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. Flex & Bison 工具入门

（1）实验内容

Lex 是 Lexical Compiler 的缩写，是 Unix 环境下非常著名的工具，主要功能是生成一个词法分析器(scanner)的 C 源码，描述规则采用正则表达式(regular expression)。Flex(The Fast Lexical Analyzer)是 GNU/Linux 下的 lex 版本。Bison（GNU Bison）是一个用于生成语法分析器的工具。它是根据 Yacc（Yet Another Compiler Compiler）开发的，用于解析和分析上下文无关文法。本实验通过渐进的方式，逐步熟悉 flex 和 Bison，最后采用联合 Bison+flex 的模式，完成中缀表达式计算器。

（2）实验过程

Task1: Flex 首次尝试

编写规则，匹配字符/n，则行数+1；匹配其他字符，则字符数+1，如图 1-1。

```
%%
\n {++num_lines;}
. {++num_chars;}
%%
```

图 1-1 匹配规则示意图

思考题答案:

1. 关键字有:if、then、begin、end、procedure、function。
2. 规则不能调换顺序。
3. 用大括号“{}”括起来的部分为注释，词法分析中处理注释后没有输出。
4. 空白符号:空格、制表符\t 和换行符\n。

Task2: 简单 Pascal-like toy 语言识别补全

利用 flex 语法规则编写对应规则，实现对整数，小数，部分关键字，标识符，操作符号的识别，输出不能识别的符号，具体代码如图 1-2。

```
DIGIT [0-9]
ID [a-z][a-z0-9]*
%%
{DIGIT}*                {printf( "An integer: %s (%d)\n",
yytext,atoi( yytext ) );}
{DIGIT}+"."{DIGIT}*     {printf( "A float: %s (%g)\n",
yytext,atof( yytext ) );
}
if|then|begin|end|procedure|function {printf( "A keyword:
%s\n", yytext );}
{ID} printf( "An identifier: %s\n", yytext );
"+"|"-"|"*"|"/"         printf( "An operator: %s\n",
yytext );
"{"[^}\n]*"}"   /* eat up one-line comments */
[ \t\n]+                /* eat up whitespace */
.       printf( "Unrecognized character: %s\n", yytext );
```

图 1-2 匹配规则

Task3: Flex 规则匹配顺序

```
%%
a*b             {printf("1");}
ca              {printf("2");}
a*ca*           {printf("3");}
%%
int main()
{
        yylex();
}
```

输入符号串为

```
abcaacacaaabbaaabcaaca
```

图 1-3 测试程序和输入

了解 flex 规则在匹配时，如果多条规则都匹配成功，程序是如何处理的，对应测试程序和输入如图 1-3。

查询资料得知，如果多条规则都匹配成功，yylex 会选择匹配长度最长的那条规则，如果有匹配长度相等的规则，则选择排在最前面的规则。

Task4：Flex 词法规则补全（PL 语言）

根据 PL 语言单词（如图 1-4）编写对应的匹配规则，如图 1-5。

PL 语言单词符号及其种别值

单词符号	种别枚举值	单词符号	种别枚举值
标识符	IDENT	]	RBRACK
整常量	INTCON	(	LPAREN
字符常量	CHARCON	)	RPAREN
+	PLUS	,	COMMA
-	MINUS	;	SEMICOLON
*	TIMES	.	PERIOD
/	DIVSYM	:=	BECOME
=	EQL	:	COLON
<>	NEQ	begin	BEGINSYM
<	LSS	end	ENDSYM
<=	LEQ	if	IFSYM
>	GTR	then	THENSYM
>=	GEQ	else	ELSESYM
of	OFSYM	while	WHILESYM
array	ARRAYSYM	do	DOSYM
program	PROGRAMSYM	call	CALLSYM
mod	MODSYM	const	CONSTSYM
and	ANDSYM	type	TYPESYM
or	ORSYM	var	VARSYM
not	NOTSYM	procedure	PROCSYM
[	LBRACK		

