南开大学
Nankai University

南　　开　　大　　学

计 算 机 学 院

编译原理实验报告

期末实验报告

2011763 黄天昊

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2023 年 1 月 15 日

摘要

在本学期，我和队友一起完成了一个简化 C 语言——SysY 语言编译器的编程工作。在这一过程中，我们确定了编译器支持的 SysY 语言特性，并运用上下文无关文法（CFG）表述 SysY 语言子集。之后，我们逐步实现了词法分析器、语法分析器、中间代码生成以及最后的汇编代码生成和代码优化。

关键字：编译器；SysY 语言；

目录

一、 实现的 SysY 子集描述

我们实现的编译器确定 SysY 语言特性如下：

1. 数据类型的实现：int，float
2. 变量声明、常量声明，常量、变量的初始化
3. 语句：赋值（=）、表达式语句、语句块、if、while、for、return
4. 表达式：算术运算（+、-、*、/、%，其中 +、-都可以是单目运算符）、关系运算（==，>，<，>=，<=，!=）和逻辑运算（&&（与）、||（或）、！（非））
5. 注释（包含单行注释和多行注释，与 C 语言相同）
6. 输入输出（实现连接 SysY 运行时库）
7. 函数、语句块：函数声明、函数调用、变量、常量作用域
8. 数组：数组（一维、二维、…）的声明和数组元素访问
9. 浮点数：浮点数常量识别、变量声明、存储、运算

二、 词法分析

在一个编译器中，词法分析的作用是将输入的字符串流进行分析，并按一定的规则将输入的字符串流进行分割，从而形成所使用的源程序语言所允许的记号 (token) 获得 token 序列，在这些记号序列将送到随后的语法分析过程中，与此同时将不符合规范的记号识别出来，并产生错误提示信息。

源代码程序被输入到扫描器（Scanner），扫描器对源代码进行简单的词法分析，运用类似于有限状态机（Finite State Machine）的算法可以很轻松的将源代码字符序列分割成一系列的记号（Token）。

词法分析产生的记号一般可以分为如下几类：关键字、标识符、字面量（包含数字、字符串等）和特殊符号（如加号、等号）。在识别记号的同时，扫描器也完成了其他工作，比如将标识符存放到符号表，将数字、字符串常量存放到文字表等，以备后面的步骤使用。

所涉及到的模块文件：lexer.l。

(一) 词法结构的正则表达式定义

词法分析首先需要根据设计的编译器所支持的语言特性，设计正则表达式定义。经过讨论和测试，我们设计的正规表达式定义如下：

词法结构的正则表达式定义

```
DECIMIAL ([1-9][0-9]*|0)
FLOATDECIMAL ([1-9][0-9]*|0)("."[0-9]*)?(("E"|"e")("+"|"-")?[0-9]+)?
OCTAL (0[0-7]+)
HEXADECIMAL (0[xX][0-9A-Fa-f]+)
ID [[:alpha:]_][[:alpha:][:digit:]_]*
EOL (\r\n|\n|\r)
```

```
WHITE [\t ]
STRING \".*\"
BLOCKCOMMENTBEGIN \/\*
BLOCKCOMMENTELEMENT .
BLOCKCOMMENTEND \*\/
%x BLOCKCOMMENT
LINECOMMENT \/\/[^\n]*
```

(二) 具体代码实现

对于词法分析部分的代码，由于大部分较难的工作已经交给 Flex 来做，因此这里其实难度并不大，值得一提的 putint 和 getint 等 IO 函数。

以 putint() 函数为例：

词法分析 IO 函数接口

```
"putint" {
    char *lexeme;
    lexeme = new char[strlen(yytext) + 1];
    strcpy(lexeme, yytext);
    yylval.strtype = lexeme;
    if(identifiers->lookup(yytext)!=nullptr){
        return ID;
    }

    Type *funcType;
    std::vector<SymbolEntry*> paramsSymbolEntry;
    paramsSymbolEntry.push_back(new TemporarySymbolEntry(TypeSystem::intType,
        SymbolTable::getLabel()));
    funcType = new FunctionType(TypeSystem::voidType, paramsSymbolEntry);

    SymbolTable* cur = identifiers;
    while(cur->getPrev())
        cur = cur->getPrev();
    SymbolEntry* se = new IdentifierSymbolEntry(
        funcType, yytext, cur->getLevel());
    cur->install(yytext, se);
    return ID;
}
```

该部分有两个需要注意的点

1. 如果该函数有参数，需要定义一个 paramsSymbolEntry 对象，并将含有对应参数类型的 TemporarySymbolEntry 插入其中，便于语义分析时检查参数类型。
2. 需要创建一个 IdentifierSymbolEntry，IdentifierSymbolEntry 的名称为函数名，并插入 SymbolTable 中。且插入位置应当为 SymbolTable 最顶层，即 level 0。关于 SymbolTable 的 level 在语法分析部分进行阐述。

其他的 token 流识别代码由于大部分都较为重复，在这里只展示一部分：

词法结构的正则表达式定义

```
"break" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("BREAK\tbreak");
    charlen += strlen("break");
    return BREAK;
}
"continue" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("CONTINUE\tcontinue");
    charlen += strlen("continue");
    return CONTINUE;
}
"&&" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("AND\t&&");
    charlen += strlen("&&");
    return AND;
}
"||" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("OR\t||");
    charlen += strlen("||");
    return OR;
}
```

三、 语法分析

词法分析得到的，实质是语法树的叶子结点的属性值，语法树所有结点均由语法分析器创建。在自底向上构建语法树时（与预测分析法相对），我们使用孩子结点构造父结点。在 yacc 每次确定一个产生式发生归约时，我们会创建出父结点、根据子结点正确设置父结点的属性、记录继承关系。

语法分析的目的是构建出一棵抽象语法树（AST），因此我们需要设计语法树的结点。结点分为许多类，除了一些共用属性外，不同类结点有着各自的属性、各自的子树结构、各自的函数实现。结点的型大体上可以分为表达式和语句，每种类型又可以分为许多子类型，如表达式结点可以分为词法分析得到的叶结点、二元运算表达式的结点等；语句还可以分为 if 语句、while 语句和块语句等。

在这一过程中，我们还需要维护好类型系统和符号表。

对于我们的编译器，类型系统较为简单，但是为了实现数组和 float，我们仍需仔细设计每一个具体的类型类。如函数类型，需要保存的不仅是一个标识函数类型的 bool 值，还需要保存对应函数的返回类型，以及函数中每个参数的类型。

符号表是编译器用于保存源程序符号信息的数据结构，这些信息在词法分析、语法分析阶段被存入符号表中，最终用于生成中间代码和目标代码。符号表条目可以包含标识符的词素、类型、作用域、行号等信息。

符号表主要用于作用域的管理，我们为每个语句块创建一个符号表，块中声明的每一个变量

都在该符号表中对应着一个符号表条目。在词法分析阶段，我们只能识别出标识符，不能区分这个标识符是用于声明还是使用。而在语法分析阶段，我们能清楚的知道一个程序的语法结构，如果该标识符用于声明，那么语法分析器将创建相应的符号表条目，并将该条目存入当前作用域对应的符号表中，如果是使用该标识符，将从当前作用域对应的符号表开始沿着符号表链搜索符号表项。

所涉及到的模块文件：parser.y, Type.h, SymbolTable.h, Ast.h, Type.cpp, SymbolTable.cpp, Ast.cpp。

(一) 语法结构的上下文无关文法定义

确定了要实现的 SysY 语言特性之后，将其抽象化、符号化，参考巴克斯瑙尔范式定义，设计相应的上下文无关文法，具体如下：

1. 声明

这一部分主要包括函数、常量、变量的声明

抽象语义	表达式
编译单元	CompUnit -> CompUnit Decl | CompUnit FuncDef | Decl |FuncDef
声明	Decl -> ConstDecl | VarDecl
基本类型	BType -> **'int'** | **'float'**
常量声明	ConstDecl -> **'const'** BType ConstDef ConstDefList **';'**
常数定义中间符号	ConstDefList -> ConstDefList **','** ConstDef | ϵ
常数定义	ConstDef -> **Ident** ConstExpList '=' ConstInitVal
常量表达式中间符号	ConstExpList -> ConstExpList '[' ConstExp ']' | ϵ
常量初值	ConstInitVal -> ConstExp | '{' ConstInitVal ConstInitValList '}' | '{' '}'
常量初值中间符号	ConstInitValList -> ConstInitValList **','** ConstInitVal | ϵ
变量声明	VarDecl -> BType VarDef VarDefList **';'**
变量声明中间符号	VarDefList -> VarDefList **','** VarDef | ϵ
变量定义	VarDef -> **Ident** ConstExpList | **Ident** ConstExpList '=' InitVal
变量初值	InitVal -> Exp | '{' InitVal InitValList '}' | '{' '}'
变量初值中间符号	InitValList -> InitValList **','** InitVal | ϵ

表 1: 声明（函数、变量、常量）部分的 CFG 设计

2. 函数

抽象语义	表达式
函数定义	FuncDef -> FuncType **Ident** '(' FuncFParams ')' Block | FuncType **Ident** '(' ')' Block
函数类型	FuncType -> **'void'** | **'int'** | **'float'**
函数形参表	FuncFParams -> FuncFParam FuncFParamList
函数形参中间符号	FuncFParamList -> FuncFParamList **','** FuncFParam | ϵ
函数形参	FuncFParam -> BType **Ident** '[' ']' ExpList | BType **Ident**
函数中间符号	ExpList -> ExpList '[' Exp ']' | ϵ

表 2: 函数部分的 CFG 设计

3. 语句

抽象语义	表达式
语句块	Block -> '{' BlockItemList '}'
语句块项中间符号	BlockItemList -> BlockItemList BlockItem | ϵ
语句块项	BlockItem -> Decl | Stmt
语句	Stmt -> LVal '=' Exp ';' | Exp ';' | ';' | Block | **'if'** '(' Cond ')' Stmt **'else'** Stmt | **'if'** '(' Cond ')' Stmt | **'while'** '(' Cond ')' Stmt | **'break'** ';' | **'continue'** ';' | **'return'** Exp ';' | **'return'** ';'

