华中科技大学

编译原理实验课程报告

题目：MiniC 语言编译器设计与实现

专业班级：网安 2104 班

学　　号：U20212131

姓　　名：邬雪菲

指导教师：刘铭

报告日期：2023 年 12 月 16 日

网络空间安全学院

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 Flex & Bison 工具入门

本环节实验的目标主要是初步掌握词法分析工具 Flex 和 Bison 的使用方法。Flex 是产生词法分析器的工具。词法分析程序是识别文本中单词模式的源程序。Bison 是产生语法分析器的工具。两种语言的结构类似，都包含定义段、规则段和用户子程序段，实验重点是掌握规则段的书写方式。

1.1.1 任务 1 Flex 入门

主要是理解 Flex 的语言框架，明白三个段各自的作用和基本编写规范。在预定义段声明统计变量，同时在规则段编写模式对应的动作代码，最后在程序段输出结果即可，较为简单，故不贴图。

1.1.2 任务 2 Flex toy 语言识别补全

本关进一步对 Flex 规则段的书写进行练习。Flex 规则段包括一系列匹配模式和动作，模式一般使用正则表达式书写，动作部分为 c 代码：模式 1｛动作 1(C 代码)｝。这一关要求识别标识符 ID、整数、浮点数，特别注意整数和小数的正则表达式的书写，尤其是小数，需要识别如 123.或.12 之类的特殊情况，但是又不能将单独的.识别为小数。下面第一张图的规则虽能通过样例，但实际上需要修改为图二的规则才能正确识别所有数字。

```
4 DIGIT [0-9]*
5 ID [a-z][a-z0-9]*
6 %%
7 {DIGIT}               {printf( "An integer: %s (%d)\n", yytext,atoi( yytext ) );}
8 {DIGIT}+"."{DIGIT}    {printf( "A float: %s (%g)\n", yytext,atof( yytext ) );}
```

```
[+-]?[0-9]+                     {printf( "An integer: %s (%d)\n", yytext,atoi( yytext ) );}
[+-]?([0-9]*\.?[0-9]+|[0-9]+\.)     {printf( "A float: %s (%g)\n", yytext,atof( yytext ) );}
```

1.1.3 任务 3 Flex 规则匹配顺序

本关主要是探究 Flex 的规则匹配顺序。在本地虚拟机上运行，交换规则顺序后尝试编译，虽然能编译成功但是会出现警告，多次尝试后发现，只要前面的正则表达式包含后面出现的表达式，在识别后面的规则时就会发出警告。同时可以观察到更改规则顺序后词法分析的结果改变。三条规则的六种不同先后顺序的分析结果如下：

```
[12/15/23]seed@VM:~/.../task103$ flex -o lab103-2.c lab103-2.l
lab103-2.l:3: warning, rule cannot be matched
[12/15/23]seed@VM:~/.../task103$ gcc -o lab103-2 lab103-2.c -lfl
[12/15/23]seed@VM:~/.../task103$ ./lab103-2 < case0.in
133311133[12/15/23]seed@VM:~/.../task103$ ./lab103 < case0.in
132311132[12/15/23]seed@VM:~/.../task103$
```

ab {printf("1");} ca {printf("2");} a*ca* {printf("3");}	132311132
ab {printf("1");} a*ca* {printf("3");} ca {printf("2");}	133311133
ca {printf("2");} ab {printf("1");} a*ca* {printf("3");}	132311132
ca {printf("2");} a*ca* {printf("3");} a*b {printf("1");}	132311132
a*ca* {printf("3");} ca {printf("2");} a*b {printf("1");}	133311133
a*ca* {printf("3");} a*b {printf("1");} ca {printf("2");}	133311133

结合资料分析，Flex 会尽量匹配长的字符串，且会靠前的规则优先级较高。也就是说当分析串能

匹配到规则时并不会马上执行动作清空分析栈，而是会继续往下读取字符，检查更长串是否有规则匹配，如果有则继续往下分析，如果没有则选择可匹配规则中最前面的一条进行匹配。

1.1.4 任务 4 Flex 补全 PL 语言

进一步练习规则编写。难点是整常量、标识符、字符常量以及未定义字符的识别。其他的较简单。

```
INTCON          [\-]?[1-9][0-9]*|0
IDENT           [A-Za-z][A-Za-z0-9]*
CHARCON         '[^\']*'
```

特别注意使用反斜杠表示转义字符。从同学那学到一个小技巧，考虑到任务 3 中学到的规则优先匹配顺序，可以直接在规则段最末加上一条.规则替换直接枚举未定义字符的规则如下：

```
[\~\!\@\#\$\%\^\&\_\|] {printf("%s: ERROR\n", yytext);}
%% /*** end ***/
```

