华 中 科 技 大 学

网络空间安全学院

本科：《编译原理》实验报告

姓　　名 _____

班　　级 _____

学　　号 _____

联系方式 _____

分　　数 _____

评 分 人 _____

实验报告和设计评分细则

评 分 项 目		满分	得分	备注
课程目标 1（选择工具完成设计、实现，分析局限性）	词法分析	20		能够对实验任务进行正确分析；通过查阅资料，设计合理；对实验结果进行正确分析；对设计的局限性进行正确分析；
	语法分析	20		
	语义分析	20		
课程目标 2（提出问题、探索求解、归纳总结、创造性）	词法分析	10		能够对任务进行反思、提出问题，正确描述问题、分析问题；能体现对问题的探索求解，能查阅资料、解决问题；能准确归纳、总结方案的优点、缺点；能体现出创造性；
	语法分析	20		
	语义分析	10		
实验报告总分		100		
教师签名			日　　期	

目　录

1 词法分析实验

1.1 实验概述

1.1.1 实验目的

通过词法分析程序的设计、实现，掌握词法分析程序的原理及不同的实现方法。能主动学习，通过查阅资料，掌握 Flex 工具的使用。分析不同方案的优缺点。

1.1.2 实验内容

（1） 手工编写词法分析程序；
（2） 了解 Flex 工具的使用；
（3） 借助 Flex 工具自动产生词法分析程序。

1.2 实验过程、结果、小结

本次实验分为三个主要任务，针对一个简单语言的词法规则进行词法分析程序的实现。语言的词法规则包括关键字、运算符、界符、标识符（ID）和整型常数（NUM），具体定义如下表所示：

表 1-1 语言的词法规则定义表

词法规则	示例
关键字（全小写）	begin、if、then、while、do、end
运算符和界符	、:=、+、-、*、/、<、<=、<>、>、>=、=、;、(、)、#
标识符（ID）	由字母开头，后接字母或数字（正规式：letter(letter|digit)*）
整型常数（NUM）	由数字组成（正规式：digit(digit)*）
空格	包括空白、制表符和换行符，用于分隔单词，词法分析阶段忽略

实验的输入为类似如下代码片段，预期输出为每行一个二元组，格式为(种别码, 单词值)，如(1,begin)、(10,x)等。

```
begin
    x := 10;
    y := x + 5;
    z := y * 2;
end
```

1.2.1 实验过程中遇到的问题及解决

在完成词法分析实验的过程中，我遇到了以下问题，并通过查阅资料和调试逐步解决：

问题 1：手工编写程序中字符回退的处理

在手工编写的 C 语言词法分析程序中，识别标识符或数字时，需要读取字符直到遇到非字母或非数字的字符，但该字符可能属于下一个单词的开始。例如，识别 x 后遇到:=，需要将:回退到输入流以供后续处理。

解决方法：使用 ungetc()函数将多读的字符回退到输入流。例如，在识别标识符后，如果读取的下一个字符不是 EOF，则调用 ungetc(ch, stdin)，确保后续处理不会丢失字符。这种方法简单有效，保证了字符流的连续性。

问题 2：多字符运算符的识别

对于如:=、<=、>=等由两个字符组成的运算符，需要在读取第一个字符（如:）后，检查下一个字符是否构成复合运算符。如果不是（如单独的:），需要正确回退并输出单字符的种别码。

解决方法：在处理:、<、>等可能构成复合运算符的字符时，读取下一个字符并进行条件判断。例如，处理:时：

```
case ':':
    int next = getChar();
    if (next == '=') {
        printf("(%d,:=)\n", 18);
    } else {
        printf("(%d,:)\n", 17);
        if (next != EOF) ungetc(next, stdin);
    }
    break;
```

通过这种方式，确保了复合运算符和单字符运算符的正确区分。

问题 3：Flex 规则的优先级问题

在使用 Flex 生成词法分析器时，初始版本的规则中，标识符的规则[a-zA-Z][a-zA-Z0-9]*与关键字的规则（如"begin"）存在潜在冲突，可能导致关键字被错误识别为标识符。

解决方法：在 Flex 中，规则的匹配遵循最长匹配原则和优先级原则（靠前的规则优先匹配）。因此，我将关键字的规则（如"begin"）放在标识符规则之前，确保关键字优先被识别。例如：

```
"begin" { printf("(1,%s)\n", yytext); }
[a-zA-Z][a-zA-Z0-9]* { printf("(10,%s)\n", yytext); }
```

此外，通过查阅 Flex 手册，确认 Flex 会优先选择最长匹配的规则，进一步避免了冲突。

问题 4：Flex 中行数统计的边界条件

在任务 2 的 Flex 任务中，统计行数和字符数时，发现当输入以非换行符结束时，行数可能少计。例如，输入 123456 没有换行符，预期行数为 1，但初始代码未正确处理。

解决方法：在 yywrap()函数中添加边界条件处理，当字符数大于 0 且行数未正确计入时，

增行数计数：

```
int yywrap() {
    if (num_chars > 0 && !(num_lines > 0 && num_chars == num_lines))
        ++num_lines;
    return 1;
}
```

