要　求

1、实验代码及报告为本人独立完成，内容真实。如发现抄袭，成绩无效；如果引用资料，需将资料列入报告末尾的参考文献，参考文献格式按华中科技大学本科毕业论文规范，并在正文中标注参考文献序号；

2、按编译原理实验任务，内容应包含：工具入门、词法分析、语法分析、语义分析及中间代码生成、目标代码生成；

3、报告中简单说明遇到的问题及解决问题的思路，特别是与众不同的、独特的部分；对设计实现中遇到的问题、解决进行记录；根据实验内容，结合能力训练的目标，对实验进行总结；完成自我评价；

4、评分标准：5 个主要实验环节按任务要求完成；采用的方法合适、设计合理；能体出研究能力、工具选择、工具开发、自主学习相关的能力；报告条理清晰、语句通顺、格式规范。

4.2 研究能力	5.2 工具选择	5.3 工具开发	12.2 自主学习	总分
25	40	10	25	100

目　录

一、实验过程记录

1.1 实验 1 工具入门

这个实验主要是入门 Flex 和 Bison 两个工具

第一关是尝试一下 flex，在定义段定义的变量可以在规则段使用，观察可知是一个统计行数和符号的功能，根据要求删掉++num_chars 即可，修改输出成功.

第二关没有本质区别，只是符号变复杂了，根据上课所学的正则表达式去匹配规则，在规则后面的{}中写匹配输出语句，是识别一个 Pascal-like toy 的语言的 token，修改成功后过关

第三关需要考虑 Flex 的匹配顺序，通过查阅手册可知，flex 尽可能匹配最长的串，如果仍然有歧义，则按规则给出的顺序匹配，例如在

```
a*b                 {printf("1");}
ca                  {printf("2");}
a*ca*               {printf("3");}
```

中，输入 abcaacacaaabbaaabcaaca

第一次匹配到 a*b，也就是 1

第二次匹配有歧义，ca 和 a*ca*均可匹配，而 ca 长度为 2，a*ca*匹配到 caa，长度为 3，所以第二次输出 3

以此类推，得到答案 132311132，即可过关

值得一提的是，一开始没明白到底要怎么给出答案，后面实验才给出要求直接填入 lab103.md

第四关是前三关的大综合，只不过非常繁琐，需要根据 PL 的单词符号表一个一个写正则表达式识别，同时会有一些边界情况，比如说 INT 需要考虑负数，ID 不能是数字开头等，然后实验样例中出现了表格中没有的字符，查看测试样例才知道需要加入 ERROR，基本方法就是排除掉边界情况再根据测试用例慢慢调整，就可以过关

第五关是 Bison 入门，计算逆波兰式，这题的关键是写出表达式，也就是

```
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;}
```

就可以过关

第六关是中缀式，我原以为跟第五关一模一样，只是改变一下表达式顺序，评测时错了才发现，逆波兰式没有歧义，而中缀式是有优先级的，不能一直从左边计算. 于是按照课上写的算法，通过添加一个中间符号解决优先级问题:

```
    exp: factor             { $$ = $1; }
  | exp ADD factor { $$ = $1 + $3; }
  | exp SUB factor { $$ = $1 - $3; }
  ;
```

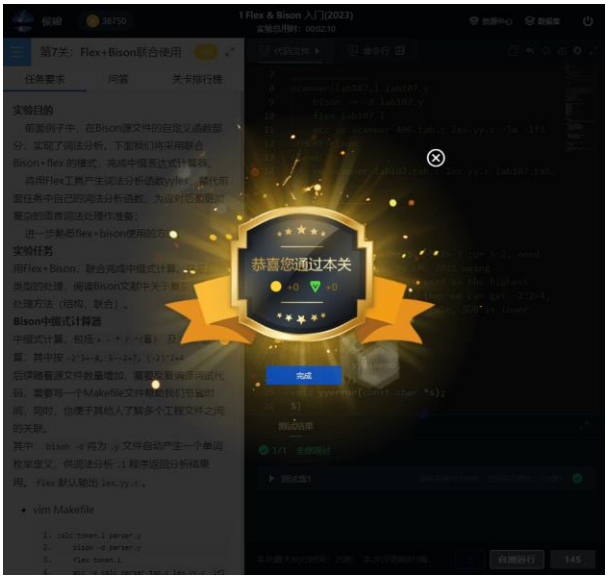
```
factor: term              { $$ = $1; }
   | factor MUL term { $$ = $1 * $3; }
   | factor DIV term { $$ = $1 / $3; }
   ;
```