表 3: 语句部分的 CFG 设计

4. 表达式

部分表达式采用了扩展的 Backus 范式表达，以简化产生式表达。其中，符号 (...) 表示可以匹配圆括号内的任一字符。

抽象语义	表达式
表达式	Exp -> AddExp
条件表达式	Cond -> LOeExp
左值表达式	LVal -> identifier Explist
基本表达式	PrimaryExp -> '(' Exp ')' | LVal | Number
数值	Number -> integer-const | floating-const
一元表达式	UnaryExp -> PrimaryExp | identifier '(' ')' | identifier '(' FuncRParams ')'
单目运算符	UnaryOp -> '+' | '−' | '!'
函数实参表	FuncRParams -> Func-Explist
函数实参列表	Func-Explist -> Func-Explist ',' Exp | Exp | ϵ
乘除模表达式	UnaryExp | MulExp '*' UnaryExp
乘除模表达式	| MulExp '/' UnaryExp
乘除模表达式	| MulExp '%' UnaryExp
加减表达式	MulExp | AddExp '+' MulExp
加减表达式	MulExp | AddExp '-' MulExp
关系表达式	RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
相等性表达式	EqExp → RelExp | EqExp ('==' | '!=') RelExp
逻辑与表达式	LAndExp → EqExp | LAndExp '&&' EqExp
逻辑或表达式	LOrExp → LAndExp | LOrExp '||' LAndExp
常量表达式	ConstExp → AddExp

表 4: 表达式部分的 CFG 设计

5. 注释

注释部分的 CFG 设计，包含了以‘//’符号开始的单行注释（不包含换行符）和以‘/*’符号开始、‘*/’符号结束的多行注释（包含换行符）。

抽象语义	表达式
注释	Comments -> '//' strings-nonLBreak | '/*' strings '*/' | ϵ
无换行符字符串	strings-nonLBreak -> char-nonLBreak-list
无换行符字符列表	char-nonLBreak-list -> char-nonLBreak-list char-nonLBreak | char-nonLBreak | ϵ
含换行符字符串	strings -> char-list
含换行符字符列表	char-list -> char-list | char | ϵ

表 5: 注释部分的 CFG 设计

6. 常量

常量包含了 int 型和 float 型常量的表达。

抽象语义	表达式
整型常量	integer-const -> decimal-const
十进制整型常量	decimal-const -> nonzero-digit | decimal-const digit
浮点型常量	floating-const -> decimal-floating-const
十进制浮点型常量	decimal-floating-const -> fractional-const exponent-part | digit-sequence exponent-part
浮点部分	fractional-const -> digit-sequence '.' digit-sequence | '.' digit-sequence | digit-sequence '.'
指数部分	exponent-part -> 'e' Sign digit-sequence | 'E' Sign digit-sequence | 'e' digit-sequence | 'E' digit-sequence | ϵ
符号	Sign -> '+' | '-'
数字序列	digit-sequence -> digit-sequence digit

表 6: 常量部分的 CFG 设计

其中，我负责设计了数据类型的实现，浮点数，变量、常量声明与初始化，语句（即上文中 SysY 语言特性 1-3 与 7-9）

(二) 具体代码实现

由于篇幅限制，在此只展示部分代码。

语法分析与语法树的创建

```
Program
    : Stmts {
        ast.setRoot($1);
    }
    ;
Stmts
    : Stmt { $$ = $1; }
    | Stmts Stmt {
        $$ = new SeqNode($1, $2);
    }
    ;
Stmt
    : AssignStmt { $$ = $1; }
    | ExprStmt { $$ = $1; }
    | BlockStmt { $$ = $1; }
    | IfStmt { $$ = $1; }
    | BreakStmt { $$ = $1; }
    | ContinueStmt { $$ = $1; }
    | WhileStmt { $$ = $1; }
    | ReturnStmt { $$ = $1; }
    | DeclStmt { $$ = $1; }
    | FuncDef { $$ = $1; }
    | BlankStmt { $$ = $1; }
    ;
```

```
LVal
    : ID {
        SymbolEntry* se;
        se = identifiers->lookup($1);
        if (se == nullptr)
            fprintf(stderr, "ID %s 未定义!\n", (char*)$1);
        $$ = new Id(se);
        delete []$1;
    }
    | ID ArrayIndices
    {
        // modified恢复数组的定义
        SymbolEntry* se;
        // 保存数组名
        se = identifiers->lookup($1);
        // 这里是用array来初始化一个ID
        $$ = new Id(se, $2);
        delete []$1;
    }
    ;
ExprStmt
    : Exp SEMICOLON {
        $$ = new ExprStmt($1);
    }
    ;
AssignStmt  // eg: a = 1 + 1; 赋值语句
    : LVal ASSIGN Exp SEMICOLON {
        $$ = new AssignStmt($1, $3);
    }
    ;
BlankStmt  // 针对;号这种空语句
    : SEMICOLON {
        $$ = new BlankStmt();
    }
    ;
BlockStmt
    : LBRACE {
        identifiers = new SymbolTable(identifiers);  // 新作用域
    }
    Stmts RBRACE {
        $$ = new CompoundStmt($3);
        SymbolTable* top = identifiers;
        identifiers = identifiers->getPrev();  // 块结束之后调整回上一级作用
            域
        delete top;
    }
    | LBRACE RBRACE {
        // TODO
```

```
72          //$$->setHaveRetStmt(false);
73          $$ = new BlankStmt();
74      }
75      ;
76  IfStmt
77      : IF LPAREN Cond RPAREN Stmt %prec THEN {
78          $$ = new IfStmt($3, $5);
79      }
80      | IF LPAREN Cond RPAREN Stmt ELSE Stmt {
81          $$ = new IfElseStmt($3, $5, $7);
82      }
83      ;
84  WhileStmt
85      : WHILE LPAREN Cond RPAREN {
86          StmtNode *whileNode = new WhileStmt($3);
87          whileStk.push(whileNode);
88      }
89      Stmt {
90          StmtNode *whileNode = whileStk.top();
91          ((WhileStmt*)whileNode)->setStmt($6);  // 设置内部stmt语句
92          $$ = whileNode;
93          whileStk.pop();
94      }
95      ;
96  BreakStmt  // break和continue要匹配while
97      : BREAK SEMICOLON {
98          $$ = new BreakStmt(whileStk.top());
99      }
100     ;
101 ContinueStmt
102     : CONTINUE SEMICOLON {
103         $$ = new ContinueStmt(whileStk.top());
104     }
105     ;
106 ReturnStmt
107     // TODO
108     : RETURN SEMICOLON
109     {
110         ReturnStmt* ret = new ReturnStmt();
111         // ret->typeCheck(curFunc);
112         // ret->setHaveRetStmt(true);
113         $$ = ret;
114     }
115     | RETURN Exp SEMICOLON
116     {
117         ReturnStmt* ret = new ReturnStmt($2);
118         // ret->typeCheck(curFunc);
119         // ret->setHaveRetStmt(true);
```

```
120         $$ = ret;
121     }
122     ;
123 Exp
124     :
125     AddExp { $$ = $1; }
126     ;
127 Cond
128     :
129     LOrExp { $$ = $1; }
130     ;
131 PrimaryExp  // 表达式最初的右值, 一般为数字0-9, 或者id
132     : LPAREN Exp RPAREN {
133         $$ = $2;
134     }
135     | LVal {
136         $$ = $1;
137     }
138     | INTEGER {
139         SymbolEntry* se = new ConstantSymbolEntry(TypeSystem::intType, $1);
140         $$ = new Constant(se);
141     }
142     | FLOATNUM {
143         SymbolEntry *se = new ConstantSymbolEntry(TypeSystem::floatType, $1);
144         $$ = new Constant(se);
145     }
146     | ID LPAREN RPAREN {
147         SymbolEntry* se;
148         se = identifiers->lookup($1);
149         $$ = new CallExpr(se);
150     }
151     | ID LPAREN FuncRParams RPAREN {
152         SymbolEntry* se;
153         se = identifiers->lookup($1);
154         // TODO
155         if (se == nullptr) {
156             fprintf(stderr, "函数 %s 未定义\n", (char*)$1);
157         }
158         else {
159             $$ = new CallExpr(se, $3);
160         }
161     }
162     ;
163 UnaryExp
164     : PrimaryExp { $$ = $1; }
165     | ADD UnaryExp { $$ = $2; }
166     | SUB UnaryExp
167     {
```

```
168         SymbolEntry* se = new TemporarySymbolEntry(TypeSystem::intType,
                SymbolTable::getLabel());
169         $$ = new UnaryExpr(se, UnaryExpr::SUB, $2);
170     }
171     | NOT UnaryExp
172     {
173         // 注意NOT是Bool类型
174         SymbolEntry* se = new TemporarySymbolEntry(TypeSystem::boolType,
                SymbolTable::getLabel());
175         $$ = new UnaryExpr(se, UnaryExpr::NOT, $2);
176     }
177     ;
```