图 1-4 PL 语言单词符号及其种别值

```
%%
of              {printf("%s: OFSYM\n", yytext);}
array           {printf("%s: ARRAYSYM\n", yytext);}
program         {printf("%s: PROGRAMSYM\n", yytext);}
mod             {printf("%s: MODSYM\n", yytext);}
and             {printf("%s: ANDSYM\n", yytext);}
or              {printf("%s: ORSYM\n", yytext);}
not             {printf("%s: NOTSYM\n", yytext);}
begin           {printf("%s: BEGINSYM\n", yytext);}
end             {printf("%s: ENDSYM\n", yytext);}
if              {printf("%s: IFSYM\n", yytext);}
then            {printf("%s: THENSYM\n", yytext);}
else            {printf("%s: ELSESYM\n", yytext);}
while           {printf("%s: WHILESYM\n", yytext);}
do              {printf("%s: DOSYM\n", yytext);}
call            {printf("%s: CALLSYM\n", yytext);}
const           {printf("%s: CONSTSYM\n", yytext);}
type            {printf("%s: TYPESYM\n", yytext);}
var             {printf("%s: VARSYM\n", yytext);}
procedure       {printf("%s: PROCSYM\n", yytext);}
{INTCON}        {printf("%s: INTCON\n", yytext);}
{IDENT}         {printf("%s: IDENT\n", yytext);}
{PLUS}          {printf("%s: PLUS\n", yytext);}
{MINUS}         {printf("%s: MINUS\n", yytext);}
{TIMES}         {printf("%s: TIMES\n", yytext);}
{DIVSYM}        {printf("%s: DIVSYM\n", yytext);}
{BECOME}        {printf("%s: BECOME\n", yytext);}
{CHARCON}       {printf("%s: CHARCON\n", yytext);}
```

图 1-5 匹配规则（部分）

Task5：Bison 入门（逆波兰式计算）

Bison 中使用的是 BNF 范式来描述产生式，故编写对应规则的 BNF 范式即可，如图 1-6。

```
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;}
```

图 1-6 语法规则

Task6：Bison 入门（中缀式计算）

同 Task5，编写对应规则即可，如图 1-7。

```
calclist:
    %empty
    |calclist exp EOL {printf("=%.10g\n",$2);}
exp:term         { $$ = $1; }
  | exp ADD term { $$ = $1 + $3; }
  | exp SUB term { $$ = $1 - $3; }
    ;

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

图 1-7 语法规则

Task7：Flex+Bison 联合使用

编写对应语法规则即可，如图 1-8。

```
%%
calclist:
    %empty
    |calclist exp EOL {printf("=%.10g\n",$2);}

exp:term
    |exp ADD exp {$$=$1+$3;}
    |exp SUB exp {$$=$1-$3;}
    |exp EXPO exp {$$=pow($1,$3);}
    |SUB term    {$$=-$2;}
    |LP exp RP {$$=$2;}
    |exp MUL exp {$$=$1*$3;}
    |exp DIV exp {$$=$1/$3;}
    |error {}
    ;

term:NUM
    ;
%%
```

图 1-8 语法规则

（3）实验总结及遇到的问题

这个实验的任务基本都是引导性质的，我们首先熟悉了 Flex 源程序的语法规则，学会了如何用正则表达式进行简单的词法分析（如 Task4 的 PL 语言的词法分析），后面又学习了 Bison 的基本语法，使用 Bison 进行简单的语法分析，最终将两者结合实现简单的计算器功能。

整体实验不难，但在完成途中也颇有收获：flex 规则多条规则都匹配成功，yylex 会选择匹配长度最长的那条规则，如果有匹配长度相等的规则，则选择排在最前面的规则；task4 中对于字符常量的匹配属实精妙，'([^']*)'，深深感受到了正则表达式的简便之处；task5 中一时脑抽，忘记了 pow 这个函数，还以为自带^可以表示指数，卡了一小会儿。

2. MiniC 词法(Flex)

（1）实验内容

在基本掌握了前面的 Flex 与 Bison 工具后，利用两种工具，逐步完成对 Mini-C 语言的结构分析。

（2）实验过程

Task1: Flex minic 词法分析（一）

编写对应的匹配规则，如图 2-1 所示。