改为

```
.  {printf("%s: ERROR\n", yytext);}
%% /*** end ***/
```

两者在本题的数据集上都是可行的，但修改过后的规则显然更优，因为不需要为具体的非法字符定义规则，灵活性和可移植性好。

1.1.5 任务 5 Bison 入门之逆波兰式计算

通过前几天的实验已经初步掌握用 Flex 构建一个词法分析器，但还需要 Bison 语言帮助构建语法分析器。Bison 和 Flex 的结构相似，但在规则段使用的是 BNF 范式来描述产生式。本关重点学习 Bison 规则段的编写，完成逆波兰式的识别和计算，较为基础。

1.1.6 任务 6 Bison 中缀表达式

本关需要实现中缀表达式的识别和计算，难点在于如何处理不同优先级的运算符。我借鉴了理论学习中消除文法二义性的方式来处理优先级不同的问题，也就是通过增设非终结符和对应的产生式达到消除二义性实现优先级区别的目的，由此实现先乘除后加减的中缀表达式的识别与计算。

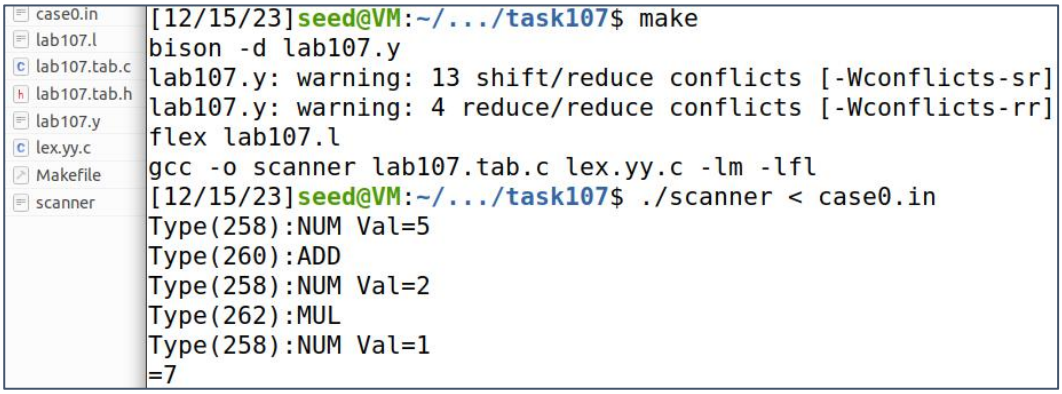
```
%%
  /* Grammar rules and actions follow.  */
 /* begin */
calclist:
        %empty
        |calclist exp EOL {printf("=%.10g\n",$2);}
exp:term
    |exp ADD term {$$=$1+$3;}
    |exp SUB term {$$=$1-$3;}
    ;

term:NUM
    |term MUL NUM {$$=$1*$3;}
    |term DIV NUM {$$=$1/$3;}
    ;
 /* end */
%%
```

1.1.7 任务 7 Flex Bison 联合使用

本关需要 Flex 和 Bison 配合使用。重点是掌握 Makefile 文件的写法，理解各个工程文件之间的关系。bison -d 将为.y 文件自动产生一个单词枚举定义，供词法分析.l 程序返回分析结果用。flex 默认输出 lex.yy.c。注意 Flex 和 Bison 联合使用时，需要在.y 中的%token 后罗列出所有终结符的种类码标识符。

```
scanner:lab107.l lab107.y
        bison -d lab107.y
        flex lab107.l
        gcc -o scanner lab107.tab.c lex.yy.c -lm -lfl
.PHONY:clean
clean:
        rm scanner lab107.tab.c lex.yy.c lab107.tab.h
```

执行 make 操作，可以观察到目录下生成.c 和.h 后缀文件，以及可执行的分析程序。



1.2 MiniC 词法

本章主要利用 Flex 语言实现对 MiniC 语言的词法分析。

1.2.1 任务 1

本关要求完成对 MiniC 语言部分单词的识别规则的编写，较为简单。

1.2.2 任务 2

本关扩增了需要识别的 MiniC 单词子集，增加了保留关键字，简单浮点数如 1.2 和 1.05e5，八进制、十进制、十六进制整数等。同时，还要求实现一定程度的容错功能：识别非法八进制如 08、非法十六进制数字如 0xGF2。

```
-?[1-9][0-9]*|0                 {flexout("INT",yytext);/*十进制整数*/}
-?[0-7]+                        {flexout("INT",yytext);/*八进制整数*/}
0[xX][0-9a-fA-F]+               {flexout("INT",yytext);/*十六进制整数*/}

-?([1-9][0-9]*|0)\.[0-9]*       {flexout("FLOAT",yytext);}
-?([1-9][0-9]*|0)?\.[0-9]*[eE][+-]?[0-9]+ {flexout("FLOAT",yytext);}
[a-zA-Z][a-zA-Z0-9]*            {flexout("ID",yytext);}

[\n]                            {yycolumn=1;}
[ \r\t]                         {/*printf("过滤空格等字符\n");*/}

.                               {printf("Error type A at Line %d: Mysterious characters '%c'\
-?0[0-9]*                       {printf("Error type A at Line %d: Illegal octal number '%s'\r
-?0[xX][a-zA-Z0-9]*             {printf("Error type A at Line %d: Illegal hexadecimal number
```