第七关是 Flex + Bison 联合使用， 计算中缀式的值，不过 Flex 已经写好了，实验需要我们根据 lab107.l 修改 lab107.y

通过查看 lab107.l 中定义的符号结合第六关中缀式的结果就可以做出来. 需要注意的是数比第六关多了幂运算，核心代码如下:

```
exp:term      {$$=$1;}
     |exp ADD exp {$$=$1+$3;}
     |exp SUB exp {$$=$1-$3;}
     |exp MUL exp {$$=$1*$3;}
     |exp DIV exp {$$=$1/$3;}
     |error {}
     ;

term:NUM   {$$=$1;}
     |exp EXPO exp {$$=pow($1,$3);}
     |SUB NUM {$$=-$2;}
     |LP exp RP {$$=$2;}
     ;
```

最后完成实验，过关:



1.2 实验 2 词法分析

通过前面的工具入门，利用 Flex 和 Bison 完成词法分析部分

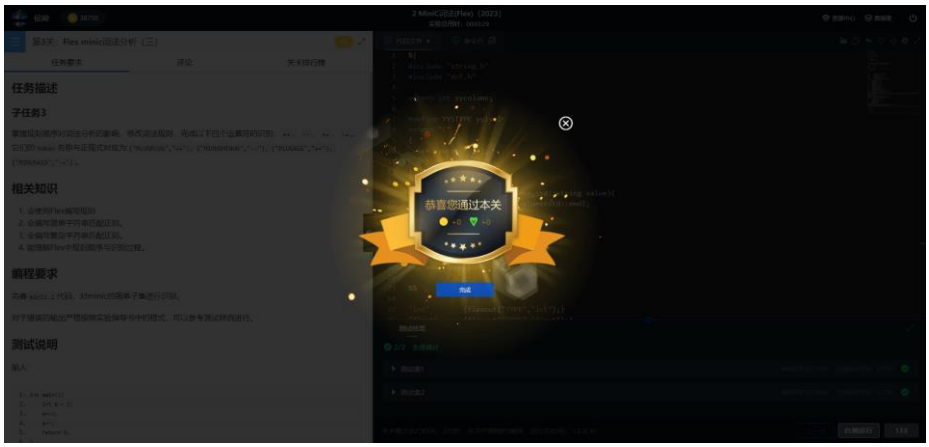
这个实验实际上跟实验 1 的 PL 语言(第四关)类似， 根据所给的 miniC 的单词表进行识别， 难度逐级提高

第一关照着样例编写即可， 没有特别困难的地方， 就是工程量比较大， 有些细节的地方需要注意

第二关开始有些难度了，主要是两种浮点数和八进制数十六进制数：

1. 一开始怎么也想不到把 1.2 和 1.05e5 这样的浮点数合起来匹配(合成一个规则)， 屡试屡错，突然灵光一闪发现可以分成两条规则， 但是都输出 FLOAT， 完全没必要合成一条， 一开始陷入了一条规则=一种 Token 的思维定势
2. 错误的八进制和正确的八进制的识别问题． 这个实际上是规则的顺序问题， 难点主要在于错误的八进制和正确的八进制， 经过尝试， 错误的八进制和十六进制必须放在正确的前面以排除掉不合理匹配， 与其限定正确的八进制， 不如思考错误的八进制是怎么样的， 这样才顺利解决把错误的八进制也识别成正确的八进制问题