这确保了即使输入没有换行符，行数也能正确统计。

1.2.2 实验结果分析

实验通过手工编写和 Flex 工具生成两种方式实现了词法分析器，以下是对实验结果的分析：

任务 1：手工编写词法分析程序

使用 C 语言，通过 getchar()逐字符读取输入，结合状态转换逻辑识别关键字、标识符、数字和运算符/界符。关键字通过预定义的 Keyword 结构体数组进行查找，运算符和界符通过 switch-case 结构处理。在 OJ 平台对输入程序片段进行分析，输出完全符合预期如下图所示。

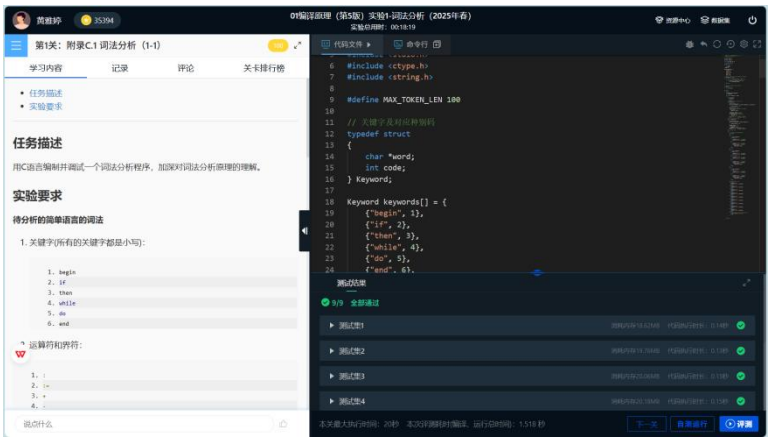

图 1-1 任务 1 测试结果

任务 2：Flex 统计行数和字符数

通过 Flex 编写词法描述文件，定义规则统计换行符（\n）和任意字符（.），在 main 函数中输出统计结果。在 OJ 平台测试结果正确，验证了 Flex 规则的有效性。

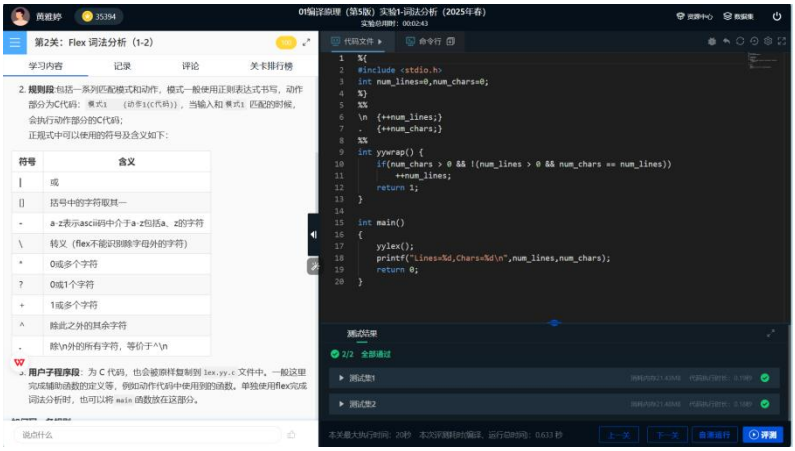

图 1-2 任务 2 测试结果

同时我们也可以在本地 Ubuntu 环境中使用 Flex 编译源文件再次验证，生成 C 语言文件，然后使用 gcc 编译 C 语言文件，生成可执行文件，如图 1-3 所示。

```
l$ flex -o lex.yy.c task1.l
l$ gcc -o mylex lex.yy.c
```

图 1-3 编译 Flex 源文件并生成可执行文件

运行 mylex 程序，向控制台中输入一些数据，按 Ctrl+D 结束输入后，程序的输出结果如图 1-4 所示，可知程序能够正确统计行数和字符数。

```
123456
abcdef
abcdef
Lines=3,Chars=18
```

图 1-4 程序运行结果

任务 3：Flex 生成词法分析程序

使用 Flex 定义词法规则，直接匹配关键字、运算符、标识符和数字，每条规则对应一个 printf 动作输出种别码和单词值。生成的 lex.yy.c 文件通过编译链接生成可执行程序。对相同输入程序片段，输出与手工编写程序完全一致，验证了 Flex 生成的词法分析器的正确性。而 Flex 生成的程序基于正则表达式和 DFA，匹配速度快，代码简洁，维护成本低。相比手工编写，Flex 在处理复杂词法规则时具有显著优势。在 OJ 平台进行测试验证如下图所示。