通过这部分代码，我们就可以将词法分析器给出的 token 流，自底向上地逐步构建出语法树。

类型系统部分在这里只展示数组与函数类：

类型系统

```
1  class FunctionType : public Type
2  {
3      private:
4       Type* returnType;
5       std::vector<Type*> paramsType;
6       std::vector<SymbolEntry*> paramsSe;   // 存储parameters的type和id
7
8      public:
9       FunctionType(Type* returnType, std::vector<Type*> paramsType, std::vector
            <SymbolEntry*> paramsSe)
10          : Type(Type::FUNC), returnType(returnType), paramsType(paramsType),
                paramsSe(paramsSe){};
11      void setParamsType(std::vector<Type*> paramsType) {
12          this->paramsType = paramsType;
13      };
14      std::vector<Type*> getParamsType() { return paramsType; };
15      std::vector<SymbolEntry*> getParamsSe() { return paramsSe; };
16      Type* getRetType() { return returnType; };
17      std::string toStr();
18 };
19
20 class ArrayType : public Type
21 {
22 private:
23      Type* elementType;
24      Type* arrayType = nullptr;
25      int length;
26      bool constant;
27 public:
28      ArrayType(Type* elementType, int length, bool constant = false) : Type(
```

```
        Type::ARRAY), elementType(elementType), length(length), constant(
        constant) {
29          size = elementType->getSize() *length;
30      };
31      std::string toStr();
32      int getLength() const { return length; };
33      Type* getElementType() const { return elementType; };
34      void setArrayType(Type* arrayType) { this->arrayType = arrayType; };
35      bool isConst() const { return constant; };
36      Type* getArrayType() const { return arrayType; };
37  };
38
39  std::string ArrayType::toStr() {
40      std::vector<std::string> vec;
41      Type* temp = this;
42      int count = 0;
43      bool flag = false;
44
45      while (temp && temp->isArray()) {
46          std::ostringstream buffer;
47          if (((ArrayType*)temp)->getLength() == -1) {
48              flag = true;
49          }
50          else {
51              buffer << "[" << ((ArrayType*)temp)->getLength() << " x ";
52              count++;
53              vec.push_back(buffer.str());
54          }
55          temp = ((ArrayType*)temp)->getElementType();
56      }
57
58      assert(temp->isInt());
59      std::ostringstream buffer;
60      for (auto iter = vec.begin(); iter != vec.end(); iter++){
61          buffer << *iter;
62      }
63      buffer << "i32";
64      while (count--){
65          buffer << ']';
66      }
67      if (flag){
68          buffer << '*';
69      }
70      return buffer.str();
71  }
72
73  std::string FunctionType::toStr() {
74      std::ostringstream buffer;
```

```
    buffer << returnType->toStr() << "(";
    for (auto it = paramsType.begin(); it != paramsType.end(); it++)
    {
        buffer << (*it)->toStr();
        if (it + 1 != paramsType.end()){
            buffer << ", ";
        }
    }
    buffer << ')';
    return buffer.str();
}
```

同样的，符号表的实现部分代码过长，这里只展示基类和 id 类的声明。

符号表

```
class SymbolEntry
{
private:

protected:
    enum { CONSTANT, VARIABLE, TEMPORARY };
    Type* type;
    int kind;
    SymbolEntry* next;  // 用来解决函数重载问题

public:
    // TODO next initialize
    // // 这里有个小知识点，之前一直报 warning:  when initialized here [-
        Wreorder]，查了一下，原来类在初始化的时候需要按照成员声明的顺序来
    SymbolEntry(Type* type, int kind);
    virtual ~SymbolEntry(){};
    bool isConstant() const { return kind == CONSTANT; };
    bool isTemporary() const { return kind == TEMPORARY; };
    bool isVariable() const { return kind == VARIABLE; };
    Type* getType() { return type; };
    void setType(Type* type) { this->type = type; };
    virtual std::string toStr() = 0;
    bool setNext(SymbolEntry* se);
    SymbolEntry* getNext() const { return next; };
};

// symbol table managing identifier symbol entries
class SymbolTable
{
private:
    std::map<std::string, SymbolEntry*> symbolTable;
    SymbolTable* prev;
    int level;
```

```
33      static int counter;
34  public:
35      SymbolTable();
36      SymbolTable(SymbolTable* prev);
37      bool install(std::string name, SymbolEntry* entry);
38      SymbolEntry* lookup(std::string name);
39      SymbolEntry* searchFunc();
40      SymbolEntry* checkRepeat(std::string name);
41      SymbolTable* getPrev() { return prev; };
42      int getLevel() { return level; };
43      static int getLabel() { return counter++; };
44      static void resetLabel() { counter = 0; };
45  };
46  extern SymbolTable *identifiers;
47  extern SymbolTable *globals;
48
49  class IdentifierSymbolEntry : public SymbolEntry {
50  private:
51      enum { GLOBAL, PARAM, LOCAL };
52      std::string name;
53      int scope;
54      int value;
55      float fvalue;
56      int label;
57      bool initial;
58      bool sysy;  // 是否为外部函数
59      int* arrayValue;
60      bool allZero;
61      int paramNo;
62      bool constant;  // 判断是否为常量的标志位
63      Operand* addr;  // The address of the identifier.
64  public:
65      IdentifierSymbolEntry(Type* type, std::string name, int scope, int
            paramNo = -1, bool sysy = false);
66      virtual ~IdentifierSymbolEntry() {};
67      std::string toStr();
68      int getScope() const { return scope; };
69      void setAddr(Operand* addr) { this->addr = addr; };
70      Operand* getAddr() { return addr; };
71      void setValue(int value);
72      int getValue() const { return value; };
73      // void setfValue(float fvalue);
74      // float getfValue() const{ return fvalue; };
75      void setArrayValue(int* arrayValue);
76      int* getArrayValue() const { return arrayValue; };
77      int getLabel() const { return label; };
78      void setLabel() { label = SymbolTable::getLabel(); };
79      void setAllZero() { allZero = true; };
```

```
    bool isAllZero() const { return allZero; };
    int getParamNo() const { return paramNo; };
    void setConst() { constant = true;};
    bool getConst() const { return constant; };
    bool isGlobal() const { return scope == GLOBAL; };
    bool isParam() const { return scope == PARAM; };
    bool isLocal() const { return scope >= LOCAL; };
    bool isSysy() const { return sysy; };
};
```

对于所有的作用域，每个作用域都新声明一个符号表类对象，用链表的形式进行连接。在进行语法分析时，通过两个 extern 变量进行作用域的切换。

```
extern SymbolTable *identifiers;
extern SymbolTable *globals;
```

setnext 函数

```
bool SymbolEntry::setNext(SymbolEntry* se) {
    SymbolEntry* s = this;
    SymbolEntry* sss = nullptr;
    int newCount = ((FunctionType*)(se->getType()))->getParamsSe().size();
        // 新声明的函数形参个数

    while (s) {
        if (newCount == int(((FunctionType*)(s->getType()))->getParamsSe().
            size())) {  // 如果当前这个已声明的函数形参个数与新声明的相等
            bool flag = 0;
            for (int i = 0; i < newCount; i++) {  // 遍历每个形参，只要有某个
                位置两个形参类型不同即可算作函数重载
                if (((FunctionType*)(s->getType()))->getParamsSe()[i]->
                    getType()->toStr() != ((FunctionType*)(se->getType()))->
                    getParamsSe()[i]->getType()->toStr()) {
                    flag = 1;
                }
            }
            if (!flag) {  // flag=0, 表示有已声明的函数形参表与新声明的一模一
                样，那么算作函数重定义
                fprintf(stderr, "函数 %s 重定义!\n", se->toStr().c_str());
                return false;
            }
        }
        sss = s;
        s = s->getNext();
    }

    fprintf(stderr, "函数 %s 重载\n", se->toStr().c_str());
    if (sss == this) {  // 这里为什么非要多此一举用ifelse？直接sss->setNext(
        se)不行吗？不行！会出现递归调用
```

```
        this->next = se;
    } else {
        sss->setNext(se);
    }
    return true;
}
```

对于符号系统，由于涉及到作用域的切换等操作，经常需要在所有符号表中去寻找一个特定的 identifier，这就用到了 lookup 函数

lookup 函数

```
SymbolEntry* SymbolTable::lookup(std::string name) {
    // 很常用，从当前的代码块对应的符号表开始查起，一直查到global的符号表
    SymbolTable *temp = identifiers;
    SymbolEntry *result = nullptr;  // 最后的搜索结果
    while (temp != nullptr && result == nullptr) {
        for (auto it : temp->symbolTable) {
            if (it.first == name) {
                // fprintf(stderr, "lookup %s\n", name.c_str());
                result = it.second;
                break;
            }
        }
        temp = temp->getPrev();
    }
    return result;
}
```

四、 类型检查

类型检查部分基本由我完成。

类型检查是编译过程的重要一步，以确保操作对象与操作符相匹配。每一个表达式都有相关联的类型，如关系运算表达式的类型为布尔型，而计算表达式的类型一般为整型或浮点型等。类型检查的目的在于找出源代码中不符合类型表达式规定的代码，在最终的代码生成之前报错，使得程序员可以根据错误信息对源代码进行修正。

由于我们基于 SysY 语言子集进行语法设计，Identifier 的基本类型仅包括 int，float，数组。这减轻了类型检查的工作任务。我们仅需要针对以下几种情况进行处理:

- 变量未声明，及在同一作用域下重复声明
- 条件判断表达式 int 至 bool 类型隐式转换
- 数值运算表达式运算数类型是否正确 (可能导致此错误的情况有返回值为 void 的函数调用结果参与了某表达式计算)
- 函数未声明，及不符合重载要求的函数重复声明
- 函数调用时形参及实参类型或数目的不一致

- return 语句操作数和函数声明的返回值类型是否匹配
- 对数组维度进行相应的类型检查
- 对 break、continue 语句进行静态检查，判断是否仅出现在 while 语句中

类型检查可以在建立语法树的过程中进行相应的识别和处理，也可以在建树完成后，自底向上遍历语法树进行类型检查。类型检查过程中，父结点需要检查孩子结点的类型，并根据孩子结点类型确定自身类型。有一些表达式可以在语法制导翻译时就确定类型，比如整数就是整型，这些表达式通常是语法树的叶结点。而有些表达式则依赖其子表达式确定类型，这些表达式则是语法树中的内部结点。

我在进行类型检查时，将一部分类型检查放到了建树过程中，另一部分在建树完成之后对语法树进行遍历检查。

所涉及到的模块文件：Type.h, SymbolTable.h, Ast.h, Type.cpp, SymbolTable.cpp, Ast.cpp。

(一) 具体代码实现

1. 变量未声明、或在同一作用域下重复声明

对于变量，我们实现的 SysY 语言子集只包括 int 和 float 两种类型，因此工作相对简单。最开始我想将其放到遍历整棵语法树的阶段去检查，但是这样做存在一些问题。如果是等到语法树生成完毕再自底向上检查，会由于 symboltable 都被删掉而出现问题。因此还是放到语法分析阶段生成 ID 节点的时候进行检查。