第三关难度反而没有那么难， 因为经过第二个识别的经验， 对于++ --的识别只是规则顺序问题，完成规则后顺利通过：



1.3 实验 3 语法分析及语法树生成

这个实验是 miniC 的语法分析和语法树部分，主要是使用 Bison

第一关尝试解决冲突，状态 6 是移进-规约冲突，通过默认左结合解决冲突，状态 7 也是移进-规约冲突，同样通过左结合解决

所以这关需要告诉 Bison “-“的左结合性，同时删除不必要的%token <sval>STR 和%nterm <sval>useless 终结符，以及 useless: STR;无用规则，就可以过关

第二关写语法规则，需要参考 Appendix_A.pdf 的语法部分，工作量大但不难
不过评测的时候发现模运算并没有支持，所以删去了 token.l 的模运算

```
g++  -o parser.o parser.cpp  -
lpthread -ldl -c
flex -o token.cpp token.l
g++  -o token.o token.cpp  -
lpthread -ldl -c
token.l: In function 'int yylex()':
token.l:53:9: error: 'MODASS' was
not declared in this scope
 "%="   {return MODASS;/*模运算可能
后期有点难，先不支持*/}
         ^
Makefile:14: recipe for target
'token.o' failed
make: *** [token.o] Error 1
```

然后通关

第三关是解决第二关的规则冲突，通过查看 Bison 的输出发现主要是运算符的结合性问题，添加了：

```
//由低到高的定义优先级
%right ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS
%left STAR DIV
%right MINUS NOT
%left LP RP LB RB DOT

%nonassoc LOWER_THEN_ELSE
%nonassoc ELSE
```

即可过关

第四关填补规则匹配后的语句，在语义动作部分，调用构建语法树的支撑函数，用语法制导方法，为每个语法单元构建语法子树。只需要修改 parse.y 文件即可过关.

第五关主要是补全 astnode.cpp 的内容以便于打出 AST，例如：

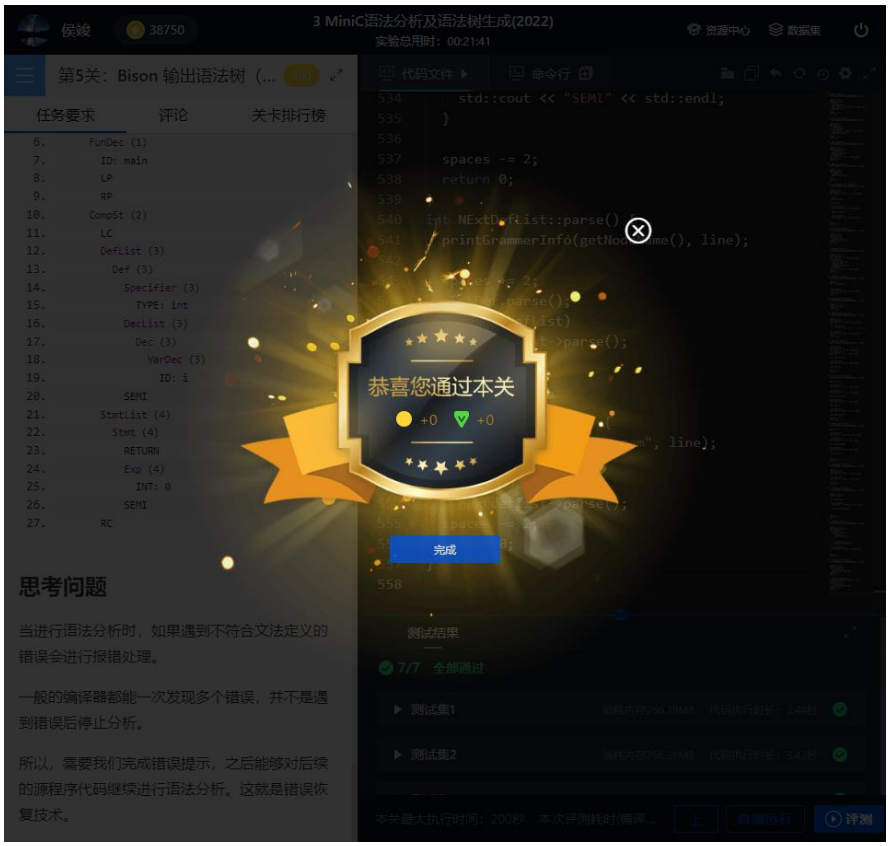
```
int NFloat::parse() {

   return 0;
}
```