```
25  "int"         {flexout("TYPE","int");}
26  "float"       {flexout("TYPE","float");}
27  "char"       {flexout("TYPE","char");}
28  "if"         {flexout("IF","if");}
29  "else"         {flexout("ELSE","else");}
30  "struct"         {flexout("STRUCT","struct");}
31  "return"      {flexout("RETURN","return");}
32  "("       {flexout("LP","(");}
33  ")"       {flexout("RP",")");}
34  "{"       {flexout("LC","{");}
35  "["       {flexout("LB","[");}
36  "}"       {flexout("RC","}");}
37  "]"       {flexout("RB","]");}
38  "=="      {flexout("RELOP","==");}
39  "++"      {flexout("PLUSPLUS","++");}
40  "--"      {flexout("MINUSMINUS","--");}
41  "+="      {flexout("PLUSASS","+=");}
42  "-="      {flexout("MINUSASS","-=");}
43  [A-Za-z][A-Za-z0-9]*      {flexout("ID",yytext);}
44  "="       {flexout("ASSIGNOP","=");}
```

图 2-1 匹配规则

Task2: Flex minic 词法分析（二）

增加了对保留关键字的测试;能够识别简单浮点数，例如 1.2，1.05e5， 八进制数、十六进制数等。同时，能做到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。

实现方法：编写对应匹配规则，如图 2-2 所示。

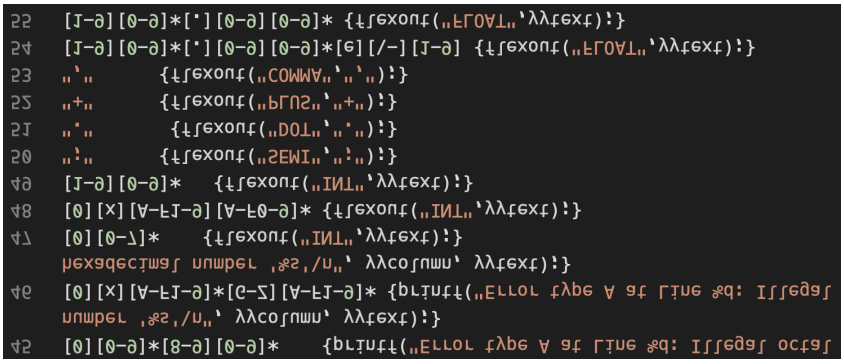


图 2-2 匹配规则

Task3: Flex minic 词法分析（三）

把握规则顺序对词法分析的影响，修改词法规则，完成以下四个运算符的识别：++，--，+=，-=。它们的 token 名称与正规式对应为("PLUSPLUS","++")、("MINUSMINUS","--")、("PLUSASS","+=")、("MINUSASS","-=")。

编写对应匹配规则，如图 2-3 所示。

```
38  "=="      {flexout("RELOP","==");}
39  "++"      {flexout("PLUSPLUS","++");}
40  "--"      {flexout("MINUSMINUS","--");}
41  "+="      {flexout("PLUSASS","+=");}
42  "-="      {flexout("MINUSASS","-=");}
```

图 2-3 匹配规则

（3）实验总结及遇到的问题

本次实验实现了对 minic 中的关键字、符号、整数、八进制和十六进制数的分析，有了第一次实验的基础，本次实验比较简单。唯一需要注意的是匹配过程的优先顺序是写在后面的规则，优先级更高。其实还有可以改进的地方：匹配规则有些许冗余。

3. MiniC 语法分析及语法树生成

（1）实验内容

进一步理解掌握 Bison 的工作原理，学习词法、语法分析识别程序的编写方法，使用 flex，完成 Mini-C 语言词法分析。采用语法制导的方法，完成语法树的输出

（2）实验过程

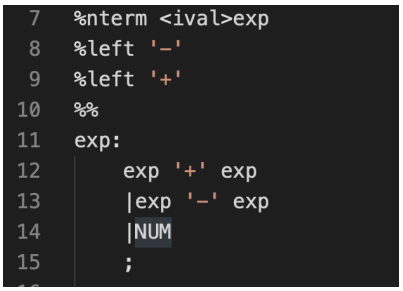
Task1: Bison 工作原理及移进规约冲突解决

回答思考题：

1.存在“移进-规约”冲突，Bison 解决了该冲突，是利用“+”的左结合性解决的；

2.存在“移进-规约”冲突，未解决冲突，未定义“-”相关的结合性。

实现方法：删除无用终结符 STR 和 useless。查看 foo.output 文件，发现未指定’-’的优先级和结合性。故按照“+”优先级高于“-”和“+”、“-”均服从左结合即可消除规约冲突，如图 3-1。