1.2.3 任务 3

本关要求掌握规则顺序对词法分析的影响，修改词法规则，完成四个运算符的识别：++，--，+=，-=。 由于 flex 的最长串优先匹配规则，如果待识别的语言是标准的 MiniC 语言，变更诸如+、++、+=几种符号的规则的顺序并不会影响识别。

1.3 MiniC 语法分析及语法树生成

本章要求学习语法分析识别程序的编写方法。采用语法制导的方法，完成语法树的输出。与理论知识的关系紧密，需要运用移进归约法的思想理解 Bison 语法分析的原理，梳理清楚分析状态的确定和转移过程，有一定难度。

1.3.1 任务 1 Bison 工作原理及移进归约冲突解决

本关要求了解 Bison 工具原理并对有问题的语法规则进行修改。Bison 采用自底向上的分析方法，有移进和归约两种动作。阅读 Bison 英文文档之后大致了解了 Bison 的工作流程。在调用 Bison 编译文件时可以使用--verbase 或--report 参数产生状态分析文本文件.output。令--report=solved，Bison 会在报告中标注已解决的冲突，使用 diff 命令比较得到如下结果：

```
[12/16/23]seed@VM:~/.../output$ diff foo.output foo.output1
111,112d110
<     Conflict between rule 1 and token '+' resolved as reduce (%left '+').
<
```

进一步查看状态分析报告文件，理解状态 5 对应内容的含义：

```
State 5

    2 exp: exp '-' . exp

    NUM  shift, and go to state 1

    exp  go to state 7
```

第一行：状态 5 对应规则 2 的项目 exp: exp ’-’ . exp 及其闭包；第二行：展望符为 NUM 时移进，并转移到状态 1；第三行：遇到 exp 时，转移到状态 7，继续分析项目 2。

```
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 '+').
```

分析使用-r solved 时产生.output 文件中的状态 6、7 内容。状态 6 遇到-时，产生“移进-规约”冲突，Bison 系统通过优先采用规则前者，优先移进的方法解决了该冲突（方括号括起的不会被采用）；遇到+时也产生“移进-规约”冲突，而 Bison 系统定义了+的左结合性，优先规约，冲突解决。

```
State 7

    1 exp: exp . '+' exp
    2    | exp . '-' exp
    2    | exp '-' exp .

    '+'  shift, and go to state 4
    '-'  shift, and go to state 5

    '+'       [reduce using rule 2 (exp)]
    '-'       [reduce using rule 2 (exp)]
    $default  reduce using rule 2 (exp)
```

状态 7 遇到-时，产生“移进-规约”冲突，系统通过优先采用规则前者，优先移进解决了冲突；遇到+时，产生“移进-规约”冲突，系统优先采用规则前者，优先移进，冲突解决。

在本题中，我通过增加运算符-的左结合性定义，且规定-的优先级低于+的方式尝试消除冲突。修改过后的状态分析文件如下：

```
State 6

    1 exp: exp . '+' exp
    2    | exp . '-' exp
    2    | exp '-' exp .

    '+'  shift, and go to state 5

    $default  reduce using rule 2 (exp)

    Conflict between rule 2 and token '-' resolved as reduce (%left '-').
    Conflict between rule 2 and token '+' resolved as shift ('-' < '+').


State 7

    1 exp: exp . '+' exp
    1    | exp '+' exp .
    2    | exp . '-' exp

    $default  reduce using rule 1 (exp)

    Conflict between rule 1 and token '-' resolved as reduce ('-' < '+').
    Conflict between rule 1 and token '+' resolved as reduce (%left '+').
```

可见，原本的移进归约冲突因为算符左结合性以及优先级的设置都得到了解决。因为在 Bison 中，声明的顺序决定了优先级，越后声明的优先级越高。当 Bison 遇到移进/规约冲突时，它将查询优先级表，通过先归约优先级高的符号解决冲突。

1.3.2 任务 2 Bison 语法规则构造

本关需要词法分析将单词信息传递到语法规则中，按照最左归约的顺序打印出归约过程使用的非终结符名称。Bison 中默认采用 LALR 分析法，会在每条产生式匹配后，执行对应的语义动作，可在此时输出该条产生式左边的非终结符。重点是理解 Bison 分析过程以及 Bison 结构。