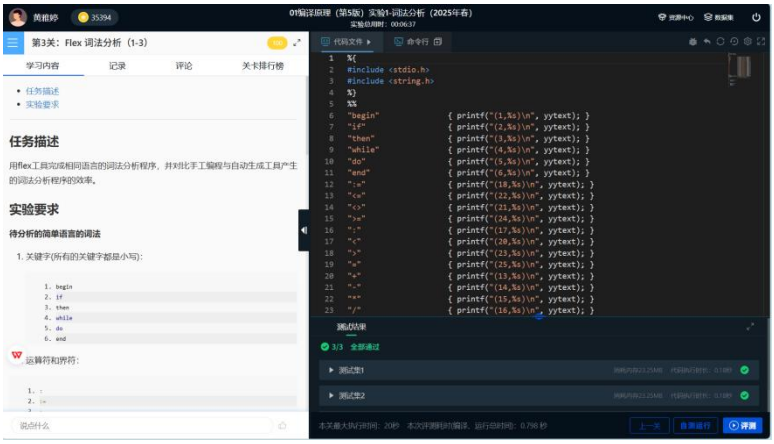

图 1-5 任务 3 测试结果

1.2.3 小结、设计存在的不足

在本次词法分析实验里，我收获颇丰。首先，通过亲手编写程序，我深入领悟了从字符流到单词流的转换过程，也切实掌握了状态机在词法分析中的具体应用。接着，我查阅 Flex 手册并调试相关示例，由此熟悉了 Flex 词法描述文件的结构，学会了正则表达式的编写，以及规则与动作的搭配使用。此外，我分别实现了手工编写和利用 Flex 生成这两种词法分析器，并对二者在开发效率、代码可读性、扩展性和运行性能等方面的差异进行了分析，进而认识到工具生成方式所具备的优势，我还将这些分析总结成了表格，如下表 1-2 所示。在实验过程中，我通过查阅 Flex 手册、C 语言参考文档等资料，成功解决了遇到的技术问题。

表 1-2 手工编写与 Flex 生成的对比

特性	手工编写程序	Flex 生成程序
开发效率	较低，需手动实现状态机逻辑	较高，仅需编写规则，自动生成代码
代码量	较多（约 150 行）	较少（约 40 行规则+少量 C 代码）
可读性	逻辑清晰，但代码复杂	规则简洁，易于理解和修改
扩展性	修改规则需调整大量代码	修改规则只需更新正则表达式
运行效率	依赖实现质量，通常稍慢	基于 DFA，运行效率高
适用场景	适合小型语言或教学目的	适合复杂语言和实际项目

2 语法分析实验

2.1 实验概述

2.1.1 实验目的

通过语法分析程序的设计、实现，掌握语法分析程序的原理及实现方法。能主动学习，通过查阅资料，掌握 Bison 工具的使用。

2.1.2 实验内容

（1） 手工编写语法分析程序；
（2） 了解 Bison 工具的使用。

2.2 实验过程、结果、小结

本次实验分为两个主要任务，针对一个简单语言的语法规则进行语法分析程序的实现。语言的语法规则以扩充的 BNF 范式定义，包含程序、语句串、语句、赋值语句、条件语句、循环语句、表达式、项、因子和逻辑运算等成分，具体定义如下：

```
<程序> ::= begin <语句串> end
<语句串> ::= <语句>{;<语句>}
<语句> ::= <赋值语句> | <条件语句> | <循环语句>
<赋值语句> ::= ID := <表达式>
<表达式> ::= <项> {+<项> | -<项>}
<项> ::= <因子> {*<因子> | /<因子>}
<因子> ::= ID | NUM | (<表达式>)
<条件语句> ::= if <逻辑运算> then <语句串> end
<循环语句> ::= while <逻辑运算> do <语句串> end
<逻辑运算> ::= <表达式> {<逻辑运算符> <表达式>}
<逻辑运算符> ::= < | <= | = | >= | > | <>
```

任务 1 是使用 C 语言实现递归下降分析器，解析输入的源程序，生成缩进形式的语法树输出，实验的输入为类似下面的代码片段，预期输出缩进形式的语法树，显示程序的结构层次。

```
begin
    x := 10;
    y := x + 5;
    if y > 15 then
        z := y * 2;
    end
    while y > 0 do
        y := y - 1;
    end
end
```

对于任务 2，任务是实现逆波兰表达式计算器，输入为后缀表达式（如 1 2 +），输出为计算结果（如 3）。

2.2.1 实验过程中遇到的问题及解决

在完成语法分析实验的过程中，我遇到了以下问题，并通过查阅资料和调试逐步解决：

问题 1：手工编写程序中输入缓冲区的处理

在任务 1 的递归下降分析程序中，输入程序需要逐行读取并拼接为一个连续的字符串，但直接使用 fgets 可能导致换行符干扰解析逻辑。例如，输入中的换行符可能被误认为是单词分隔符。

解决方法：在 main 函数中，读取输入时将换行符替换为空格，确保输入字符串的连续性：

```
while ((ch = getchar()) != EOF && idx < sizeof(input) - 1) {
    input[idx++] = (ch == '\n') ? ' ' : ch;
}
input[idx] = '\0';
```

这种方法消除了换行符的影响，保证了词法单元的正确分隔。

问题 2：逻辑运算符的识别

在解析逻辑运算（如<、<=、>=、<>)时，需要正确处理单字符和双字符运算符。例如，读取<后需检查是否跟=或>，否则可能误判为单个<。

解决方法：在 logical_expression 函数中，使用条件判断检查后续字符：

```
while ((ch = getchar()) != EOF && idx < sizeof(input) - 1) {
    input[idx++] = (ch == '\n') ? ' ' : ch;
}
input[idx] = '\0';
```