```
%nterm <ival>exp
%left '-'
%left '+'
%%
exp:
    exp '+' exp
    |exp '-' exp
    |NUM
    ;
```

图 3-1 优先级顺序

Task2: Bison 语法规则构造（一）语法规则构造

输入 Mini-C 的源文件，找到最左规约序列，判断其语法是否正确，输出错误信息，并按最左规约的顺序，打印出规约过程中使用的非终结符名称。

根据 Mini-C 语法，编写语法规则，输出非终结符名称。语法规则部分如图 3-2:

```
program: ExtDefList             {std::cout << "Program" << std::endl;}
    ;
ExtDefList:
    %empty                      {std::cout << "ExtDefList" << std::endl;}
    | ExtDef ExtDefList         {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;}
    ;
ExtDecList:
    VarDec                      {std::cout << "ExtDecList" << std::endl;}
    | VarDec COMMA ExtDecList   {std::cout << "ExtDecList" << std::endl;}
```

图 3-2 语法规则

Task3: Bison 语法规则构造（二）冲突状态解决

按附录中符号相应的优先级顺序编写程序即可，优先级顺序从上到下依次增高，如图 3-3.

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

%nonassoc ELSE
```

图 3-3 优先级顺序

通过添加以上语句声明 ELSE 不存在结合性，可以解决运算符冲突中的移进-规约冲突，从而解决绝大部分冲突问题。但当运行 bison -v 命令时，如果发现第 107 个状态仍然存在冲突，这通常是由于 if-else 语句导致的移进-规约冲突。简而言之，通过声明 ELSE 不存在结合性，我们可以解决这个冲突问题。

Task4: Bison 输出语法树（一）

本任务要求对 Mini-C 的简单样例进行分析，按实验指导书要求，补充部分语法动作完成简单的函数代码的语法树即可，输出其对应的语法树。

在学习 astnode.h、astnode.cpp 和 parser.y 这些文件后，我明白了在语法制导翻译中，通常会为语法中的每个非终结符定义一个对应的类（或称为 AST 节点类），并通过构造函数来表示语法规则中的生成式。这些类负责表示语法单元的结构，并在构造时处理语法制导翻译的逻辑。在 Bison 的语法文件（通常以 .y 结尾）中，每个生成式的右部描述了如何构造相应的语法单元，而左部则表示生成的语法单元。通过在生成式中调用相应的构造函数，可以构建语法树的节点，并按照规定的语法规则连接起来。通过这样的方式，整个语法树就能够得以构建。这种方法使得我们能够

通过在生成式中调用合适的构造函数，按照语法规则构建语法树的节点，实现了语法制导翻译的目标。这对于构建编译器前端部分，将源代码转换为中间表示或目标代码，具有重要的帮助作用。

查看 astnode.h 中的 ExtDeflist 的定义，如图 3-4

```
class NInteger : public NExpression {
public:
    int value;

    NInteger(int value) : value(value) {}

    int parse();

    int handle() { return 0; }
};
```

图 3-4 NInteger 定义

NExpression 类是继承自 Node 类的，而 NInteger 类则是继承自 NExpression 类的。在语法规则中，对于非终结符 Exp 的处理如下：