Bison 语法规则段照着 MiniC 语言附录给出的文法规则参考写就好，只需要补充输出左端非终结符操作即可。以及需要理解代码中的预定义部分含义，union 结构指定了符号的语义值类型。

```
%union {
    int *p;
    std::string *text;
    int    type_int;
    float  type_float;
    int    type_char;
    char   type_id[32];
}
//union的默认结构体类型为YYSTYPE，相当于自己把YYSTYPE重新定义为union类型。所以相应的词法分析中yylval也变为union类型。
//这个union类型-d选项编译时会放在头文件中
//% type  用于定义非终结符的语义值类型
%type  <p> program ExtDefList ExtDef Specifier ExtDecList DecList VarDec FunDec CompSt DefList VarList ParamDec Dec Def StmtList Exp Stmt Args
OptTag  Tag StructSpecifier
//% token 用于定义终结符的语义值类型
%token <type_int> INT                   //指定INT的语义值是type_int，由词法分析得到的数值
%token <type_id> ID RELOP TYPE STRUCT   //指定ID,RELOP 的语义值是type_id，由词法分析得到的标识符字符串mmecpy得到的
%token <type_float> FLOAT               //指定ID的语义值是type_float，由词法分析得到的float
%token <type_char> CHAR
%token LP RP LC RC LB RB SEMI COMMA     //用bison对该文件编译时，带参数-d，生成的exp.tab.h中给这些单词进行编码，可在lex.l中包含parser.tab.h使用这些单词种类码
%token DOT PLUS MINUS STAR DIV MOD ASSIGNOP AND OR NOT IF BREAK ELSE WHILE RETURN PLUSASS MINUSASS STARASS DIVASS MODASS PLUSPLUS MINUSMINUS
//由低到高的定义优先级
```

```
%%
/*High-level Definitions*/ /*Begin*/
program: ExtDefList {std::cout<<"Program"<<std::endl;} /*显示规则对应非终结符*/
        ;

ExtDefList: ExtDef ExtDefList {std::cout<<"ExtDefList"<<std::endl;}
          | /*empty*/ {std::cout<<"ExtDefList"<<std::endl;}
          ;

ExtDef: Specifier ExtDecList SEMI {std::cout<<"ExtDef"<<std::endl;}
      | Specifier SEMI {std::cout<<"ExtDef"<<std::endl;}
      | Specifier FunDec CompSt {std::cout<<"ExtDef"<<std::endl;}
      ;
```

1.3.3 任务 3 Bison 冲突状态解决

本关目标是通过算符优先级定义实现移进归约冲突的消除，以及理解各个工程文件的关联。

```
//由低到高的定义优先级
%right ASSIGNOP PLUSASS MINUSASS STARASS DIVASS MODASS
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right NOT UMINUS PLUSPLUS MINUSMINUS
%left LP RP LB RB DOT

%nonassoc LOWER_THEN_ELSE
%nonassoc ELSE
```

1.3.4 任务 4 Bison 输出语法树

本关要求用语法制导方法，为每个语法单元构建语法子树，同时理解教学提供的代码结构。makefile 用于环境变量准备和编写编译指令；token.l 是词法分析文件，为了辅助完成语法树的构造输出，需要记录行号，可以通过重定义 lex 提供的 YY_USER_ACTION 宏为一个函数，获得 yylloc 结构信息，包括行号、列号，用来定位出错位置。也可以直接使用 yylineno 获取当前行号，将数值存入 yylval.line 中，这样 Bison 可读取 yylval 之中的值；parser.y 是语法分析文件，为了构造语法树，使用面向对象的方法对每个语法对象构造对应的语义分析方法，具体来说，为每条规则左边的非终结符定义一个类，存储其生成的子树，并定义对应的输出方法。本关任务量较大，需要耐心阅读给出的 astnode 参考程序，理解面向对象的编程方式。

```
StmtList: {$$=nullptr;}
        | Stmt StmtList  {$$=new NStmtList(*$1,$2); $$->line=$1->line;}
        ;
Stmt:     Exp SEMI {$$=new NExpStmt(*$1); $$->line=$1->line;}
        | CompSt {$$=new NCompStStmt(*$1);} //复合语句结点直接最为语句结点，不再生成新的结点
        | RETURN Exp SEMI {$$=new NRetutnStmt(*$2); $$->line=$1;}
```