这确保了逻辑运算符的正确识别，并通过 printf 输出相应的运算符。

问题 3：条件语句和循环语句的嵌套处理

在解析 if 和 while 语句时，statement_list 需要正确处理嵌套的语句串，直到遇到对应的 end 关键字。初始版本中，if_statement 和 while_statement 错误地调用了 statement 而不是 statement_list，导致无法解析嵌套语句。

解决方法：修改 if_statement 和 while_statement 函数，调用 statement_list 以支持嵌套语句串：

```
void if_statement(int indent) {
    printf("%*s 条件语句:\n", indent, "");
    pos += 2; skip_whitespace();
    logical_expression(indent + 4);
    skip_whitespace();
    if (strncmp(input + pos, "then", 4) == 0) {
        pos += 4; skip_whitespace();
        statement_list(indent + 4); //  修正为调用 statement_list
        skip_whitespace();
        if (strncmp(input + pos, "end", 3) == 0) {
```

```
                pos += 3;
                printf("%*s 结束条件语句\n", indent, "");
            } else printf("错误：缺少 end");
        } else printf("错误：缺少 then");
    }
```

类似修正也应用于 while_statement，确保了嵌套结构的正确解析。

问题 4：Bison 中词法分析器的集成

在第 2 关的 Bison 任务中，词法分析器 yylex 需要与 Bison 生成的语法分析器无缝协作。初始版本的 yylex 未正确处理数字输入，可能导致浮点数解析错误。

解决方法：通过 scanf 读取浮点数，并将值存储在 yylval 中：

```
if (c == '.' || isdigit(c)) {
    ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUM;
}
```

此外，确保 yylex 返回正确的终结符（如 NUM 或运算符字符），与 Bison 的语法规则匹配。

问题 5：Bison 中运算优先级和结合性

在实现逆波兰表达式计算器时，运算符的优先级和结合性需要正确定义，以确保 1 2 + 3 * 被解析为(1 + 2) * 3 而不是 1 + (2 * 3)。

解决方法：在 Bison 文件中使用%left 和%right 定义运算符的结合性和优先级：

```
%left '+' '-'
%left '*' '/'
%right 'n'
%right '^'
```

其中，*和/的优先级高于+和-，^（幂运算）和 n（取负）具有最高优先级且右结合，确保了表达式的正确求值。

2.2.2 实验结果分析

实验通过手工编写递归下降分析器和 Bison 工具生成语法分析器完成了两个任务，以下是对实验结果的分析：

任务 1：手工编写语法分析程序

使用 C 语言实现递归下降分析器，按照 BNF 文法定义的规则，分别实现 program、statement_list、statement、assignment_statement、if_statement、while_statement、logical_expression、expression、term 和 factor 等函数。每个函数对应一个非终结符，通过递归调用解析输入字符串，生成缩进形式的语法树。在 OJ 平台进行测试验证如下图所示。

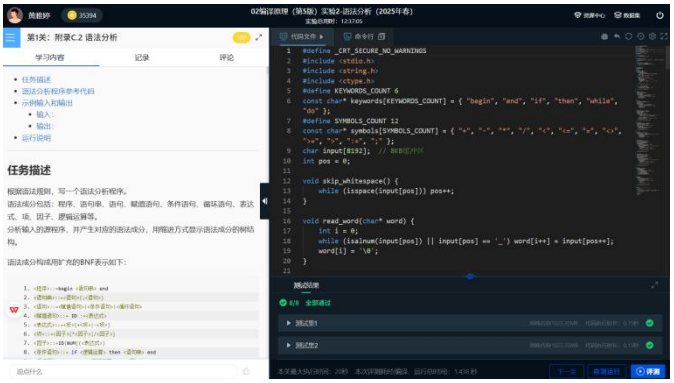

图 2-1 任务 1 测试结果

任务 2：Bison 生成逆波兰表达式计算器

参考样例所给的加法计算的语法规则，在其后添加剩余规则，如下所示：

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

使用 Bison 编写语法描述文件，定义逆波兰表达式的 BNF 文法，结合 yylex 词法分析器实现加法、减法、乘法、除法、幂运算和取负运算。生成的 calc.tab.c 文件通过编译链接生成可执行程序。在 OJ 平台进行测试验证如下图所示。

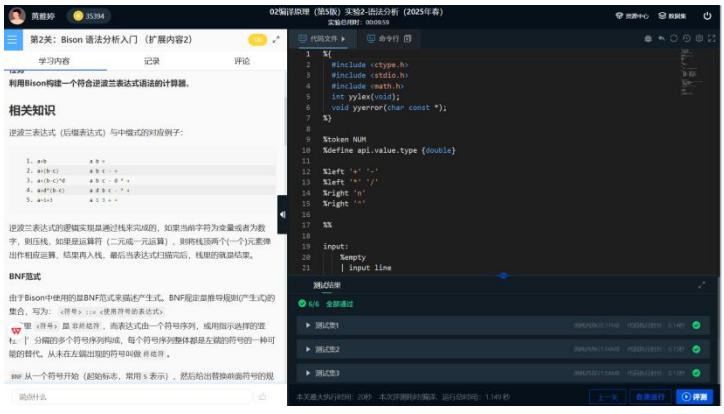

图 2-2 任务 2 测试结果

编译 Bison 源程序需要特别注意的是，使用 gcc 编译 C 语言程序时，需添加参数-lm，否则将无法调用 math.h 头文件中声明的库函数 pow()。程序编译完成后，输入若干后缀表达式进行测试，测试效果如图 2-3 所示，结果表明程序能够正确完成后缀表达式的计算。