补全：

```
int NFloat::parse() {
   printGrammerInfo(getNodeName(), line);
   spaces += 2;
   printspaces();
   std::cout << "FLOAT"
                  << ": " << value << std::endl;
   spaces -= 2;
```

```
    return 0;
}
```

实际上将会打印出 ASTNode 的类型和值，以此类推，补全所有的 Node 即可，因为骨架已经搭好，剩下的只是需要根据不同的 Node 输出而已，完成实验 3:



1.4 实验 4 语义分析及中间代码生成

实验 4 是语义分析，用中间语言描述语言的语义，包括语义检查等。前面已经完成了 AST 的生成，希望将 AST 翻译为 LLVM IR，完成之后，再借助 LLVM 的强大功能，操纵 IR 代码，进行优化、目标代码生成等。

第一关是两个简单的任务:

1. 修改 LLVM IR 使其输出 HUSTCSE

阅读任务可知关键在于 begin 前面的两行，也就是

```
    %3 = load i32, i32* %2, align 4
    %4 = call i32 @putchar(i32 %3)
```

输出了 H 字符，那么剩下的字符就以此类推即可，代码:

```
define dso_local i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    %5 = alloca i32, align 4
    %6 = alloca i32, align 4
```

```
    %7 = alloca i32, align 4
    store i32 72, i32* %1, align 4
    store i32 85, i32* %2, align 4
    store i32 83, i32* %3, align 4
    store i32 84, i32* %4, align 4
    store i32 67, i32* %5, align 4
    store i32 83, i32* %6, align 4
    store i32 69, i32* %7, align 4
    %8 = load i32, i32* %1, align 4
    %9 = call i32 @putchar(i32 %8)
    %10 = load i32, i32* %2, align 4
    %11 = call i32 @putchar(i32 %10)
    %12 = load i32, i32* %3, align 4
    %13 = call i32 @putchar(i32 %12)
    %14 = load i32, i32* %4, align 4
    %15 = call i32 @putchar(i32 %14)
    %16 = load i32, i32* %5, align 4
    %17 = call i32 @putchar(i32 %16)
    %18 = load i32, i32* %6, align 4
    %19 = call i32 @putchar(i32 %18)
    %20 = load i32, i32* %7, align 4
    %21 = call i32 @putchar(i32 %20)
    ret i32 0
}
```

2. 任务 2 是 if-else 的语句翻译, 变量存在%2 中, 只需要阅读前面的 LLVM 很容易就能知道怎么写:

```
define dso_local i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    store i32 89, i32* %1, align 4
    store i32 78, i32* %2, align 4
    %4 = load i32, i32* %1, align 4
    %5 = load i32, i32* %2, align 4
    %6 = call i32 @getchar()
    store i32 %6, i32* %3, align 4
    %7 = icmp eq i32 %6, 97
    br i1 %7, label %IfEqual, label %IfUnequal
IfEqual:
    %8 = call i32 @putchar(i32 %4)
    ret i32 %8
IfUnequal:
    %9 = call i32 @putchar(i32 %5)
    ret i32 %9
    ret i32 0
}
```