语法分析阶段对于变量的处理

```
VarDef  // 定义语句
    : ID {
        // eg: int a;
        SymbolEntry* se;
        se = new IdentifierSymbolEntry(TypeSystem::intType, $1, identifiers->
            getLevel());
        identifiers->install($1, se);
        $$ = new DeclStmt(new Id(se));
        delete []$1;
    }
    | ID ASSIGN InitVal {
        if(declType->isFloat()) {
            declType = TypeSystem::constFloatType;
        }
        if(declType->isInt()) {
            declType = TypeSystem::constIntType;
        }

        // 可以简写
        SymbolEntry* se = new IdentifierSymbolEntry(TypeSystem::intType, $1,
            identifiers->getLevel());
        identifiers->install($1, se);

        // 进行类型的判断选择赋值，符号表之中并没有存储任何的数值，包括const
            和var都得补充
```

```
23          /* 这里需要注意的是需要补充对expr node的属性的值的计算
24          if(declType->isFloat()){
25              ((IdentifierSymbolEntry*)se)->setfValue($3->getvalue());
26          }
27          if(declType->isInt()){
28              ((IdentifierSymbolEntry*)se)->setiValue($3->getvalue());
29          }
30          */
31
32          ((IdentifierSymbolEntry*)se)->setValue($3->getValue());
33          $$ = new DeclStmt(new Id(se), $3);
34          delete []$1;
35      }
36      | ID ArrayIndices {
37          // eg: int a[10];
38          std::vector<int> vec;  // 分别存放维度值, 可能是很多维
39          SymbolEntry* se;
40          ExprNode* temp = $2;    // array
41          // 编译每个维度数组, 从高维到低维, 这里用next全部遍历
42          while(temp) {
43              vec.push_back(temp->getValue());
44              temp = (ExprNode*)(temp->getNext());
45          }
46
47          // TODO
48          Type* type = TypeSystem::intType;    // 目前仅有int类型
49          Type* temp1;
50
51          // 注意这里是倒序遍历, 即从右向左, a[2][3]: 2个数组指针 每行3个整型元
                素
52          while(!vec.empty()) {
53              // 数组一层一层的类型存储
54              temp1 = new ArrayType(type, vec.back());
55              // 考虑多维数组, 每个元素是数组指针
56              // 如果元素是数组, type设置为数组维度
57              if(type->isArray()) {
58                  ((ArrayType*)type)->setArrayType(temp1);
59              }
60
61              type = temp1;
62              vec.pop_back();
63          }
64
65          // type保存最低维, 即a[2][4]即保存2对应类型
66          arrayType = (ArrayType*)type;
67          se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
68
69          // 初始化为0
```

```
        ((IdentifierSymbolEntry*)se)->setAllZero();

        // 设置整型空间, 即长度*大小
        int *p = new int[type->getSize()];
        ((IdentifierSymbolEntry*)se)->setArrayValue(p);
        identifiers->install($1, se);
        $$ = new DeclStmt(new Id(se));
        delete []$1;
    }
    | ID ArrayIndices ASSIGN InitVal
    {
        SymbolEntry* se = nullptr;
        std::vector<int> vec;
        ExprNode* temp = $2;

        while (temp) {
            vec.push_back(temp->getValue());
            temp = (ExprNode*)(temp->getNext());
        }

        // 获取数组的各个维度??
        Type* type = TypeSystem::intType;
        Type* temp1 = new ArrayType(type, vec.back());

        $<se>$ = se;

        ((ArrayType*)type)->setArrayType(temp1);
        // modified
        se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
        arrayValue = new int[arrayType->getSize()];
        notZeroNum = 0;
    }
    ;
```

install 函数

```
bool SymbolTable::install(std::string name, SymbolEntry* entry) {
    // fprintf(stderr, "install %s\n", name.c_str());
    SymbolEntry* se = nullptr;
    if (entry->getType()->isFunc()) {  // 如果是函数, 需要到上一级符号表去找
        se = identifiers->getPrev()->checkRepeat(name);
    }
    else {
        se = identifiers->checkRepeat(name);
    }

    if (se) {  // 判断是否在同一作用域下重复定义
        if (se->getType()->isFunc()) {
            return se->setNext(entry);
```

```
        }
        else {
            fprintf(stderr,"Id %s 重定义!\n", name.c_str());
            return false;
        }
    }

    symbolTable[name] = entry;
    return true;
}
```

install 函数原本只负责将每个 identifier 插入符号表，在类型检查阶段，其又多了一个工作——检查是否出现变量在同一作用域下重复定义的问题。

2. 函数未声明、及不符合重载要求的重复声明

函数的未声明或重复声明问题与变量类似，但是由于函数存在重载，因此会更加复杂。总体上还是与变量的类型检查一致，在建树过程中就对这一问题进行检查。

语法分析器对于函数 id 的处理

```
FuncDef
    : Type ID {
        identifiers = new SymbolTable(identifiers);  // 申请新的符号表，此时为该函数的符号表
        paramNum = 0;  // 标记参数的id
    }
    LPAREN OptFuncFParams RPAREN {
        Type* funcType;
        std::vector<Type*> vec;
        std::vector<SymbolEntry*> vec1;

        // 将所有形参全部装入params
        DeclStmt* temp = (DeclStmt*)$5;
        while(temp) {
            vec.push_back(temp->getId()->getSymbolEntry()->getType());
            vec1.push_back(temp->getId()->getSymbolEntry());
            temp = (DeclStmt*)(temp->getNext());
        }

        // 输入参数类型和符号表项
        funcType = new FunctionType($1, vec, vec1);
        SymbolEntry* se = new IdentifierSymbolEntry(funcType, $2, identifiers->getPrev()->getLevel());
        identifiers->getPrev()->install($2, se);     // 将函数ID和函数类型装入符号表，这个是该函数上一级的符号表

        $<se>$ = se;  // 下面使用
    }
    BlockStmt {
```

```
        $$ = new FunctionDef($<se>7, (DeclStmt*)$5, $8);
        SymbolTable* top = identifiers;
        identifiers = identifiers->getPrev();  // 回到上一级符号表
        delete top;
        delete []$2;
    }
    ;
```

可以看到，语法分析阶段对于函数 id 的处理依旧是通过 install 函数来插入到符号表中。

install 函数

```
bool SymbolTable::install(std::string name, SymbolEntry* entry) {
    // fprintf(stderr, "install %s\n", name.c_str());
    SymbolEntry* se = nullptr;
    if (entry->getType()->isFunc()) {  // 如果是函数，需要到上一级符号表去找
        se = identifiers->getPrev()->checkRepeat(name);
    }
    else {
        se = identifiers->checkRepeat(name);
    }

    if (se) {  // 判断是否在同一作用域下重复定义
        if (se->getType()->isFunc()) {
            return se->setNext(entry);
        }
        else {
            fprintf(stderr,"Id %s 重定义!\n", name.c_str());
            return false;
        }
    }

    symbolTable[name] = entry;
    return true;
}
```

在 install 函数中，如果当前的 symbolEntry 是一个函数类型的 id，处理起来比变量会复杂一些。这里会使用 setNext 函数进行进一步检查。

setNext 函数

```
bool SymbolEntry::setNext(SymbolEntry* se) {
    SymbolEntry* s = this;
    SymbolEntry* sss = nullptr;
    int newCount = ((FunctionType*)(se->getType()))->getParamsSe().size();
        // 新声明的函数形参个数

    while (s) {
        if (newCount == int(((FunctionType*)(s->getType()))->getParamsSe().
            size())) {  // 如果当前这个已声明的函数形参个数与新声明的相等
            bool flag = 0;
```

```
            for (int i = 0; i < newCount; i++) {  // 遍历每个形参，只要有某个位置两个形参类型不同即可算作函数重载
                if (((FunctionType*)(s->getType()))->getParamsSe()[i]->getType()->toStr() != ((FunctionType*)(se->getType()))->getParamsSe()[i]->getType()->toStr()) {
                    flag = 1;
                }
            }
            if (!flag) {  // flag=0, 表示有已声明的函数形参表与新声明的一模一样，那么算作函数重定义
                fprintf(stderr, "函数 %s 重定义!\n", se->toStr().c_str());
                return false;
            }
        }
        sss = s;
        s = s->getNext();
    }

    fprintf(stderr, "函数 %s 重载\n", se->toStr().c_str());
    if (sss == this) {  // 这里为什么非要多此一举用ifelse？直接sss->setNext(se)不行吗？不行！会出现递归调用
        this->next = se;
    } else {
        sss->setNext(se);
    }
    return true;
}
```

在这里，setNext 函数是专门用来分辨函数的重载和重定义问题。比如，如果符号表中已经有了名称为 f 的函数，那么如果语法分析再遇到一个名称为 f 的函数，根据 install 函数的逻辑，其将会调用 setNext 函数进行进一步检查。

对于 setNext 函数，我参考了时浩铭学长的代码，但是发现他的代码虽然做了函数重载和重定义的区分，但是并不符合真正的 C 语言语法规范。如果存在 $f(inta, intb)$ 和 $f(inta, floatb)$ 两个函数的声明，那么他的代码会将后者识别为重定义，但其实函数参数数量相同但类型不同也属于重载。我的代码在其基础上进行了改进。既检查了重名函数的参数数量是否相同，如果在参数数量相同的情况下，会进一步检查每个参数的类型是否相同，只要有一组参数类型不同，则说明两个函数声明属于重载，而不是重复声明。

3. 数值运算表达式运算数类型是否正确

这部分检查很简单，只需要检查一下运算数是否为 void（函数返回值可能会出现 void），代码如下：

```
bool BinaryExpr::typeCheck(Type* retType) {
    this->expr1->typeCheck();
    this->expr2->typeCheck();
    Type *type1 = this->expr1->getSymbolEntry()->getType();
    Type *type2 = this->expr2->getSymbolEntry()->getType();
    if (type1->toStr() == "void") {  // 判断是否有函数返回void但是参与了运算
```

```
        fprintf(stderr, "expr1为 void 类型, 不能参与计算\n");
    }
    else if (type2->toStr() == "void") {
        fprintf(stderr, "expr2为 void 类型, 不能参与计算\n");
    }

    return false;
}
```