1.3.5 任务 5 Bison 语法树构造

本关需要补全 astnode.cpp 完成语法单元对应的语法子树的支撑，同时在 parser.y 中完成其他子树的构造。重点是将每个语法成分抽象为类，为每个类实现 parse 函数，根据语义输出类的语法树。浏览 astnode.h 将其中的类与 parser.y 对应，根据 main.cpp 中的在 astnode.cpp 中补充语法节点的 parse 函数，在 parse.y 中修改 Expressions 部分的语法动作。

```
int NChar::parse() {
  printGrammerInfo(getNodeName(), line);
  spaces += 2;
  printspaces();
  std::cout << "CHAR"
            << ": " << value << std::endl;
  spaces -= 2;
  return 0;
}
```

```
177 /*Expressions*/
178 Exp:      Exp ASSIGNOP Exp  {$$=new NAssignment("ASSIGNOP",*$1,ASSIGNOP,*$3);$$->line=yylineno;}//$$结点type_id空置未用，正好存放运算符
179         | Exp PLUSASS Exp   {$$=new NAssignment("PLUSASS",*$1,PLUSASS,*$3);$$->line=yylineno;}//复合赋值运算
180         | Exp MINUSASS Exp  {$$=new NAssignment("MINUSASS",*$1,MINUSASS,*$3);$$->line=yylineno;}
181         | Exp STARASS Exp   {$$=new NAssignment("STARASS",*$1,STARASS,*$3);$$->line=yylineno;}
182         | Exp DIVASS Exp    {$$=new NAssignment("DIVASS",*$1,DIVASS,*$3);$$->line=yylineno;}
183
184         | PLUSPLUS Exp %prec UPLUSPLUS   {$$=new NSingleOperator("PLUSPLUS-",PLUSPLUS,*$2);$$->line=yylineno;}//这里利用BISON %prec表示和UMINUS同优先级
185         | MINUSMINUS Exp %prec UMINUSMINUS   {$$=new NSingleOperator("MINUSMINUS-",MINUSMINUS,*$2);$$->line=yylineno;}//这里利用BISON %prec表示和UMINUS同优先级
186         | Exp PLUSPLUS   {$$=new NSingleOperator("-PLUSPLUS",PLUSPLUS,*$1);$$->line=yylineno;}//这里利用BISON %prec表示和UMINUS同优先级
187         | Exp MINUSMINUS  {$$=new NSingleOperator("-MINUSMINUS",MINUSMINUS,*$1);$$->line=yylineno;}//这里利用BISON %prec表示和UMINUS同优先级
```

1.4 MiniC 语义分析及中间代码生成

本章内容是学习 LLVM IR，这是一种中间代码表示。

1.4.1 任务 1 LLVM IR 初识

本关要求初步掌握 LLVM IR 中间代码的格式和书写方式，实现字符串的输入输出以及实现条件判断分支，内容较为基础。LLVM IR 有许多与汇编代码相似的地方，可以比较学习。

```
[12/16/23]seed@VM:~/.../task401$ lli ./task1.ll
HUSTCSE
[12/16/23]seed@VM:~/.../task401$ lli ./task2.ll
a
Y
[12/16/23]seed@VM:~/.../task401$ lli ./task2.ll
b
N
```

1.4.2 任务 2 LLVM IR API

本关要求使用 LLVM 提供的 API 实现 MiniC 翻译为 LLVM IR。在前三章的工作中，我已根据语法规则为不同的语义类型定义了不同的类，在 main.cpp 的 main 函数中使用 p->parse()完成语法树的打印。下一步要调用 InitializeModuleAndPassManager()初始化函数初始化 LLVM 的配置，为几个 LLVM 的全局变量构建实例。然后 p->codegen()生成中间代码输出到文件中，这个过程需要使用之前建立好的 AST 语法树。难点在于 LLVM API 的使用，需要手工翻译上一关的两个程序。

```
 96  //根据输入的单字符，判断，如果是'a'，则输出'Y'，否则输出'N'。//设置返回类型
 97  //begin
 98  Type *retType = Type::getInt32Ty(*theContext);
 99  FunctionType *mainType = FunctionType::get(retType, false);
100  Function *mainFunc = Function::Create(mainType, Function::ExternalLinkage, "main", theModule.get());
101  BasicBlock *bb = BasicBlock::Create(*theContext, "entry", mainFunc);
102  BasicBlock *thenBB = BasicBlock::Create(*theContext, "then", mainFunc);
103  BasicBlock *elseBB = BasicBlock::Create(*theContext, "else", mainFunc);
104  BasicBlock *returnBB = BasicBlock::Create(*theContext, "return", mainFunc);
105  builder->SetInsertPoint(bb);  // entry:
106  // %a_get = call i32 @getchar()
107  Value *a_get = builder->CreateCall(getFunc,std::vector<Value *>{}, "a_get");
108  // %a_eq = icmp eq i32 %a_get, 97('a')
109  Value *a_eq = builder->CreateICmpEQ(a_get, builder->getInt32('a'), "a_eq");
110  builder->CreateCondBr(a_eq, thenBB, elseBB);// br %a_eq, label %then, label %else
111  builder->SetInsertPoint(thenBB);  // then:
112  builder->CreateCall(putFunc, {builder->getInt32('Y')});// call i32 @putchar(i32 89('Y'))
113  builder->CreateBr(returnBB);// br label %return
114  builder->SetInsertPoint(elseBB);// else:
115  builder->CreateCall(putFunc, {builder->getInt32('N')});// call i32 @putchar(i32 78('N'))
116  builder->CreateBr(returnBB);// br label %return
117  builder->SetInsertPoint(returnBB);// return :
118  // end
119  //设置返回值
120  builder->CreateRet(builder->getInt32(0));
121  verifyFunction(*mainFunc);
```