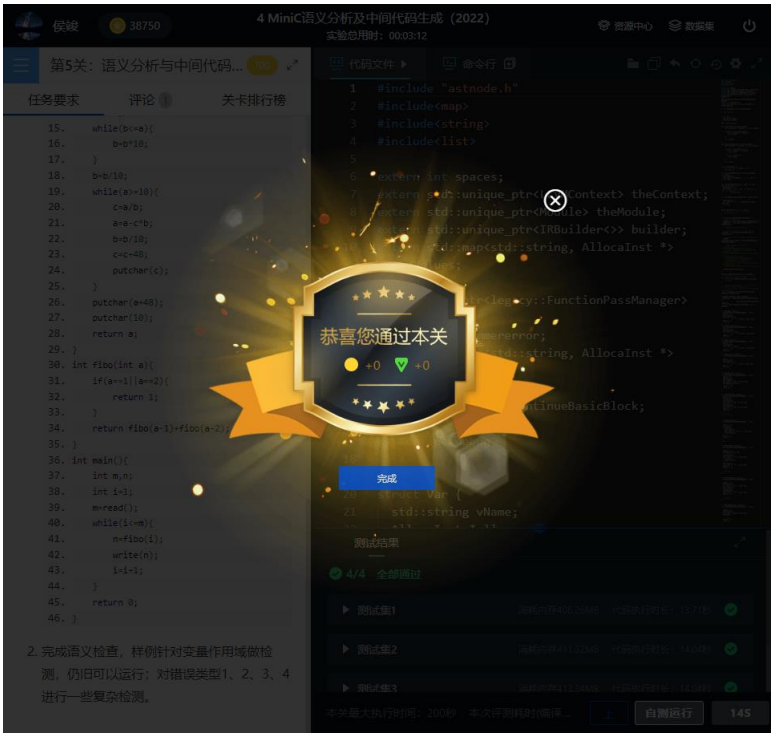
然后过关:

第二关是利用 API 翻译出刚刚的 LLVM IR 语言，由于实验并没有对 LLVM IR 进行限定，只需要结果正确，所以实际上找到 AST 到 LLVM IR 的 pattern，按顺序生成即可，过关.

第三关，第四关，第五关是第二关加强版，主要是补充 astnode.cpp 的代码生成函数，一方面要生成 IR 语言，另一方面还需要解决 8 种错误，所以定义了一个 printSemanticError(int type, int line, info)函数来确定八种错误，在生成 LLVM IR 的时候顺便判断错误，例如在生成变量的时候:

```
Value *NIdentifier::codegen() {
    Var *var = search(name);
    if(var != nullptr) {
        return builder->CreateLoad(var->alloc->getAllocatedType(),var->alloc);
    } else {
        printSemanticError(1,line);
        exit(0);
    }
}
```

错误 1 是”变量在使用时未经定义。”, search 函数是找到变量作用域，如果做不到就打印错误 1，其他的 Node 以此类推，补充完全部的 Node 之后过关



1.5 实验 5 目标代码生成

第一关分几个任务

任务一按照实验步骤，得到 IR 代码过关:

```
; ModuleID = 'test'
source_filename = "test"
declare i32 @putchar(i32)
declare i32 @getchar()
define i32 @calc(i32 %k, i32 %t) {
entry:
    br label %ret
```

```
ret:                                              ; preds = %entry
  %0 = mul i32 %k, %t
  ret i32 %0
}
define i32 @main() {
entry:
  %0 = call i32 @getchar()
  %1 = sub i32 %0, 48
  %2 = call i32 @getchar()
  %3 = sub i32 %2, 48
  br label %xzc1

xzc1:                                             ; preds = %xzc2, %entry
  %target.0 = phi i32 [ 1, %entry ], [ %6, %xzc2 ]
  %i.0 = phi i32 [ 0, %entry ], [ %5, %xzc2 ]
  %4 = icmp slt i32 %i.0, %3
  br i1 %4, label %xzc2, label %xzc3

xzc2:                                             ; preds = %xzc1
  %5 = add i32 %i.0, 1
  %6 = mul i32 %target.0, %1
  br label %xzc1

xzc3:                                             ; preds = %xzc1
  %7 = add i32 %target.0, 48
  %8 = call i32 @putchar(i32 %7)
  br label %ret

ret:                                              ; preds = %xzc3
  ret i32 0
}
```

此外通过对比优化前后的代码，发现内存操作的代码减少了许多

任务二是学习一下 opt 工具的使用就可以过关，实际上是把优化的几个命令写出一个 shell 脚本:

```
opt -mem2reg test.txt -S>u.txt
llc test.txt -o test.s
clang test.s -o test
```