4. 条件判断表达式的隐式类型转换

这里其实对于条件判断表达式只是检查了一下是否是 int 类型，然后新建了一个隐式类型转换类，对其进行表示。

```
IfElseStmt::IfElseStmt(ExprNode* cond, StmtNode* thenStmt, StmtNode* elseStmt
    ) : cond(cond), thenStmt(thenStmt), elseStmt(elseStmt) {
    if (cond->getType()->isInt() && cond->getType()->getSize() == 32) {
        ImplictCastExpr* temp = new ImplictCastExpr(cond);
        this->cond = temp;
    }
}

WhileStmt::WhileStmt(ExprNode* cond, StmtNode* stmt) : cond(cond), stmt(stmt)
     {
    if (cond->getType()->isInt() && cond->getType()->getSize() == 32) {
        ImplictCastExpr* temp = new ImplictCastExpr(cond);
        this->cond = temp;
    }
}
```

```
ImplictCastExpr::ImplictCastExpr(ExprNode* expr) : ExprNode(nullptr,
    IMPLICTCASTEXPR), expr(expr) {
    type = TypeSystem::boolType;
    dst = new Operand(new TemporarySymbolEntry(type, SymbolTable::getLabel())
        );
}
```

隐式类型转换类

```
// int2bool, int2float, float2int
class ImplictCastExpr : public ExprNode
{
private:
    ExprNode* expr;
public:
    ImplictCastExpr(ExprNode* expr);
    bool typeCheck(Type* retType = nullptr) { return false; };
    void genCode();
    void output(int level);
```

```
    ExprNode* getExpr() const { return expr; };
};
```

5. return 语句操作数和函数声明的返回值类型是否匹配

return 语句的类型检查放到了遍历语法树时。

语法分析阶段对于 return 和 funcdefinition 的处理

```
ReturnStmt
    // TODO
    : RETURN SEMICOLON
    {
        ReturnStmt* ret = new ReturnStmt();
        // ret->typeCheck(curFunc);
        // ret->setHaveRetStmt(true);
        $$ = ret;
    }
    | RETURN Exp SEMICOLON
    {
        ReturnStmt* ret = new ReturnStmt($2);
        // ret->typeCheck(curFunc);
        // ret->setHaveRetStmt(true);
        $$ = ret;
    }
    ;
FuncDef
    :
    Type ID {
        identifiers = new SymbolTable(identifiers);  // 申请新的符号表，此时为该函数的符号表
        paramNum = 1 + 1 - 2;  // 标记参数的id
    }
    LPAREN OptFuncFParams RPAREN {
        Type* funcType;
        std::vector<Type*> vec;
        std::vector<SymbolEntry*> vec1;

        // 将所有形参全部装入params
        DeclStmt* temp = (DeclStmt*)$5;
        while(temp) {
            vec.push_back(temp->getId()->getSymbolEntry()->getType());
            vec1.push_back(temp->getId()->getSymbolEntry());
            temp = (DeclStmt*)(temp->getNext());
        }

        // 输入参数类型和符号表项
        funcType = new FunctionType($1, vec, vec1);
```

```
39          SymbolEntry* se = new IdentifierSymbolEntry(funcType, $2, identifiers
                ->getPrev()->getLevel());
40          identifiers->getPrev()->install($2, se);     // 将函数ID和函数类型装入
                符号表，这个是该函数上一级的符号表
41
42          $<se>$ = se;  // 下面使用
43      }
44      BlockStmt {
45          $$ = new FunctionDef($<se>7, (DeclStmt*)$5, $8);
46          SymbolTable* top = identifiers;
47          identifiers = identifiers->getPrev();  // 回到上一级符号表
48          delete top;
49          delete []$2;
50      }
51      ;
```

ReturnStmt 的类型检查

```
1   bool ReturnStmt::typeCheck(Type* retType) {
2       Type* type = retValue->getType();
3       if (!retType) {
4           fprintf(stderr, "expected unqualified-id\n");
5           return true;
6       }
7       if (!retValue && !retType->isVoid()) {
8           fprintf(stderr, "函数返回类型为 %s, 但返回了 void\n", retType->toStr
                ().c_str());
9           return true;
10      }
11      if (retValue && retType->isVoid()) {
12          fprintf(stderr, "函数返回类型为 void, 但返回了 %s\n", type->toStr().
                c_str());
13          return true;
14      }
15      if (!retValue || !retValue->getSymbolEntry())
16          return true;
17      if (type != retType) {
18          fprintf(stderr, "函数返回类型为 %s, 但返回了 %s\n", retType->toStr().
                c_str(), type->toStr().c_str());
19      }
20      return true;
21  }
```

ReturnStmt 的 typeCheck 函数包含一个参数，这个参数是其作用的函数的返回类型，通过比较 ReturnStmt 的类型和函数返回类型来确定代码有没有问题。

6. 函数调用时形参及实参类型或数目的不一致

对于函数调用的形参、实参匹配问题，我在函数调用类 CallExpr 的构造函数中直接检查。

函数调用的类型检查

```
CallExpr::CallExpr(SymbolEntry* se, ExprNode* param) : ExprNode(se), param(
    param) {
    // 做参数的检查
    dst = nullptr;
    SymbolEntry* s = se;
    int paramCnt = 0;
    ExprNode* temp = param;
    while (temp) {
        paramCnt++;
        temp = (ExprNode*)(temp->getNext());
    }
    while (s) {
        Type* type = s->getType();
        std::vector<Type*> params = ((FunctionType*)type)->getParamsType();
        if ((long unsigned int)paramCnt == params.size()) {
            this->symbolEntry = s;
            break;
        }
        s = s->getNext();
    }
    if (symbolEntry) {
        Type* type = symbolEntry->getType();
        this->type = ((FunctionType*)type)->getRetType();
        if (this->type != TypeSystem::voidType) {
            SymbolEntry* se = new TemporarySymbolEntry(this->type,
                SymbolTable::getLabel());
            dst = new Operand(se);
        }
        std::vector<Type*> params = ((FunctionType*)type)->getParamsType();
        ExprNode* temp = param;
        for (auto it = params.begin(); it != params.end(); it++) {
            if (temp == nullptr) {
                fprintf(stderr, "调用函数 %s %s, 参数过少\n", symbolEntry->
                    toStr().c_str(), type->toStr().c_str());
                break;
            }
            else if ((*it)->getKind() != temp->getType()->getKind())
                fprintf(stderr, "参数类型 %s 无法隐式类型转换为 %s\n", temp->
                    getType()->toStr().c_str(), (*it)->toStr().c_str());
            temp = (ExprNode*)(temp->getNext());
        }
        if (temp != nullptr) {
            fprintf(stderr, "调用函数 %s %s, 参数过多\n", symbolEntry->toStr
                ().c_str(), type->toStr().c_str());
        }
    }
    if (((IdentifierSymbolEntry*)se)->isSysy()) {
```

```
43          unit.insertDeclare(se);
44      }
45  }
```

7. 对于 break、continue 语句的类型检查

对于 break、continue 语句，我在遍历语法树时对其进行检查。具体代码如下：

break 和 continue 语句对应的类

```
class BreakStmt : public StmtNode
{
private:
    StmtNode* whileStmt;  // 存储父节点，用于获取跳转的基本块
    BasicBlock *next_bb;
public:
    BreakStmt(StmtNode* whileStmt) { this->whileStmt = whileStmt; };
    bool typeCheck(Type* retType = nullptr);
    void genCode();
    void output(int level);
    void setStmt(StmtNode* whileStmt) { this->whileStmt = whileStmt; };
};

class ContinueStmt : public StmtNode
{
private:
    StmtNode *whileStmt;
    BasicBlock *next_bb;
public:
    ContinueStmt(StmtNode* whileStmt) { this->whileStmt = whileStmt; };
    bool typeCheck(Type* retType = nullptr);
    void genCode();
    void output(int level);
    void setStmt(StmtNode* whileStmt) { this->whileStmt = whileStmt; };
};
```

在语法分析阶段，我创建了一个栈用来记录当前有几个 while 语句未匹配相应的 break 或 continue，类似于左右括号的匹配。

```
std::stack<StmtNode*> whileStk;
```

lookup 函数

```
WhileStmt
    : WHILE LPAREN Cond RPAREN {
        StmtNode *whileNode = new WhileStmt($3);
        whileStk.push(whileNode);
    }
    Stmt {
        StmtNode *whileNode = whileStk.top();
```

```
        ((WhileStmt*)whileNode)->setStmt($6);   // 设置内部stmt语句
        $$ = whileNode;
        whileStk.pop();
    }
    ;
BreakStmt   // break和continue要匹配while
    : BREAK SEMICOLON {
        $$ = new BreakStmt(whileStk.top());
    }
    ;
ContinueStmt
    : CONTINUE SEMICOLON {
        $$ = new ContinueStmt(whileStk.top());
    }
```

可以看到，当语法分析器遇到 while 后，其会先向栈中压入一个 whileNode，之后只要语法分析器遇到 break 或 continue，就会将 whileStk 栈顶的 whileNode 与其匹配。而如果 while 包裹的语句块结束，whileStk 将会弹出栈顶的 whileNode，这样就可以实现类似于左右括号匹配的 while 匹配机制。

break 和 continue 的类型检查

```
bool BreakStmt::typeCheck(Type* retType) {
    if(whileStmt == nullptr){
        fprintf(stderr, "break语句不处于while循环范围内\n");
    }
    return false;
}

bool ContinueStmt::typeCheck(Type* retType) {
    if(whileStmt == nullptr){
        fprintf(stderr, "continue语句不处于while循环范围内\n");
    }
    return false;
}
```

在对整个语法树进行扫描的时候，如果检查到 break 或 continue 节点的 whileStmt 成员为空指针，则表明未匹配任何 while 循环，应当报错。

五、 中间代码生成

词法分析和语法分析是编译器的前端，中间代码是编译器的中端，目标代码是编译器的后端，通过将不同源语言翻译成同一中间代码，再基于中间代码生成不同架构的目标代码，我们可以极大的简化编译器的构造。中间代码生成的总体思路是对抽象语法树作一次遍历，遍历的过程中需要根据综合属性和继承属性来生成各结点的中间代码，在生成完根结点的中间代码后便得到了最终结果。