图 2-3 逆波兰式计算程序结果

2.2.3 小结、设计存在的不足

在本次语法分析实验中，我通过手工编写递归下降分析器，深入理解了 BNF 文法的应用及语法树的构建逻辑，切实掌握了递归下降分析的具体实现方法。实验结果显示，手工编写的递归下降分析器与 Bison 生成的 LALR (1)分析器均能正确解析输入程序并生成预期输出。对比来看，Bison 工具在开发效率与性能表现上更具优势，更适用于实际编译器的开发场景。

3 语义分析实验

3.1 实验概述

3.1.1 实验目的

通过语义分析程序的设计、实现，掌握语法制导的语义分析程序的原理及实现方法。能主动学习，通过查阅资料，掌握 Bison 工具的使用。

3.1.2 实验内容

（1） 对算术表达式、赋值语句进行语义分析，并生成四元式序列；
（2） 分析语法分析器中的冲突，修改文法规则以消除无用非终结符和移进-归约冲突；
（3） 联合使用 Flex、Bison，完成中缀式计算；
（4） 对条件语句、循环语句进行语义分析，并生成四元式序列。

3.2 实验过程、结果、小结

本次实验分为四个主要任务，针对一个简单语言的语法规则进行语义分析，生成四元式序列。语言的语法规则以 BNF 范式定义，涉及算术表达式、赋值语句、条件语句和循环语句。实验的输入和预期输出在各任务中已明确，例如下面左边为任务 1 给出的需要进行语义分析的输入，右边为预期输出为四元式序列。

begin
a:=2+3*4;
x:=(a+b)/c
end
	t1=3*4
	t2=2+t1
	a=t2
	t3=a+b
	t4=t3/c
	x=t4

任务 4 输入包括条件语句和循环语句，预期输出包含控制流四元式（如 ifFalse 和 goto）。

3.2.1 实验过程中遇到的问题及解决

在完成语义分析实验的过程中，我遇到了以下问题，并通过查阅资料和调试逐步解决：

问题 1：四元式生成中的临时变量管理

在任务 1 和任务 4 中，生成四元式需要动态分配临时变量（如 t1、t2），但初始版本未正确管理内存，可能导致内存泄漏或重复使用临时变量。

解决方法：定义 new_temp 函数生成唯一的临时变量名，并使用 strdup 和 free 管理字符串内存。例如：

```
char* new_temp() {
    char* temp = (char*)malloc(10);
    sprintf(temp, "t%d", ++temp_count);
```

```
        return temp;
    }
```

在生成四元式后，及时释放不再使用的临时变量：

```
    char* result = parse_term();
    free(result);
    result = temp;
```

这确保了内存使用的正确性。

问题 2：Bison 移进-归约冲突

在任务 2 中，分析 foo.y 文件时，foo.output 显示状态 6 和状态 7 存在移进-归约冲突。例如，状态 6 中：

```
    State 6
        1 exp: exp . '+' exp
        1      | exp '+' exp .
        2      | exp . '-' exp
        '-'    shift, and go to state 5
        '-'            [reduce using rule 1 (exp)]
        $default    reduce using rule 1 (exp)
        Conflict between rule 1 and token '+' resolved as reduce (%left '+').
```

冲突原因是解析器在遇到+或-时，无法确定是继续移进（解析新的 exp）还是归约（完成当前 exp）。

解决方法：通过 Bison 的优先级声明（%left '+'）解决冲突，Bison 选择归约操作以符合左结合性。此外，删除无用的非终结符 useless 和终结符 STR，简化文法：

```
    %union {
        int ival;
    }
    %token <ival> NUM
    %nterm <ival> exp
    %left '+' '-'
    %%
    exp:
        exp '+' exp { $$ = $1 + $3; }
        | exp '-' exp { $$ = $1 - $3; }
        | NUM { $$ = $1; }
    ;
    %%
```

这消除了无用规则和冲突，生成正确的语法分析器。

问题 3：Flex 与 Bison 的接口一致性

在任务 3 中，Flex 生成的词法分析器需要与 Bison 的语法分析器协作，确保返回的终结符和值（如 NUM 和 double 类型）一致。初始版本中，词法分析器未正确处理负数，导致-2^2 被解析为-4 而不是 4。

解决方法：在 Bison 规则中添加一元负号规则，并设置最高优先级：

```
%right NEG
%%
exp:
    SUB exp %prec NEG { $$ = -$2; }
```

在 Flex 中确保正确识别数字（包括小数），并将值存储在 yylval 中：

```
[0-9]+(\.[0-9]+)? { yylval = atof(yytext); return NUM; }
```