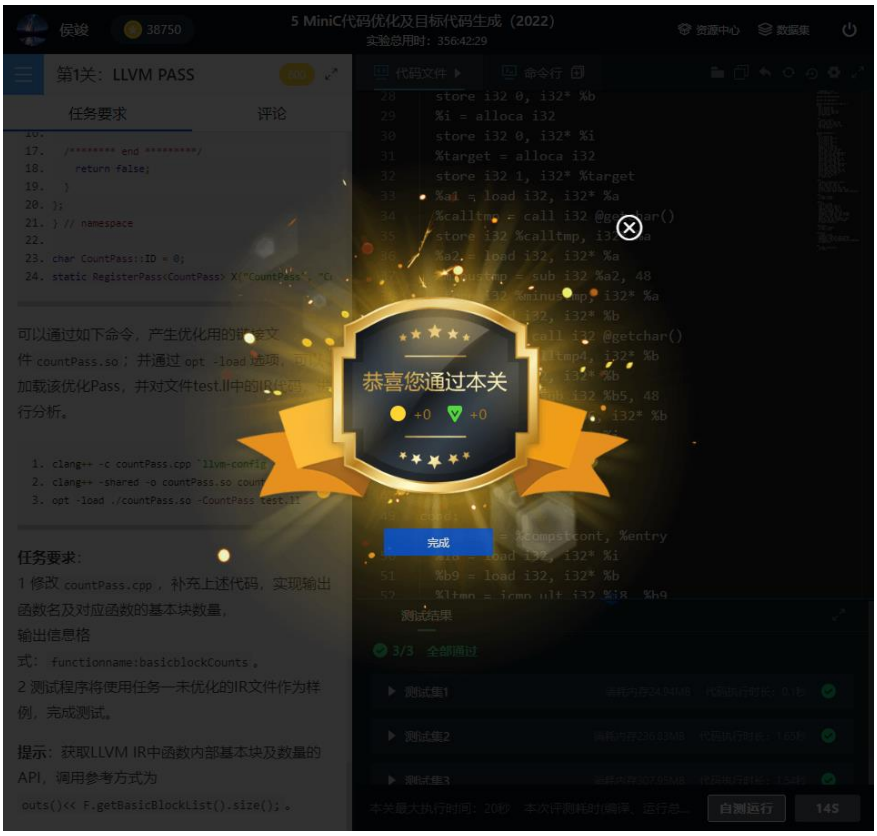
任务三是自定义优化，通过查阅资料, F.getBasicBlockList().size()可以得到基本块数量，因此把

```
outs().write_escaped(F.getName()) << '\n';
```

改成

```
outs().write_escaped(F.getName()) << ':'<< F.getBasicBlockList().size()<<'\n';
```

就完成了目标，过关:



二、实验心得

4.2（25 分）：

在本次编译原理实验中，我通过头哥课程的学习，分别完成了工具入门，词法分析，语法分析，中间代码生成(LLVM IR)，目标代码优化五个实验，其中有许多小实验带我一级一级通关，工具入门的 Flex 和 Bison 为后面的每一步都奠定了基础，词法分析利用 Flex 分析 token；再到基于 Bison 的语法分析和语法树生成，为每一个 Node 补全::parse()代码生成语法树；再到中间代码生成，先学习 LLVM IR，然后补全每一个 Node 的::codegen()代码来生成中间语言；最后以一个简单的优化任务结束了实验. 正好对应了课程学习的每个章节. 实验是通过自底向上设计来完成的，在骨架上做实验，也让整个实验难度小了很多，让编写的思路变得清晰.

5.2（40 分）：

在实验中，我选择的是 Flex、Bison 和 LLVM 这些现代工具进行设计、预测、模拟与实现。局限性分析：通过查阅资料发现，Flex 适合一些语言特性不那么强的语言，作为 miniC 这样的 C 语言原型来说 Flex 是适合的，但像 C++或者 Python 这样的语言是手工编写词法分析代码，而不是借助 Flex 这样的工具，是因为 Flex 的语法支持可能不够强(因为是上下文无关文法)，对于这些有着丰富特性的语言来说(比如说 Python 运行时的动态特性)，手写是比较好的选择；此外，如果语言经常需要增加新特性的话，手工代码也更易于控制一些，而且编译器前端所需的更复杂错误分析，手写代码也更容易实现.