1.对于叶子节点，即遇到单词 INT，会调用 NInteger 类的构造函数，用 INT 的值作为参数，构造相应的节点。

2.然后，将词法分析器中用于记录行号的 yylino 变量的值赋给 NInteger 类的成员变量 line，用于保存行号。

3.这样，NInteger 类的实例在构造时获取了 INT 值作为参数，同时也记录了行号信息。

这是一种常见的在语法分析树中记录词法信息的方式，通过在节点类中添加成员变量来保存相关信息，以便后续在语义分析或其他处理阶段使用。

查看 astnode.h 中的 NAssignment 的定义，如图 3-5

```
class NAssignment : public NExpression {
public:
    int op;
    std::string name;
    NExpression &lhs;
    NExpression &rhs;

    NAssignment(NExpression &lhs, int op, NExpression &rhs)
        : lhs(lhs), op(op), rhs(rhs) {}

    NAssignment(std::string &name, NExpression &lhs, int op, NExpressio
        : name(name), lhs(lhs), op(op), rhs(rhs) {}

    int parse();
    int handle() { return 0; }
};
```

图 3-5 NAssignment 的定义

NAssignment 类继承自 NExpression 类，表示赋值操作。

1.有两个构造函数，允许处理两种不同情况的赋值操作：

第一个构造函数接受两个 NExpression 类型的引用 lhs 和 rhs，以及一个整数 op，用于构造赋值操作节点。

第二个构造函数接受一个 std::string 类型的引用 name，以及两个 NExpression 类型的引用 lhs 和 rhs，还有一个整数 op，用于构造带有名称的赋值操作节点。

2.成员变量包括一个整数 op 用于表示赋值操作的运算符，一个字符串 name 用于保存赋值操作的名称（如果有的话），以及两个 NExpression 类型的引用 lhs 和 rhs 分别表示赋值操作的左右表达式。

3.成员函数包括一个 parse 函数和一个 handle 函数，其中 parse 函数可能包含有关如何解析赋值操作的逻辑，而 handle 函数则返回整数 0，可能是一个默认的处理逻辑。

由此分析完这两个类之后，就弄懂了整套原理，便可以编写语法规则构建语法树如图 3-6

```
program:
    ExtDefList                  { p = new NProgram($1); if($1) p->line = $1->line; } /*显示语法树*/
    ;
ExtDefList:
    %empty                      { $$ = nullptr; }
    | ExtDef ExtDefList         { $$ = new NExtDefList(*$1, $2); $$->line = $1->line; }
    ;
ExtDef:
    Specifier ExtDecList SEMI   { $$ = new NExtDef(*$1, $2); $$->line = $1->line; }
    | Specifier SEMI            { $$ = new NExtDef(*$1); $$->line = $1->line; }
    | Specifier FunDec CompSt   { $$ = new NExtDef(*$1, $2, $3); $$->line = $1->line; }
    ;
ExtDecList:
    VarDec                      { $$ = new NExtDecList(*$1, nullptr); $$->line = $1->line; }
    | VarDec COMMA ExtDecList   { $$ = new NExtDecList(*$1, $3); $$->line = $1->line; }
    ;
```

图 3-6 语法规则

Task5：Bison 输出语法树（二）

在任务 4 的基础上，需要根据 astnode.h 中的定义补完 astnode.cpp 中对应的代码，该任务首先需要理解 astnode.cpp 中打印的方式和打印的对应控制。以 NDec 为例，其部分对应 cpp 代码如图 3-7

```cpp
int NDec::parse() {
    printGrammerInfo(getNodeName(), line);
    spaces += 2;
    vardec.parse();
    if (exp) {
        printspaces();
        std::cout << "ASSIGNOP" << std::endl;
        exp->parse();
    }
    spaces -= 2;
    return 0;
}
```

图 3-7 NDec 的 parse

首先，对于每个语法单元，调用 printGrammerInfo() 函数按要求打印相应的语法信息。然后，spaces 变量增加 2，相当于生成两个空格的缩进。在匹配语法规则之前，生成的一定是非终结符 specifier。就能通过递归调用 specifier.parse()函数进行语法树的遍历和 print。在判断语法规则之前，首先判断其后是否生成非终结符 fundec，如果是，则递归地调用 fundec.parse() 和 compst.parse() 进行语法树遍历和 print。否则，再判断其是否生成非终结符 NExtDecList，如果是同样进行递归调用。接着，将遇到终结符 SEMI，按规则先调用 printspaces() 函数打印相应缩进，然后打印对应的词法单元。其余的 .cpp 文件填充实现与该部分类似，因此在此不再赘述其实现流程。在构建完成 .cpp 文件后，再在 parser.y 中仿照任务 4 完成其余 Bison 执行语句即可。