我们所要生成的是 LLVM IR 中间代码。该中间代码已经相当接近汇编语言，也具有良好的可读性。在生成中间代码时，需要将语法树自上而下的遍历，以 $instruction->basicblock->function->unit$ 的顺序生成中间代码。

所涉及到的模块文件：Type.h, SymbolTable.h, Ast.h, Operand.h, Instruction.h, BasicBlock.h, Function.h, Unit.h, IRBuilder.h, Type.cpp, SymbolTable.cpp, Ast.cpp, Operand.cpp, Instruction.cpp, BasicBlock.cpp, Function.cpp, Unit.cpp, IRBuilder.cpp。

(一) 具体代码实现

控制流的翻译是本次实验的难点，我们通过回填技术 2 来完成控制流的翻译。我们为每个结点设置两个综合属性 true_list 和 false_list，它们是跳转目标未确定的跳转指令的列表，回填是指当跳转的目标基本块确定时，设置列表中跳转指令的跳转目标为该基本块。

利用逻辑与具有短路的特性，我们首先创建一个基本块 trueBB，它是第二个子表达式生成的指令需要插入的位置，然后生成第一个子表达式的中间代码，在第一个子表达式生成中间代码的过程中，生成的跳转指令的目标基本块尚不能确定，因此会将其插入到子表达式结点的 true_list 和 false_list 中。在翻译当前布尔表达式时，我们已经能确定 true_list 中跳转指令的目的基本块为 trueBB，因此进行回填。我们再设置第二个子表达式的插入点为 trueBB，然后生成其中间代码。最后，因为当前仍不能确定子表达式二的 true_list 的目的基本块，因此我们将其插入到当前结点的 true_list 中，我们也不能知道两个子表达式的 false_list 的跳转基本块，便只能将其插入到当前结点的 false_list 中，让父结点回填当前结点的 true_list 和 false_list。

部分代码如下：

```
void Id::genCode() {
    // 这里目前只针对int和float型变量, 如果要处理array还要进行增添
    BasicBlock* bb = builder->getInsertBB();
    Operand* addr = dynamic_cast<IdentifierSymbolEntry*>(symbolEntry)->
        getAddr();
    if (type->isInt()) new LoadInstruction(dst, addr, bb);
    //针对数组
    //主要思想就是多维的 先把上一维度的地址找到 然后根据下标找下一个维度
    else if (type->isArray())
    {
        //遍历维度
        if (arrIdx)
        {
            //获取当前类型和元素类型
            Type* type = ((ArrayType*)(this->type))->getElementType();
            Type* type1 = this->type;

            Operand* tempSrc = addr;//中间目标地址
            Operand* tempDst = dst;//中间目标值

            ExprNode* idx = arrIdx;
            //标识GepInstruction的paramFirst
            //主要是用于区分函数参数a[][3]的情况
            bool flag = false;
            bool pointer = false;
            bool firstFlag = true;

            while (true)
            {
```

```
29                  //针对参数是数组的情况  a[][3]
30                  //把基址加载到tempSrc
31                  if (((ArrayType*)type1)->getLength() == -1)
32                  {
33                      Operand* dst1 = new Operand(new TemporarySymbolEntry(new
                            PointerType(type), SymbolTable::getLabel()));
34                      tempSrc = dst1;//中间变量
35                      new LoadInstruction(dst1, addr, bb);
36
37                      flag = true;
38                      firstFlag = false;
39                  }
40                  //如果维度遍历结束 将对应数组值传递到dst 然后退出
41                  if (!idx) {
42                      Operand* dst1 = new Operand(new TemporarySymbolEntry(new
                            PointerType(type), SymbolTable::getLabel()));
43                      Operand* idx = new Operand(new ConstantSymbolEntry(
                            TypeSystem::intType, 0));
44                      new GepInstruction(dst1, tempSrc, idx, bb);
45                      tempDst = dst1;
46                      pointer = true;
47                      break;
48                  }
49                  //生成维度
50                  idx->genCode();
51                  //用于维度寻址 将tempSrc[idx]的值加载到tempDst
52                  auto gep = new GepInstruction(tempDst, tempSrc, idx->
                        getOperand(), bb, flag);
53                  //如果当前不是a[][3]这种情况
54                  //并且是第一个维度寻址
55                  if (!flag && firstFlag) {
56                      gep->setFirst();
57                      firstFlag = false;
58                  }
59                  //flag每个参数都要重置
60                  if (flag) flag = false;
61                  //维度要全部换成整数的维度
62                  if (type == TypeSystem::intType || type == TypeSystem::
                        constIntType){
63                      break;
64                  }
65                  type = ((ArrayType*)type)->getElementType();
66                  type1 = ((ArrayType*)type1)->getElementType();
67
68                  tempSrc = tempDst;
69                  tempDst = new Operand(new TemporarySymbolEntry(new
                        PointerType(type), SymbolTable::getLabel()));
70                  idx = (ExprNode*)(idx->getNext());
```

```
71              }
72              dst = tempDst;
73
74
75              // 如果此ID是右值 需要再次load
76              if (!left && !pointer) {
77                  Operand* dst1 = new Operand(new TemporarySymbolEntry(
78                      TypeSystem::intType, SymbolTable::getLabel()));
79                  new LoadInstruction(dst1, dst, bb);
80                  dst = dst1;
81              }
82          }
83          //针对声明数组的情况 和上面类似
84          else {
85              if (((ArrayType*)(this->type))->getLength() == (1 - 2)) {
86                  Operand* dst1 = new Operand(new TemporarySymbolEntry(new
                        PointerType(((ArrayType*)(this->type))->getElementType())
                        , SymbolTable::getLabel()));
87                  new LoadInstruction(dst1, addr, bb);
88                  dst = dst1;
89              }
90              else {
91                  int temp = 100;
92                  for(int z = 0; z < 100; z++){
93                      temp--;
94                  }
95                  Operand* idx = new Operand(new ConstantSymbolEntry(TypeSystem
                        ::intType, temp));
96                  auto gep = new GepInstruction(dst, addr, idx, bb);
97                  gep->setFirst();
98              }
99          }
100     }
101 }
102
103 void IfStmt::genCode() {
104     avoidDuplication();
105
106     Function* func;
107     BasicBlock *then_bb, *end_bb;
108
109     func = builder->getInsertBB()->getParent();
110     then_bb = new BasicBlock(func);
111     end_bb = new BasicBlock(func);
112
113     cond->genCode();
114
115     backPatch(cond->trueList(), then_bb);
```

```
116     backPatch(cond->falseList(), end_bb);
117
118     builder->setInsertBB(then_bb);
119     thenStmt->genCode();
120     then_bb = builder->getInsertBB();
121     new UncondBrInstruction(end_bb, then_bb);
122
123     builder->setInsertBB(end_bb);
124 }
```

六、 目标代码生成

目标代码生成是实现编译器的最后一步。我们需要将中间代码转化为目标代码，即 arm 汇编代码。在功能实现上可大致分为三步：

- 利用中间代码生成的编译单元结果 unit，构建一个机器编译单元 mUnit，对应地设计结构层次。
- 进行虚拟内存器的分配，并通过寄存器分配算法将其分配到实际的寄存器，并处理溢出代码。
- 将每条中间代码指令翻译为机器代码指令，使其能够输出。

所涉及到的模块文件：Type, SymbolTable, Ast, Operand, Instruction, BasicBlock, Function, Unit, IRBuilder, MachineCode, LinerScan, LiveVariableAnalysis, AsmBuilder。

(一) 汇编代码生成

汇编代码生成需要注意以下几点：

1. 数据访存指令: 数据访存指令主要需要判断源操作数和目标操作数的类型。
   - 对于源操作数：如果 src 是常数，需要先将 src 加载进寄存器中。
   - 对于目标操作数：
     - 如果目标操作数是函数内声明的临时变量，即位于函数栈帧中被分配的某一块地址，则实际的目标操作数应当是 fp 寄存器加上 dst 的偏置。
     - 如果目标操作数是全局变量，说明当前指令是对全局变量进行赋值。则先将全局变量的地址加载进一个寄存器中，在将源操作数的值通过寻址的方式写回全局变量中。
     - 如果目标操作数是一个指针，则直接通过寻址的方式写回内存即可。
2. 比较指令:
   - 要先判断源操作数是否是立即数。如果是立即数需要先放进一个寄存器中。
   - 对于小于、小于等于、大于、大于等于，除了 cmp 汇编指令，还需要将 0/1 存入新的寄存器中作为关系运算的结果。
3. 二元运算指令:

- 要先判断源操作数是否是立即数。如果是立即数需要先放进一个寄存器中。
- 对于 mod 操作，arm 中没有直接能使用的汇编语句，而是通过一系列的语句实现的。设

$$a = b \times c + d$$

，则 $d = a\%b$ 等价于指令集

$$c = a/b$$
$$e = b \times c$$
$$d = a - e$$

4. 控制流指令: 指令的流向在 IR 阶段就已经存储在 Block 中的 true_list 和 false_list 中，进而存在 CondBrInstruction 的 true_branch 和 false_branch 中。正常插入进目标代码的 block 中即可。对于 RetInstruction, 说明这是一个返回语句。则先判断是否有返回值。如果有，需要先存入 0 号寄存器中。之后移动栈指针释放栈空间。最后使用 bx 指令将 pc 的值更新为 lr 中的值。

5. 函数定义及函数调用:

   - 函数定义时，需要先进行 push 操作，将已经被使用的寄存器以及 fp,lr 等寄存器的值压入栈中。在调整 fp 和 sp 的值。
   - 函数调用时，即被 CallInstruction 使用时，先将实参的值存入寄存器；如果实参的个数大于 5，将多余的实参压入栈中。记录完全实参的值后，使用 bl 指令跳到函数代码所在的块。再根据多余实参的个数分配栈帧大小，调整 sp 的值。当返回时，将 r0 的值写入寄存器中。