5.3（10 分）：

在编译原理实验中，我完成了一个 C 语言子集 miniC，一个简单的编译器原型。作为一个原型，通过横向对比其他语言，可以看到许多的不足，主要是工程上的：

1. 不支持类型推断：现代编程语言大多已经支持类型推断(c++的 auto, go, rust)，这样可以让代码简洁很多，同样，也不支持泛型
2. 代码优化：miniC 的优化相当有限
3. 缺少高级特性：包括泛型，预定义宏，异步，接口，指针等特性

4. 没有包管理和引入包机制: 没有标准库, 同时也没有引入其他包的语法

12.2（25 分）：

在实验过程中，我阅读了大量的英文文档和网站资料，通过关卡自学了 Flex、Bison 和 LLVM 的高级功能。对于不同的部分可以查阅资料, 通过搜索引擎和相应的英文文档解决问题. 通过简单的实践和调试，我快速掌握了这些工具的基础和一些高级特性。在解决编译器构造过程中遇到的具体问题时，使我能够独立思考并应用所学知识找到解决方案。

这次实验不仅提高了我的专业技能，还锻炼了我的自主学习能力，这对我的职业发展大有裨益。

三、实验目标达成度的自我评价

通过实验，结合前面实验心得中的内容，在下面的表格中，完成自我评价。

毕业目标	自我评价的具体内容	目标达成的满意度 自评 ☑标记
4.2 能够基于科学原理和方法，根据需求选择路线，设计方案；	在实验过程中，包括词法、语法、语义分析、中间代码生成、目标代码优化过程中，需要根据课程中学习的编译理论课的原理和方法，确定合理的完成实验的路线，设计方案。包括：语言语法结构的取舍；选择分析路径，自下而上分析/自上而下分析；为了完成最终的代码生成，是否拟定/跟随完成了前序的学习任务（工具及相应语言的学习）等。	☑非常满意 ☐满意 ☐一般 ☐不满意
5.2 选择、使用现代工具设计、预测、模拟与实现，分析局限；	实验中，要求使用现代工具，如新的词法工具 Flex、语法工具 Bison 及中间代码框架 LLVM 的内容，完成实验的设计、实现；对相应工具实现时的局限性，进行适当的分析；甚至拟定出进一步完善的方案或方向。	☑非常满意 ☐满意 ☐一般 ☐不满意
5.3 开发满足特定需求的现代工具，分析其局限	编译原理实验中，引入了一个 C 语言的子集，并进行了相对快速的编译实验，本质上，为今后开发领域语言或者代码优化、代码分析工作，进行了准备。能对实现的简单编译器原型，进行局限性分析。	☑非常满意 ☐满意 ☐一般 ☐不满意
12.2 获取和职业发展需要的自主学习的能力，并表现出相应的成效。	通过阅读实验任务给的资料（英文 Flex、Bison、LLVM 网站）及自行搜索、整理、归纳互联网上的资源，见参考文献列表及相关文献在正文中的合适引用；通过完成铺垫关卡任务，快速掌握三种语言的基本功能；对于给出的工程问题（编译器构造）中，利用学习收获的知识，设计出相应的方案并实现其原型，从而获取了和职业发展需要的自主学习能力。	☑非常满意 ☐满意 ☐一般 ☐不满意

四、实验建议（可选）

参考文献

[1] 刘铭，徐兰芳， 骆婷. 编译原理（第 4 版）. 北京：电子工业出版社，2018

[2] Alfred V. Aho,Monica S.Lam, Ravi Sethi etc. 编译原理. 北京：机械工业出版社，2009

[3] Flex, version 2.5

[4] Appendix A C—语言文法

[5] 语法分析自动生成工具 Bison 3.7.6