（3）实验总结及遇到的问题

前三个任务比较简单，主要是关于在 Bison 中如何利用-v 参数对.y 文件生成状态分析文件从而分析并利用 Bison 的优先级/结合性等方法解决移进规约冲突。但后两个任务就比较难了，学习了利用 Bison 工具对给定的 MiniC 语法规则编写对应的语法分析程序，通过语法制导方法为语法单元构建语法树并完成语法树的输出，这两关需要自学 c++的面向对象的基本知识，包括继承，多态，构造函数，虚构函数等。 只要完成了第四个任务之后，第五个任务在实现思路上并没有过多的改变，主要是工作量比较大，要补全 cpp 代码并完善 parser.y 文件。

4．MiniC 语义分析及中间代码生成

（1）实验内容

掌握 LLVM IR 的基本使用，将 Mini-C 翻译成 LLVM IR;学习 LLVM ，提供的 API 和 AST 节点 codegen 方法，使用 AST 语法树和 LLVM IR API 完成语义分析和中间代码生成。

（2）实验过程

Task1：LLVM IR 初识

任务 1 是打印字符串 "HUSTCSE"，其中样例已经提供了如何打印字符 "H" 的示例代码。我将依照相同的方法，逐个字符地仿写打印剩余的字符串。

任务 2 是使用 icmp 和 br 指令来判断输入字符是否为 "a"，如果是则输出 "Y"，否则输出 "N"。为了完成这个任务，我将参考相关资料，仿写使用 icmp 和 br 指令的代码来实现输入字符的判断和输出结果。

Task2：LLVM IR API

本次任务是任务 1 的进一步扩展，需要利用 LLVM IR 提供的 API 函数来实现汇编 IR 文件的自动生成。

对于习题 1，我们可以参考已经给出的代码示例来仿写并修改，以实现打印字符串 "HUSTCSE" 的功能。需要注意的是，在返回值部分，我们需要新声明一个为 0 的常量作为返回值，这样程序在后续运行过程中才会显示正常退出。

对于习题 2，主要需要注意如何将 if-else 语句翻译成对应的 LLVM IR。我们可以参考学习资料中的 "LLVM 的接口函数使用举例" 部分的 "分支结构" 小节，根据其中的示例来实现。

综上所述，通过使用 LLVM IR 提供的 API 函数，我们可以完成手工翻译，并生成对应的汇编 IR 文件。通过平台的测试，我们可以验证任务的成功完成。

Task3: 语义分析与中间代码生成（一）

本关利用 LLVM IR 来描述 Mini-C 语言的语义。通过为 AST 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，在进行语义分析完善 codegen 具体实现时，需要参考 parser.y 和 astnode.h 文件，了解 codegen 函数需要分析的参数以及参数的处理方式。

错误类型 1 针对的是在 NIdentifier::codegen 函数中，使用 findVar 函数来检查变量名是否存在于变量名表中。如果变量名不存在，则报告错误。

错误类型 2 针对的是在 NMethodCall::codegen 函数中，检查 getFunction 返回的值。如果返回 nullptr，则表示调用的函数没有声明，因此报告错误。

错误类型 3 针对的是在 NExtDefFunDec::codegen 函数中，生成 VarDec 后，检查生成的 VarDec 的名字是否已经存在于变量表中。如果存在，则表示变量重复定义，需要报告错误。

Task4：语义分析与中间代码生成（二）

本题只需要补全错误类型 5、6、7、8 的检查即可（错误 1-3 的检查在 task3 中已完成，错误 4 的检查在样例代码中已给出）

在处理赋值符号两边表达式类型检测时，可以使用 getType 函数获取赋值运算符两边的类型，然后进行比较。如果类型不一致，则抛出错误。对于返回类型的检测，可以使用 getReturnType() 函数获得预期值，然后将其与实际值进行比较。至于参数检查，可以通过使用 arg_size() 函数来检查参数的数量，而类型检查则需要遍历函数的参数，借助 getArg() 和 getType() 函数来实现。