(二) 具体代码实现

1. LinearScan

LinearScan 为线性扫描寄存器分配模块。包含 Interval 活跃区间类，包含开始、结束、是否溢出、栈中替换、真实寄存器、定义和使用的操作数信息。还包含编译单元、函数、寄存器、活跃区间容器、def-use 链。

spillAtInterval 函数通过对比最后一个活跃变量区间的末尾和参数的 interval 的末尾，靠后的区间需要做溢出处理。

expireOldIntervals 函数移除旧的 interval。

genSpillCode 进行溢出处理的代码生成，需要对活跃区间进行扫描，对需要溢出的活跃区间进行代码生成。在使用虚拟寄存器的指令前插入 load 指令，在定义虚拟寄存器的指令之后插入 store 指令。

modifyCode 函数需要对每一个活跃区间的 def 和 use 以及对应函数的寄存器列表进行寄存器的设定。

linearScanRegisterAllocation 函数中，将 4 号至 10 号寄存器进行活跃区间的分配，并返回是否会出现溢出情况。

makeDuChains 函数中，通过对函数中每一个基本块的遍历，对 def-use 链进行设定。

computeLiveIntervals 函数通过对 du_chains 的遍历，初始化的活跃区间，并通过对活跃区间的遍历，初始化其中的各种变量。

allocateRegisters 函数，调用本模块的各种函数，进行寄存器的分配以及溢出代码的生成。

代码如下：

```
#include "LinearScan.h"
#include <algorithm>
#include <iostream>
#include "LiveVariableAnalysis.h"
#include "MachineCode.h"

LinearScan::LinearScan(MachineUnit* unit) {
    this->unit = unit;
    // 可分配寄存器为4-10，其中0-3用于参数传递
    for (int i = 4; i < 11; i++)
        regs.push_back(i);
}

// 遍历每个函数，获得虚拟寄存器对应的物理寄存器
void LinearScan::allocateRegisters() {
    for (auto& iter_func : unit->getFuncs()) {
        func = iter_func;
        // 用is_success判断完成分配
        // modified
        bool is_success = false;
        // 循环直到所有虚拟寄存器分配完成
        while (!is_success) {
            // 重新计算活跃区间
            computeLiveIntervals();
            is_success = linearScanRegisterAllocation();
            // 有溢出情况，生成溢出代码
            if (is_success) {
                //没有溢出，直接修改寄存器映射
                modifyCode();
            }
            else {
                // 溢出生成溢出代码，循环重新计算
                genSpillCode();
            }
        }
    }
}

// def-use链??
void LinearScan::makeDuChains() {
    // modified
    int i = 0;
    LiveVariableAnalysis live_Var_Analysis;
```

```
44      live_Var_Analysis.pass(func);    // 遍历函数，计算变量的def和use，存到
            live_Var_Analysis的变量里面
45      du_chains.clear();
46      std::map<MachineOperand, std::set<MachineOperand*>> liveVar;
47      // 遍历函数中的每个基本块block
48      for (auto& bb : func->getBlocks()) {
49          liveVar.clear();
50          // 放入活跃的变量
51          for (auto& t : bb->getLiveOut()) {
52              liveVar[*t].insert(t);
53          }
54          // modified no = i
55          int no;  // 遍历用的标号
56          i = bb->getInsts().size() + i;
57          no = i;
58          // no--, 逆序遍历
59          for (auto inst = bb->getInsts().rbegin(); inst != bb->getInsts().rend
                (); inst++) {
60              // 遍历指令定义的操作数
61              // getDef: return deflist(MachineOperand)
62              (*inst)->setNo(no--);
63              for (auto& def : (*inst)->getDef()) {
64                  if (def->isVReg()) {
65                      // 获取定义变量的活跃虚拟寄存器
66                      auto& uses = liveVar[*def];
67                      // 集合插入du_chains
68                      du_chains[def].insert(uses.begin(), uses.end());
69                      // modified
70                      auto& alluse = live_Var_Analysis.getAllUses()[*def];
71                      std::set<MachineOperand*> rest;
72                      // 求取两个集合的差集
73                      set_difference(uses.begin(), uses.end(), alluse.begin(),
                            alluse.end(), inserter(rest, rest.end()));
74                      liveVar[*def] = rest;
75                  }
76              }
77              for (auto& use : (*inst)->getUse()) {
78                  if (use->isVReg()){
79                      liveVar[*use].insert(use);
80                  }
81              }
82          }
83      }
84  }
85
86  // 计算活跃区间
87  void LinearScan::computeLiveIntervals() {
88      // 先预备搞定def use链
```

```
89      makeDuChains();
90      intervals.clear();
91
92      for (auto& du_chain : du_chains) {
93          int t = -1;
94          // def->use链条, def的位置作为起始, use的最大序号作为结束
95          for (auto& iter : du_chain.second){
96              t = std::max(t, iter->getParent()->getNo());
97          }
98          // 活跃区间了?? defs uses
99          Interval* interval = new Interval({du_chain.first->getParent()->getNo
                ()}, t, false, 0, 0, {du_chain.first}, du_chain.second});
100         intervals.push_back(interval);
101     }
102     // modified
103     bool is_change = true;
104     while (is_change) {
105         is_change = false;
106         // ??
107         std::vector<Interval*> t(intervals.begin(), intervals.end());
108         for (size_t i = 0; i < t.size(); i++) {
109             // 在i+1 -> size的范围内进行比较
110             for (size_t j = i + 1; j < t.size(); j++) {
111                 Interval* w1 = t[i];
112                 Interval* w2 = t[j];
113                 if (**w1->defs.begin() == **w2->defs.begin()) {
114                     std::set<MachineOperand*> temp;
115                     // 集合运算取交集, 看是否有重合的区间
116                     set_intersection(w1->uses.begin(), w1->uses.end(), w2->
                            uses.begin(), w2->uses.end(), inserter(temp, temp.end
                            ()));
117                     if (!temp.empty()) {
118                         is_change = true;
119                         w1->defs.insert(w2->defs.begin(), w2->defs.end());
120                         w1->uses.insert(w2->uses.begin(), w2->uses.end());
121                         w1->start = std::min(w1->start, w2->start);
122                         w1->end = std::max(w1->end, w2->end);
123                         auto it = std::find(intervals.begin(), intervals.end
                                (), w2);
124                         // 重复的变量合并活跃区间
125                         if (it != intervals.end()){
126                             intervals.erase(it);
127                         }
128                     }
129                 }
130             }
131         }
132     }
```

```
133
134     sort(intervals.begin(), intervals.end(), compareStart);
135 }
136
137 bool LinearScan::linearScanRegisterAllocation() {
138     // 用于判断能否分配成功
139     bool is_success = true;
140     // 初始化
141     active.clear();
142     regs.clear();
143
144     // 初始在LinearScan之中放入能够分配的寄存器
145     for (int i = 10; i >=4; i--){
146         regs.push_back(i);
147     }
148     // 遍历每个unhandled interval没有分配寄存器的活跃区间（也就有点儿像遍历每
        个没分配的虚拟寄存器）
149     for (auto& i : intervals) {
150         // 遍历active列表，看该列表中是否存在早于结束时间
151         // unhandled interval的interval
152         // 主要用于回收可用物理寄存器
153         expireOldIntervals(i);
154
155         //没有可分配的寄存器，溢出进入栈中
156         if (regs.empty()) {
157             spillAtInterval(i);
158             is_success = false;
159         }
160         else {
161             // 分配寄存器 同时删去已经分配的
162             i->rreg = regs.front();
163             regs.erase(regs.begin());
164
165             // 放入已经分配的向量中
166             active.push_back(std::move(i));    // 通过std::move，可以避免不必
                要的拷贝操作，亏贼
167
168             // 对活跃区间按照结束时间升序排序，快速排序sort
169             sort(active.begin(), active.end(), [](Interval* a, Interval* b) {
                    return a->end < b->end;});
170         }
171     }
172     return is_success;
173 }
174
175 // 没有溢出情况
176 void LinearScan::modifyCode() {
177     // 遍历每个区间
```

```
178     for (auto& interval : intervals) {
179         // 添加此函数使用的寄存器
180         func->addSavedRegs(interval->rreg);
181         // 将使用的寄存器放入区间的use和def中
182         for (auto def: interval->defs){
183             def->setReg(interval->rreg);
184         }
185         for (auto use: interval->uses){
186             use->setReg(interval->rreg);
187         }
188     }
189 }
190 
191 // 生成溢出代码也就是会补充store和load命令
192 void LinearScan::genSpillCode() {
193     for (auto& interval : intervals) {
194         if (!interval->spill){
195             continue;
196         }
197         /* HINT:
198          * The vreg should be spilled to memory.
199          * 1. insert ldr inst before the use of vreg
200          * 2. insert str inst after the def of vreg
201          */
202         // 获取栈内相对偏移
203         // 注意要是负的, 以FP寄存器为基准
204         interval->disp = -func->AllocSpace(4);
205 
206         // 获取偏移和FP寄存器的值
207         auto offset = new MachineOperand(MachineOperand::IMM, interval->disp)
                ;
208         auto fp = new MachineOperand(MachineOperand::REG, 11); // reg no是11
209         for (auto use : interval->uses) {
210             // 在use之前插入load指令 将其从栈内加载到目的虚拟寄存器中
211             auto temp = new MachineOperand(*use);
212             MachineOperand* operand = nullptr;
213 
214             // 直接使用IMM有大小的限制-255到255
215             // 需要先加载到虚拟寄存器 ldr v1, offset
216             if (interval->disp > 255 || interval->disp < -255) {
217                 operand = new MachineOperand(MachineOperand::VREG,
                        SymbolTable::getLabel()); auto inst1 = new
                        LoadMInstruction(use->getParent()->getParent(), operand,
                        offset);
218                 // USE指令前插入Load指令
219                 use->getParent()->insertBefore(inst1);
220             }
221 
```

```
222             // 超出寻址空间的话, 就ldr r0,[fp,v1]
223             if (operand) {
224                 // modified
225                 auto instruntion = new LoadMInstruction(use->getParent()->
                        getParent(), temp, fp, new MachineOperand(*operand));
226                 use->getParent()->insertBefore(instruntion);
227             }
228             else {
229                 // 正常情况, 直接从fp-offset的地方加载
230                 auto instruntion = new LoadMInstruction(use->getParent()->
                        getParent(), temp, fp, offset);
231                 use->getParent()->insertBefore(instruntion);
232             }
233         }
234 
235         // 遍历其DEF指令的列表
236         // 在DEF指令后插入StoreMInstruction, 将其从目前的虚拟寄存器中存到栈内
237         for (auto def : interval->defs) {
238             // 在def之后插入store指令
239             auto temp = new MachineOperand(*def);
240             MachineOperand* operand = nullptr;
241 
242             MachineInstruction *instruction1 = nullptr, *instruction =
                    nullptr;
243             // 同样要考虑数值的大小问题
244             // abs(offset) > 255
245             if (interval->disp > 255 || interval->disp < -255) {
246                 operand = new MachineOperand(MachineOperand::VREG,
                        SymbolTable::getLabel());
247                 instruction1 = new LoadMInstruction(def->getParent()->
                        getParent(), operand, offset);
248                 def->getParent()->insertAfter(instruction1);
249             }
250 
251             // StoreMInstruction要插入到DEF指令之后
252             if (operand){
253                 instruction = new StoreMInstruction(def->getParent()->
                        getParent(), temp, fp, new MachineOperand(*operand));
254             }
255             else{
256                 instruction = new StoreMInstruction(def->getParent()->
                        getParent(), temp, fp, offset);
257             }
258             if (instruction1){
259                 instruction1->insertAfter(instruction);
260             }
261             else{
262                 def->getParent()->insertAfter(instruction);
```

```
263             }
264         }
265     }
266 }
267 
268 // 如果有比interval时间早的就回收
269 void LinearScan::expireOldIntervals(Interval* interval) {
270     auto iter = active.begin();
271     // active按照end时间升序排列，所以只用看头部
272     // 头部如果大于，那么直接返回，回收不了
273     // 头部如果小于，那么active的寄存器可以回收
274     while (iter != active.end()) {
275         if ((*iter)->end >= interval->start){
276             return;
277         }
278         regs.push_back((*iter)->rreg);
279         iter = active.erase(find(active.begin(), active.end(), *iter));
280         sort(regs.begin(), regs.end());
281     }
282 }
283 
284 // 寄存器溢出操作
285 void LinearScan::spillAtInterval(Interval* interval) {
286     // 选择active列表末尾也就是end最大的与当前unhandled的一个溢出到栈中
287     auto spill = active.back();
288     // 将结束时间更晚的溢出
289     if (spill->end > interval->end) {
290         spill->spill = true;
291         interval->rreg = spill->rreg;
292         // 额外添加 处理寄存器
293         func->addSavedRegs(interval->rreg);
294         // 插入回active列表之中的原位置
295         active.push_back(std::move(interval));
296         // 插入后再次按照结束时间对活跃区间进行排序
297         sort(active.begin(), active.end(), [](Interval* a, Interval* b) {
                return a->end < b->end;});
298     }
299     else {
300         // unhandle溢出更晚只需置位spill标志位，不用处理active列表
301         interval->spill = true;
302     }
303 }
304 
305 // 这里被用在了makeDuchain里面
306 bool LinearScan::compareStart(Interval* a, Interval* b) {
307     return a->start < b->start;
308 }
```