1.4.3 任务 3 语义分析与中间代码生成（一）

本关需要利用 LLVM IR 来描述 MiniC 的语义。通过为 AST 语法树节点 codegen 方法，完成对应语法成分的语义表达，并完成语义检查。任务是补充 astnode.cpp 文件中节点的 codegen()函数定义。

语义检查是通过以下几个部分辅助实现的。预定义部分 extern std::map<std::string, AllocaInst *> namedValues;是用来存当前函数局部变量，extern std::map<std::string, AllocaInst *> curNamedValues;存当前所有变量，Function *getFunction(std::string Name)用来查找函数，extern int grammererror;是语义错误码。当发现语义错误时，程序调用

printSemanticError 函数打印错误信息。举个例子，当出现未声明变量使用错误时，程序通过 AllocaInst *var=namedValues[name]试图找到给定的 name 变量失败，返回 NULL 值，由此可判断出现了未声明变量的语义错误，调用错误函数打印相关信息。

执行 make 操作生成 minic 可执行程序，测试用例观察到，如果代码语义正确，minic 可正常生成 LLVM IR 代码，如果出现语义错误也可以识别并打印错误信息。

```
Function *func = theModule->getFunction(id.name);
if(!func)
  printSemanticError(2, line, "Undeclared function " + id.name);
```

```
AllocaInst *leftVar = namedValues[lhs.name];
if(!leftVar)
  printSemanticError(1, line, "Undeclared variable " + lhs.name);
```

```
if (theModule->getFunction(Id.name)) {
  printSemanticError(4, line, "Redefined " + Id.name);
  return nullptr;
}
```

```
Type *varType = nSpecifier.getType();
for (NDecList *item = nDecList; item; item = item->nDecList) {
  std::string &varName = item->dec.vardec.Id.name;
  // check if it exists the same name of var
  if(curNamedValues.find(varName)!=curNamedValues.end())
    printSemanticError(3, line, "Redefined " + varName);

  // create var
  AllocaInst *allocaVar = builder->CreateAlloca(varType, nullptr, varName);
  if(item->dec.exp != nullptr){
    Value *initVal = item->dec.exp->codegen();
    builder->CreateStore(initVal, allocaVar);
  }
  // add var to namedValues
  namedValues[varName] = allocaVar;
  curNamedValues[varName] = allocaVar;
}
```

```
[12/16/23]seed@VM:~/.../task403$ ./minic 0.in
; ModuleID = 'test'
source_filename = "test"

declare i32 @putchar(i32)

declare i32 @getchar()

define i32 @main() {
entry:
  %b = alloca i32
  %a = alloca i32
  %c = alloca i32
  %d = alloca i32
  store i32 0, i32* %d
  store i32 0, i32* %b
  store i32 2, i32* %c
  %0 = load i32, i32* %b
  %1 = load i32, i32* %c
  %2 = add i32 %0, %1
  store i32 %2, i32* %a
  br label %ret

ret:
  %3 = load i32, i32* %a
  ret i32 %3
}
[12/16/23]seed@VM:~/.../task403$ ./minic 1.in
Error type 1 at Line 3.
[12/16/23]seed@VM:~/.../task403$ ./minic 2.in
Error type 2 at Line 3.
[12/16/23]seed@VM:~/.../task403$ ./minic 3.in
Error type 3 at Line 3.
```

本地测试时注意到此时 MiniC 的编译过程耗时较长，因为需要链接到 LLVM 的对应库。

1.4.4 任务 4 语义分析与中间代码生成（二）

本关在上一关的基础上继续实现更多类型的语义错误检查。

```
Function *func = theModule->getFunction(id.name);
if(!func)
  printSemanticError(2, line, "Undeclared function " + id.name);

std::vector<Value *> argV;
NArgs *arg = nargs;
for(auto &argI : func->args()) {
  if(!arg)
    printSemanticError(8, line, "Too few arguments in function " + id.name);
  Value *argVal = arg->codegen();
  if(argVal->getType() != argI.getType())
    printSemanticError(8, line, "Type mismatched for argument " + argI.getName().str() + " in function " + id.name);
  argV.push_back(argVal);
  arg = arg->nArgs;
}
if(arg)
  printSemanticError(8, line, "Too many arguments in function " + id.name);

return builder->CreateCall(func, argV);
```

```
Value * rightVal = rhs.codegen();
if(lhs.name == "")
  printSemanticError(6, line, "The left-hand side of an assignment must be a variable");
AllocaInst *leftVar = namedValues[lhs.name];
if(!leftVar)
  printSemanticError(1, line, "Undeclared variable " + lhs.name);
if(leftVar->getAllocatedType() != rightVal->getType())
  printSemanticError(5, line, "Type mismatch in assignment");
```