这种方法通过提供 API 函数来获取表达式和函数的类型信息，使得类型检查变得更加灵活和方便。

在比较类型时，可以根据需要调用相应的函数获取类型信息，从而更准确地进行类型检测。

Task5：语义分析与中间代码生成（三）

针对新增的样例，包括 if_else 和 while 语法结构，需要特别考虑变量的作用域控制问题。解决变量作用域的控制可以采用以下方案：在进入 CompStStmt 块时，首先备份原有的局部变量表，并将这个块内的所有变量视为全局变量。如果有新定义的变量与原有的全局变量同名，那么就覆盖原有的全局变量。在退出 CompStStmt 块时，将变量环境恢复到进入这个块之前的状态，即将原有的变量表恢复回来。

这种方法通过临时改变变量的作用域，将块内的变量视为全局变量，有效地解决了变量作用域的问题。当离开这个块时，将变量环境还原，以确保在不同的作用域中变量名的唯一性和正确性。

（3）实验总结及遇到的问题

本实验难度较大，实现了对 Mini-C 源文件的 LLVM IR 实现，完成了动态编译的目的，主要难点在任务 3 上（理解了任务 3 之后，4、5 就相对简单了），需要熟悉 LLVM IR 相关的 API，完善不同类型错误的检查工作，在理解对应语意规则之后，调用对应 API 即可。不足之处有：只完善了部分 codegen，即可通过测试，不过剩余的 codegen 完成也只是时间问题，没有技术难度。

5．MiniC 代码优化及目标代码生成

（1）实验内容

学习使用使用 LLVM 代码优化框架。

（2）实验过程

Task1：编译器中调用 LLVM 支持的优化函数

按照任务书要求以及提示逐步实现，调用 LLVM 中的优化函数，观察 IR 代码发生的变化。 由于编写的问题，代码无任何变化。

Task2：LLVM 命令行完成优化及目标代码生成

按照任务书要求以及提示逐步实现，文件 test.txt 中的 IR 进行优化，并最终能够产生可执行的二进制程序 test。

Task3：自定义优化函数

按照任务书，修改 countPass.cpp 文件，利用 F.getBasicBlockList().size()函数获得数据并按要求输出即可。

（3）实验总结及遇到的问题

此次实验任务较为简单，按照任务书修改执行即可，没有遇到什么问题。

二、实验心得

1. 能够基于科学原理和方法，根据需求选择路线，设计方案：

开始实验时，已经完成编译原理前四章的知识学习，对理论知识有了全面和系统的认识，并借此可做以下分析：

- 需求分析：明确编译器的需求和目标，包括支持的语言特性、目标平台、性能要求等。
- 词法分析：设计合适的词法规则，使用工具 Flex 来生成词法分析器。
- 语法分析：选择适合的上下文无关文法，使用工具 Bison 来生成语法分析器，并生成语法树。
- 语义分析：定义语义规则，对语法树进行语义分析，包括类型检查、作用域分析等。
- 语法制导翻译：根据语义规则，生成中间表示（如 LLVM IR）或目标代码。
- 优化：对生成的中间表示或目标代码进行各种优化，提高性能和效率。
- 测试和调试：设计合适的测试用例，对编译器进行测试和调试，确保其正确性和稳定性。

2. 选择、使用现代工具设计、预测、模拟与实现，分析局限：

根据编译原理的实验要求，可以使用现代工具来辅助设计、预测、模拟和实现编译器。一些常用的工具如下。

- Flex：用于生成词法分析器，通过正则表达式来匹配和提取词法单元。
- Bison：用于生成语法分析器，通过上下文无关文法来解析语法结构。
- LLVM：提供了丰富的 API 和工具，用于生成和优化中间表示（LLVM IR），以及生成目标代码。
- 编程语言：使用现代编程语言来实现编译器的各个阶段和组件。