这确保了负号的正确处理和词法-语法接口的一致性。

问题 4：控制流四元式的生成

在任务 4 中，条件语句和循环语句需要生成控制流四元式（如 ifFalse 和 goto），但初始版本未正确处理循环的回跳逻辑，可能导致 goto 标签错误。

解决方法：在 parse_while 函数中，添加起始标签和回跳逻辑：

```
void parse_while() {
    match("while");
    char* label_start = new_label();
    char* label_end = new_label();
    add_quadruple("label", "", "", label_start);
    char* cond = parse_condition();
    add_quadruple("ifFalse", cond, "", label_end);
    parse_block();
    add_quadruple("goto", "", "", label_start);
    add_quadruple("label", "", "", label_end);
}
```

这确保了循环体的正确跳转和退出。

3.2.2 实验结果分析

通过手工编写和 Flex+Bison 联合实现完成了语义分析任务，以下是对实验结果的分析：

任务 1：算术表达式和赋值语句的语义分析

使用 C 语言实现递归下降分析器，结合语义动作生成四元式。parse_factor、parse_term 和 parse_expression 函数解析算术表达式，生成中间临时变量的四元式；parse_assignment 生成赋值语句的四元式。在 OJ 平台进行测试验证如下图所示，代码正确可以通过所有测试例。

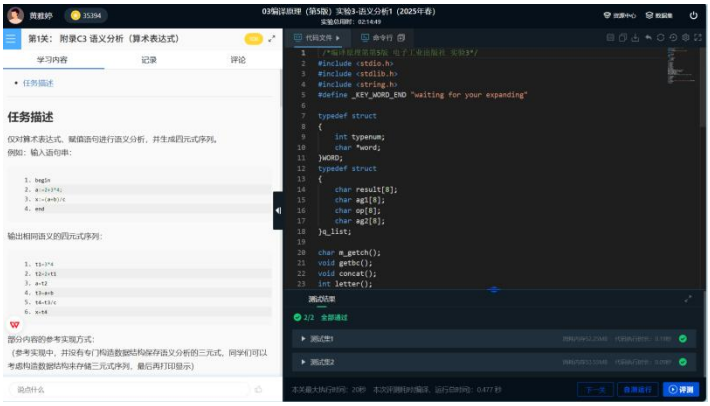

图 3-1 任务 1 测试结果

任务 2：Bison 移进-归约冲突解决

首先我们对任务 2 学习内容中提出的问题进行逐一的理解和解答。

问题 1 使用以下两个命令产生状态分析文件，对比二者不同：

```
bison -v foo.y                  // 获得对应状态分析文件
mv foo.output foo.output1       // 将产生的文件重命名为 foo.output1
bison -r solved foo.y           // 产生的文件为 foo.output
diff foo.output1 foo.output     // 查看两个文件不同之处
```

如下图 3-2 所示，我们可以看出很明显看到他们的区别在于 solved 文件明确指出 **Conflict between rule 1 and token '+' resolved as reduce (%left '+')**。说明 Bison 在处理语法分析时遇到了一个移进-归约冲突：按照规则 1，既可以移进下一个符号'+'，也可以将其归约。而 Bison 利用'+'的优先级和结合性解决了冲突（详见问题 3）。