1.4.5 任务 5 语义分析与中间代码生成（三）

本关仍然是对之前操作的加强检查，在此不赘述。

1.5 MiniC 代码优化及目标代码生成

前面的章节借助 LLVM 提供的框架，完成了开发语言的前端部分。本章要求使用 LLVM 代码优化框架及命令行，调用流程 PASS 进行特定优化并输出最终程序，完成后端部分。LLVM 的后端，由代码生成分析器和变换流程 PASS 组成，这些流程将 LLVM IR 转换为目标代码或汇编代码。优化过程也通过流程完成，流程大致分为分析类、变换类、其它工具类。

1.5.1 任务 1 编译器中调用 LLVM 支持的优化函数

InitializeModuleAndPassManager()函数是用于初始化 LLVM 配置的，它创建了一个新的 LLVM 上下文 theContext，并用此生成了新模块 test，创建新的 IR 构建器与新的流程 PASS 管理对象实例 theFPM。本关要求使用 theFPM 加入指定的优化流程（属于变换类 Transform Passes）。对比优化前后产生的中间代码，可见内存操作指令的数量明显减少。

```
[12/16/23]seed@VM:~/.../lab5$ diff 1.ll 2.ll
10,13d9
<   %t2 = alloca i32
<   %k1 = alloca i32
<   store i32 %k, i32* %k1
<   store i32 %t, i32* %t2
17,20c13,14
<   %0 = load i32, i32* %k1
<   %1 = load i32, i32* %t2
<   %2 = mul i32 %0, %1
<   ret i32 %2
---
>   %0 = mul i32 %k, %t
>   ret i32 %0
25,29d18
<   %a = alloca i32
<   %b = alloca i32
<   %i = alloca i32
<   %target = alloca i32
<   store i32 1, i32* %target
31,33c20
<   store i32 %0, i32* %a
<   %1 = sub i32* %a, i32 48
<   store i32* %1, i32* %a
---
>   %1 = sub i32 %0, 48
```

```
35,42c22,23
<   store i32 %2, i32* %b
<   %3 = sub i32* %b, i32 48
<   store i32* %3, i32* %b
<   store i32 0, i32* %i
<   %4 = load i32, i32* %target
<   %5 = add i32 %4, 48
<   %6 = call i32 @putchar(i32 %5)
<   br label %ret
---
>   %3 = sub i32 %2, 48
>   br label %cond
44c25,29
< cond:                                           ; No predecessors!
---
> cond:                                           ; preds = %run, %entry
>   %target.0 = phi i32 [ 1, %entry ], [ %6, %run ]
>   %i.0 = phi i32 [ 0, %entry ], [ %5, %run ]
>   %4 = icmp slt i32 %i.0, %3
>   br i1 %4, label %run, label %after
46c31,41
< ret:                                            ; preds = %entry
---
> run:                                            ; preds = %cond
>   %5 = add i32 %i.0, 1
>   %6 = mul i32 %target.0, %1
>   br label %cond
>
> after:                                          ; preds = %cond
>   %7 = add i32 %target.0, 48
>   %8 = call i32 @putchar(i32 %7)
>   br label %ret
>
> ret:                                            ; preds = %after
```

1.5.2 任务 2 LLVM 命令行完成优化及目标代码生成

本关要求使用 opt 命令行对 IR 代码进行优化。使用 opt -h 查看命令行参数，--mem2reg 参数实现 Promote Memory to Register

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

1.5.3 任务 3 自定义优化函数

我们还可自行构造 PASS，从而实现对生成 IR 的各种不同程度的变形。本关要求补充 countPass.cpp 文件实现输出 IR 中所有函数名的功能如下所示：

```cpp
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct CountPass : public FunctionPass {
  static char ID;
  CountPass() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override {
    outs() << F.getName() << ':'<<F.getBasicBlockList().size() << '\n';
    return false;
  }
};
} // namespace

char CountPass::ID = 0;
static RegisterPass<CountPass> X("CountPass", "Count Pass", false, false);
```