通过本次实验，我对 Flex、Bison、LLVM 等工具有了深入浅出的认识和理解，特别是在处理语法规约冲突的时候，我对 Bison 的理解又上了一个档次，能基本熟练使用工具帮助我实现一个简易的编译器。

3. 开发满足特定需求的现代工具，分析其局限：

在编译原理实验中我们使用一种简化的语言 Mini-C，Mini-C 在教学和学习编译原理的过程中仍然是非常有价值的。它可以帮助学生理解和实践编译器的基本原理和技术，但在实际的编译器开发和应用中，可能需要考虑更完整、更强大的语言和工具，以下是它的一些局限性：

- 缺少高级优化和代码生成：Mini-C 并不提供复杂的优化和代码生成能力。在编译过程中，编译器通常会进行各种优化，如常量折叠、循环优化、内联等，以提高生成的目标代码的性能和效率。而 Mini-C 作为一个简化的语言，它的编译器可能无法实现这些高级优化技术。
- 缺少调试和错误处理支持：Mini-C 作为一个教学语言，通常没有提供完善的调试和错误处理机制。这可能会对开发和调试造成一定的困难，因为在生成的目标代码中可能缺少符号信息和调试信息，导致调试过程不够方便。

4. 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：

在编译原理实验中，我通过独立学习和实践克服了 Mini-C 语言作为目标语言的一些限制，并在这个过程中取得了显著的进展。通过深入阅读实验任务提供的 Flex、Bison、LLVM 的官方文档，我对编译原理实验的内容有了更深入的理解。

在实验过程中，我遇到了一些工程问题，如规约冲突、优先级顺序、面相对象的构造函数和虚构函数等挑战。但通过自学，问老师同学问题，在网上搜索相关博客，我成功解决了问题，并成功实现了原型。这种实践经验不仅加深了我对编译原理理论知识的理解，也培养了我解决问题的独立能力。

通过这个实验，我不仅掌握了编译原理的基本概念和技术，还培养了自学和解决问题的能力。我相信这将使我在未来的学习和工作中更具自信和竞争力。

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

本次实验将整个大项目分解成多个关卡、多个小任务，能让学生循序渐进逐步深入：在实验一中我学习到了对 Flex,Bison 的基本使用，能够对文本中的词法进行简单的分析以及一些简单的语法分析；在实验二中我学习到了更为复杂的词法分析，能够对八进制数，十六进制数以及小数等进行具体的识别；在实验三中我学习到了 Bison 冲突的解决方法以及语法树的构建；在实验四中我学习到了 LLVM IR 的编写以及如何依据语法树来动态生成 LLVM IR 文件；在实验五中我学习到了 LLVM 中的一些代码优化函数，以及编写了自己的一些函数。

前两个实验的体验还是很好的，但是实验三的 task4 可以说是难度剧增，不仅要求你熟练使用 bison（实验一二对 bison 的理解在这里是不够用的），而且要会面相对象的基础知识，再对照着庞大的工程文件一顿学习分析，网络上也缺乏相关的资源，作为新手的我们基本上是举步维艰。希望能将实验三 task4 难度降低一点，亦或是分解为更小的子任务，使知识的学习和过度更加平滑。至于实验四和实验五说实话，没太多必要，斯以为我们课程安排上存在一些不合理的地方：一是学业压力太大，特别是实验课太多了，这学期少说已经写了 10 个报告了，都是在不停的做实验，学生精力实在有限，独立完成做好每个实验属实有点强人所难；二是编译原理学时有限，理论课中语义分析即后半部分属实是囫囵吞枣，只懂一二，实验课这部分自然就显得有点画蛇添足。希望要么将编译原理分多点课时，学好学精，做好实验；要么则删去没必要的实验部分，让学生能有多的时间在感兴趣的领域探索！

最后，由衷地感谢各位老师在实验过程中的教导和辛勤付出！

参考文献

[1] 许畅 等编著. 《编译原理实践与指导教程》.机械工业出版社
[2] Jobn Levine 著，陆军 译. 《Flex 与 Bison》.东南大学出版社
[3] 刘铭, 骆婷, & 徐丽萍. (2018). 编译原理（第 4 版）. 电子工业出版社.