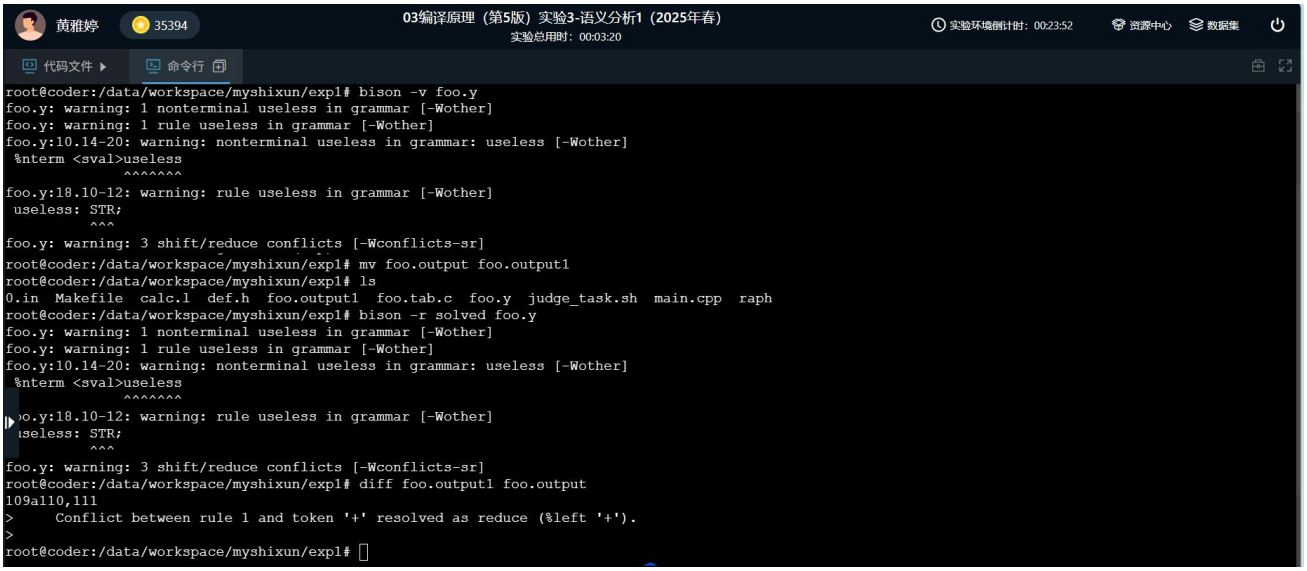

图 3-2 引入部分输出截图

问题 2 解释状态 5 中三行内容的含义（内容如图 3-3 所示）：

第一行显示当前处理的是规则 2，其内容为 “exp: exp '-' exp”。点号 “·” 标识当前解析位置，表明已完成 “exp '-'” 的归约，后续需归约下一个 exp。第二行指出，若下一个读入的符号为终结符 NUM，则执行移进操作，并转移至状态 1。第三行说明，若分析栈顶部为非终结符 exp，则转移至状态 7。

图 3-3 foo.output 中状态 5 输出截图

问题 3 根据使用-r solved 产生的文件，分析状态 6，7 中冲突的解决办法：

从图中对状态机的分析可知，状态 6 存在两处移进-归约冲突。第一处冲突位于（1）和（2）之间，由于预先设定了“+”为左结合属性，该矛盾得以化解；第二处冲突出现在（2）和（3）之间，因未明确“+”与 “-”的优先级关系，导致分析器无法判断是将 exp '+' exp 归约为 exp，还是移进 “-”符号，最终按照默认规则选择移进“-”。

状态 7 同样存在两组冲突：在（1）和（3）之间，因缺乏“+”与“-”的优先级定义，分析器无法抉择是将 exp '-' exp 归约为 exp，还是移进“+”，默认采取移进“+”的操作；（2）和（3）之间的冲突则源于“-”结合性的缺失，分析器在归约 exp '-' exp 和移进“-”之间选择了后者。

为消除这些冲突，对 foo.y 文件进行优化：移除冗余的非终结符 useless 和终结符 STR，并对语法规则进行调整。将“+”和“-”均设定为左结合，同时赋予“+”比“-”更高的优先级，通过上述修改，成功实现了状态机的无冲突解析。

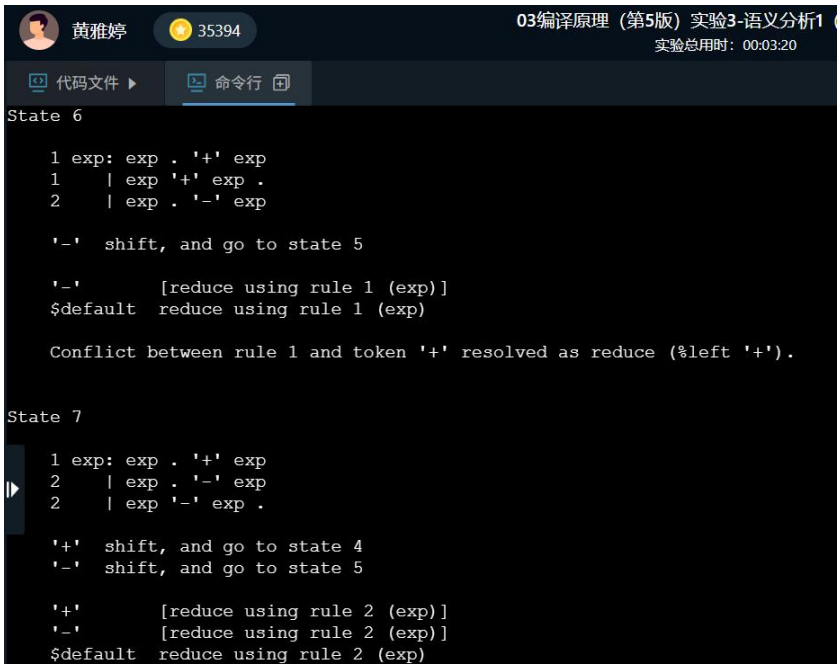

图 3-4 foo.output 中状态 6 和 7 输出截图

修改后的 foo.y 生成正确的语法分析器，在 OJ 平台进行测试验证通过，如下图所示。

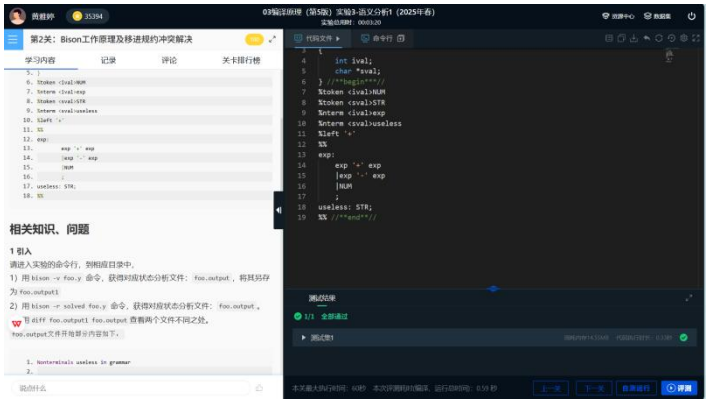

图 3-5 任务 2 测试结果

任务 3：Flex+Bison 联合实现中缀表达式计算器

使用 Flex 定义词法规则（如数字、运算符），Bison 定义语法规则和语义动作，计算中缀表达式的值。Makefile 自动化编译流程。在 OJ 平台进行测试验证如下图所示。