二、实验心得

4.2：在本次实验过程中，我先是完成了对应理论部分的学习和巩固，再开启具体部分的实验，理论辅导了实践而实践又很好地充实验证了理论，两者配合较好。在词法和语法分析部分我掌握得较好，由于在理论上花费了许多时间钻研具体的分析方法，加上在真正动手做实验前对 Flex 和 Bison 工具的资料进行了仔细的研读学习，对于 Bison 的自下而上移进归约工作流程理解得比较清晰，实验进行得很顺利。但到语义分析部分就逐渐有些吃力了，首先是理论部分高度抽象，理解起来比较困难，其次这部分的理论知识和实验并没有很好的衔接，而且代码量陡增，面向对象的编程方式也恰好是我不擅长的，导致后半部分花了很长时间去尝试理解示例代码并仿写。不过最终还是艰难完成了。

5.2：本次实验中的工具选择和使用是重中之重。Flex 和 Bison 语言简单易上手，且与理论知识联系紧密，无论是正则表达式的编写还是 BNF 范式的编写，亦或是对移进归约的自下而上分析流程的理解，都是对理论学习的检验和巩固。且两者的文档给出了大量的示例和说明，非常有助于我这样的初学者快速理解和学习使用。但是 LLVM 编译器框架则不那么容易上手，虽然本实验只是借助 LLVM IR 实现中间代码生成，但是由于 LLVM 本身知识体系庞大，其复杂性很容易让初学者感到迷茫和困惑，同时 LLVM 生成的代码也比较冗长，阅读起来较为困难。实验中虽然借助了 API 函数完成翻译，但是全部手工操作代码量还是稍微大了些，希望以后能开发出自动化工具能根据构建好的语义分析代码生成对应的 IR 翻译器，至少也要简化 LLVM 的某些接口和函数操作，提高使用的便携性和可理解性。

5.3：编译原理实验主要完成了一个 MiniC 语言编译器，这不仅仅是实现了 C 语言子集的简单编译器原型，更是对现代工具的应用和开发进行了探索，让我初步接触了语言开发、代码生成优化等领域。虽然实现的这个编译器功能十分局限，只能对 MiniC 语言进行分析，且依赖 Flex、Bison 语言进行词法语法分析，套用 LLVM 框架才最终实现了编译器原型，总体上基本都是在借助工具，并没有尝试着从零开始完成一个小型编译器，非常多实践的细节其实都被忽略了。不过以我现在对理论的掌握

程度和代码编写水平，完全独立地编写一个编译器也不现实。目前的实验方案已经让我对编译器的理解更深刻了，这已经足够了，更深更细致的研究就留给未来的自己吧。

12.2：在自主学习方面，得益于老师提供的详细的资料，加上英文阅读对我来说挑战并不算很大，所以整个过程也比较顺利。通过阅读官方文档，以及得益于编译原理理论知识的学习，我可以快速了解上手 Flex 和 Bison 这两种语言，并应用到实验中。而对于复杂的 LLVM 框架我也能做到耐心阅读大量的代码并尝试理解，在再进行简单应用。本次实验比较友好的一点就是环境配置方面，不需要耗费大量的时间在虚拟机配置上，三门语言的安装和使用流程都很简单，本地实验过程很顺利。偶尔遇到问题时，在联网搜索和与同学讨论后也基本都能得到解决方案。

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

这次实验内容循序渐进，且与理论知识的学习过程相联系，由简单到复杂来实现了一个小型编译器的构建。总体难度合适，提供的资料丰富详实，且前三部分与理论知识高度配合，使得完成实验的体验良好。而且实验的环境较容易在本地搭建，使得学生的重心更多地放在实验工具学习和代码编写的重点上。且头歌平台也给出了大量的示例供参考，报错信息比较合理容易理解，统一的评分标准也方便学生发现错误并修正。

但是少部分实验如语法分析和语义分析的后半部分难度较大，代码量大，参考资料的说明晦涩难懂，且 LLVM 大框架本身就很复杂，虽然只是借用框架实现部分功能，但是由于缺乏对整个体系的理解，加上这部分实验要求完成的任务需要的能力和理论知识关联几乎没有，会导致学生在尝试完成的过程中难以推进任务，建议这部分减少任务量，或者加入更多提示，或者考虑换成和理论部分更贴近的实验模块。

参考文献

[1] 许畅，陈嘉，朱晓瑞. 《编译原理实践与指导教程》.机械工业出版社，2015

[2] 刘铭，徐兰芳，骆婷. 《编译原理（第 4 版）》.电子工业出版社，2018

[3] https://blog.csdn.net/weixin_44007632/article/details/108666375

[4] https://blog.csdn.net/weixin_46222091/article/details/105990745

[5] https://www.zhihu.com/column/c_1267851596689457152

[6] https://llvm.org/docs/LangRef.html