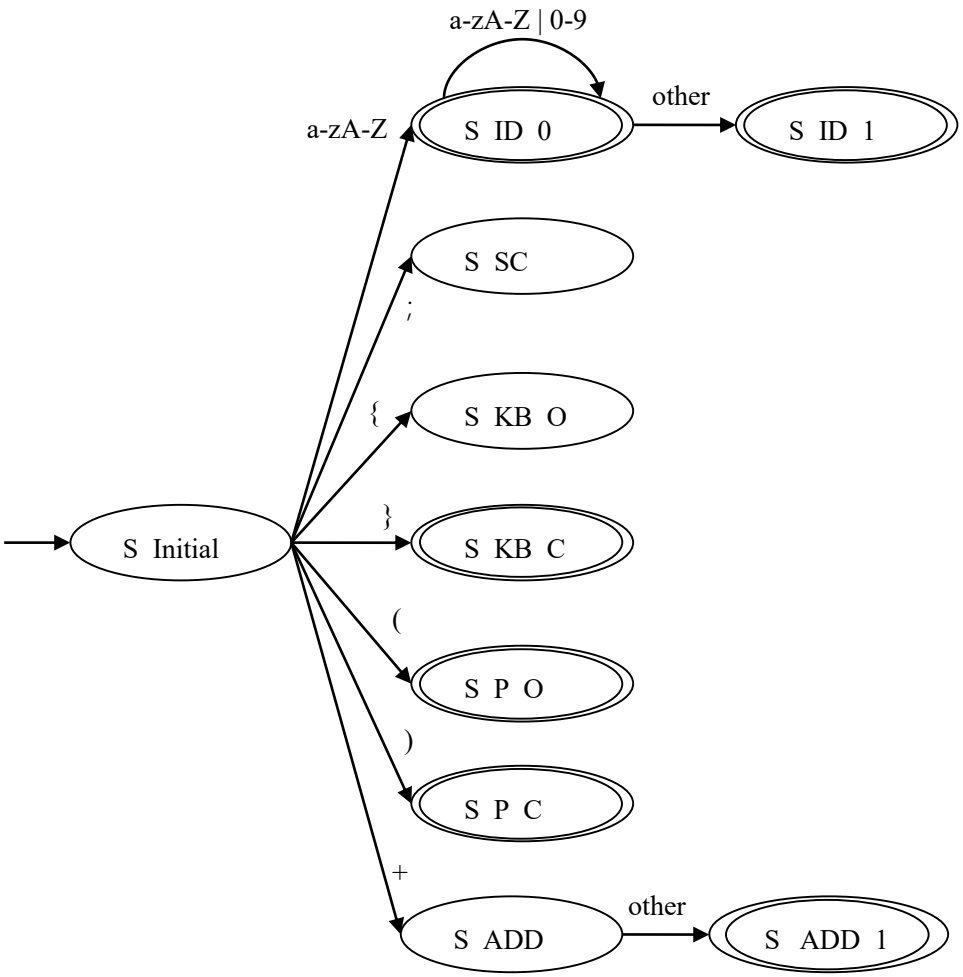
string-literal→encoding-prefix " s-char-sequence "
encoding-prefix → u8 | u | U | L
s-char-sequence → s-char | s-char-sequence s-char
s-char → any member of the source character set except the double-quote ", backslash \, or new-line character
| escape-sequence

C 语言运算符和界限符定义如下:

[	]	(	)	{	}	.	->
++	--	&	*	+	-	~	!
/	%	<<	>>	<	>	<=	>=
==	!=	^	|	&&	||		
?	:	;	...				
=	*=	/=	%=	+=	-=	<<=	
>>=							
&=	^=	|=	,	#	##	<:	:>
<%	%>	%:	%:%:				

从 github 下载 BIT-MINICC 框架，下载网址为：https://github.com/jiweixing/bit-minic-compiler；编写测试程序，并使用内置的词法分析器对输入进行测试，观察词法分析器的输入和输出；选择实现的语言，设计实现自己的 C 语言词法分析器。

如果编码实现词法分析器，则可以首先设计 DFA，以如下识别标识符和少数操作符的 DFA 为例：



基于示例实现，对于如下的测试程序 scanner_example.c:

```
int sum(int a, int b) { return a + b; }
```

输出的属性字流存储在 scanner_example.tokens 文件中，具体内容见下:

```
[@0,0:2='int',<'int'>,1:0]
[@1,4:6='sum',<Identifier>,1:4]
[@2,8:8='(',<'('>,1:8]
[@3,10:12='int',<'int'>,1:10]
[@4,14:14='a',<Identifier>,1:14]
[@5,16:16=',',<','>,1:16]
[@6,18:20='int',<'int'>,1:18]
[@7,22:22='b',<Identifier>,1:22]
[@8,24:24=')',<')'>,1:24]
[@9,26:26='{',<'{'>,1:26]
[@10,28:33='return',<'return'>,1:28]
[@11,35:35='a',<Identifier>,1:35]
[@12,37:37='+',<'+'>,1:37]
[@13,39:39='b',<Identifier>,1:39]
[@14,41:41=';',<';'>,1:41]
[@15,43:43='}',<'}'>,1:43]
[@16,47:46='<EOF>',<EOF>,2:0]
```

在这个输出的 token 流中，每行为一个 token，以@开头的数字表示 token 的序号，紧接着的 xx:xx 表示 token 文本对应的开始列和结束列，“=”后面给出了这个范围之内 token 的具体文本，“<>”之内表示 token 的类型，最后一个数字对 xx:xx 表示起始行和起始列。需要说明的是，在这个例子中，运算符等的类型就是其自身，属性流的最后放置了一个“EOF”表示属性字的结束位置。

BITMiniCC 词法分析器的实例代码 ExampleScanner.java 中给出了一个基于这个 DFA 的参考实现。

1.1.5. 实验提交内容

使用 Java 语言实现的,请提交框架 src 目录下源码和配置文件 config.xml，不用包括 lib 等其他文件夹;

使用 C/C++或者其他语言实现的，请创建 src 文件夹，并将源码放入该文件夹；创建文件夹 bin，将源码对应的可执行程序放置到该目录下。如果用框架调用过，请提交相应的配置文件 config.xml。

该实验需提交实验报告，具体内容如下:

- 实验目的和内容
- 实现的具体过程和步骤
- 运行效果截图
- 实验心得体会

请在实验报告里具体描述词法分析器的设计和实现方法，例如手动编写应给出自动机设计结果，词法分析器实现所使用的主要数据结构和算法。