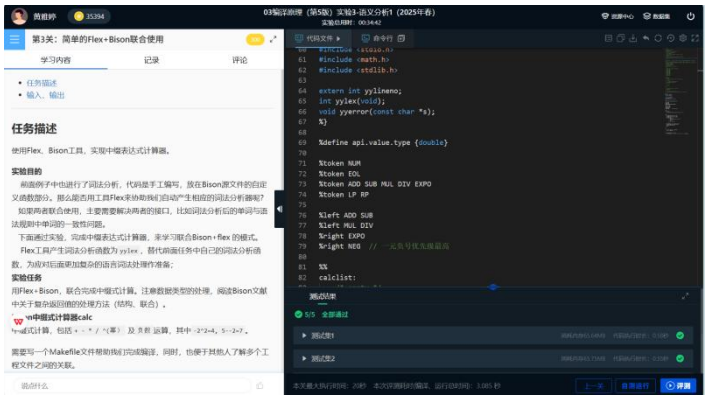

图 3-6 任务 3 测试结果

我也在本地环境尝试编译并运行了该程序，如图 3 - 7 所示。参考实验平台提供的测试集对程序进行测试，结果如图 3 - 8 所示，表明该程序能够通过 Flex 和 Bison 的联合使用，正确计算较为复杂的中缀表达式。

```
$ bison -d parser.y
$ flex token.l
$ gcc -o calc parser.tab.c lex.yy.c -lm -lfl
```

图 3-7 计算中缀表达式代码编译

```
1+3^2
Type(258):NUM Val=1
Type(260):ADD
Type(258):NUM Val=3
Type(264):EXPO
Type(258):NUM Val=2
=10
-5+3--1
Type(261):SUB
Type(258):NUM Val=5
Type(260):ADD
Type(258):NUM Val=3
Type(261):SUB
Type(261):SUB
Type(258):NUM Val=1
=-1
```

图 3-8 程序运行结果

任务 4：算术表达式、条件语句和循环语句的语义分析

使用 C 语言实现递归下降分析器，结合 new_label 和 add_quadruple 生成控制流四元式。parse_if 和 parse_while 处理条件和循环逻辑。在 OJ 平台进行测试验证如下图所示，全部通过。

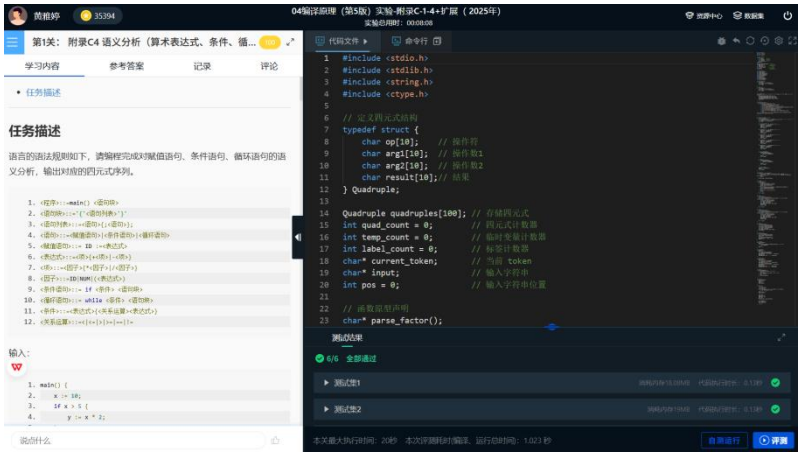

图 3-9 任务 4 测试结果

3.2.3 小结、设计存在的不足

在本次语义分析实验中，我收获了丰富且深入的知识与技能。通过手动编写程序，我清晰地理解了从语法树转化为四元式中间代码的生成过程，熟练掌握了控制流四元式的构造方法。与此同时，我还学会了将 Flex 与 Bison 联合使用。在这个过程中，我查阅了 Bison 和 Flex 手册，掌握了二者的接口设计和协作方式，成功实现了中缀表达式计算器。另外，通过对 foo.output 的分析，我理解了移进 - 归约冲突产生的原因，并通过优先级声明和文法简化的方法解决了冲突。

实验结果显示，无论是手动编写的程序还是由 Flex+Bison 生成的程序，都能够正确生成四元式序列。不过，Flex+Bison 在开发效率和扩展性方面表现得更为出色。

原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

已阅读并同意以下内容。

判定为不合格的一些情形：

（1） 请人代做或冒名顶替者；

（2） 替人做且不听劝告者；

（3） 实验报告内容抄袭或雷同者；

（4） 实验报告内容与实际实验内容不一致者；

（5） 实验代码抄袭者。

作者签名：

日期： **2025 年 06 月 08 日**