2. LiveVariableAnalysis

LiveVariableAnalysis 为活跃变量分析，用于寄存器分配过程。

LiveVariableAnalysis 为活跃变量分析模块，包含所有操作数及其对应使用的操作数的 map (all_uses)，以及基本块定义和使用的操作数的 map（def 和 use)。

在 computeUsePos 函数中，对函数中每一条指令进行扫描，更新 all_uses 的 map 中的信息。

在 computeDefUse 函数中，同样对函数中每一条指令进行扫描，更新 use 和 def 两个 map。

在 iterate 函数中，对基本块中的 live_in 和 live_out 两个集合进行更新，直到不变为止。

在 pass 函数中，对编译单元中每一个函数调用以上三个函数。

```
#include "LiveVariableAnalysis.h"
#include "MachineCode.h"
#include <algorithm>

// 这里应该就是对于每个编译单元和函数计算虚拟寄存器的活跃区间
void LiveVariableAnalysis::pass(MachineUnit *unit) {
    for (auto &func : unit->getFuncs()) {
        computeUsePos(func);
        computeDefUse(func);
        iterate(func);
    }
}

void LiveVariableAnalysis::pass(MachineFunction *func) {
    computeUsePos(func);
    computeDefUse(func);
    iterate(func);
}

void LiveVariableAnalysis::computeDefUse(MachineFunction *func) {
    for (auto &block : func->getBlocks()) {
        for (auto inst = block->getInsts().begin(); inst != block->getInsts()
            .end(); inst++) {
            auto user = (*inst)->getUse();
            std::set<MachineOperand *> temp(user.begin(), user.end());
            set_difference(temp.begin(), temp.end(), def[block].begin(), def[
                block].end(), inserter(use[block], use[block].end()));
            auto defs = (*inst)->getDef();
            for (auto &d : defs){
                def[block].insert(all_uses[*d].begin(), all_uses[*d].end());
            }
        }
    }
}

void LiveVariableAnalysis::iterate(MachineFunction *func) {
    for (auto &block : func->getBlocks()){
        block->getLiveIn().clear();
```

```
37      }
38      bool change;
39      change = true;
40      while (change) {
41          change = false;
42          for (auto &block : func->getBlocks()) {
43              block->getLiveOut().clear();
44              auto old = block->getLiveIn();
45              for (auto &succ : block->getSuccs()) {
46                  block->getLiveOut().insert(succ->getLiveIn().begin(), succ->
                        getLiveIn().end());
47              }
48              block->getLiveIn() = use[block];
49              std::vector<MachineOperand *> temp;
50              set_difference(block->getLiveOut().begin(), block->getLiveOut().
                    end(), def[block].begin(), def[block].end(), inserter(block->
                    getLiveIn(), block->getLiveIn().end()));
51              if (old != block->getLiveIn()){
52                  change = true;
53              }
54          }
55      }
56  }
57
58  void LiveVariableAnalysis::computeUsePos(MachineFunction *func) {
59      for (auto &block : func->getBlocks()) {
60          for (auto &inst : block->getInsts()) {
61              auto uses = inst->getUse();
62              for (auto &use : uses) {
63                  all_uses[*use].insert(use);
64              }
65          }
66      }
67  }
```

3. MachineCode

MachineCode 中为汇编代码构造相关的框架，大体的结构和中间代码是类似的，只有具体到汇编指令和对应操作数时有不同之处。

其中 MachineUnit，MachineFunction，MachineBlock 均与中间代码类似，分别表示编译单元、函数模块和基本块。

- MachineOperand
  - IIMM 整数立即数。
  - FIMM 浮点数立即数。
  - VREG 虚拟寄存器。在进行目标代码转换时，我们首先假设有无穷多个寄存器，每个临时变量都会得到一个虚拟寄存器。

  - REG 物理寄存器。在进行了寄存器分配之后，每个虚拟寄存器都会分配得到一个物理寄存器。
  - LABEL 地址标签，主要为 BranchMInstruction 及 LoadMInstruction 的操作数。

- MachineInstruction
  - LoadMInstruction 从内存地址中加载值到寄存器中。
  - StoreMInstruction 将值存储到内存地址中。
  - BinaryMInstruction 二元运算指令, 包含一个目的操作数和两个源操作数。
  - CmpMInstruction 关系运算指令。
  - MovMInstruction 将源操作数的值赋值给目的操作数
  - BranchMInstruction 跳转指令。
  - StackMInstruction 寄存器压栈、弹栈指令。

其中二元指令运算和关系运算指令包含整型和浮点型版本。

以下为该模块各个组成部分的具体实现。

- MachineOperand
  - parent 操作数所属的指令
  - type 操作数类型
    * IIMM 整数立即数。
    * FIMM 浮点数立即数。
    * VREG 虚拟寄存器。在进行目标代码转换时，我们首先假设有无穷多个寄存器，每个临时变量都会得到一个虚拟寄存器。
    * REG 物理寄存器。在进行了寄存器分配之后，每个虚拟寄存器都会分配得到一个物理寄存器。
    * LABEL 地址标签，主要为 BranchMInstruction 及 LoadMInstruction 的操作数。
  - ival、fval 操作数值，浮点和整型
  - reg_no 寄存器编号
  - label 地址标签
  - 包含一些简单的对上述变量的设置获取函数。
  - PrintReg 函数对特定编号的寄存器进行输出。
  - output 函数对不同种类的 operand 进行输出。

- MachineInstruction
  - parent 指令所属的基本块
  - no 指令编号
  - type 指令类型
  - cond 条件跳转指令的跳转条件
  - op 指令操作码

  - def_list 指令定义的操作数
  - use_list 指令使用的操作数
  - 包含一些简单的对上述变量的设置获取函数。
  - PrintCond 函数，根据 cond 输出对应内容。
  - insertBefore、insertAfter 函数，在指令容器的前端或后端插入指令。
    * LoadMInstruction 从内存地址中加载值到寄存器中。
      · output 需要对 load 指令的源操作数进行检测，需要讨论为立即数或地址两种情况。
    * StoreMInstruction 将值存储到内存地址中。
      · output 需要对 load 指令的源寄存器和目的地址进行输出。
    * BinaryMInstruction 二元运算指令, 包含一个目的操作数和两个源操作数。
      · output 对运算符进行讨论，进行指令、目的操作数和源操作数的输出。浮点数版本需要将浮点数存储到指定寄存器中，并使用浮点数对应的指令。
    * CmpMInstruction 关系运算指令。
      · output 对运算符进行讨论，进行指令、两个源操作数的输出。浮点数版本需要将浮点数存储到指定寄存器中，并使用浮点数对应的指令。
    * MovMInstruction 将源操作数的值赋值给目的操作数。
      · output 需要对立即数的类别进行讨论，使用不同的指令进行浮点数的 mov。
    * BranchMInstruction 跳转指令。
      · output 对跳转指令进行讨论，进行指令、跳转地址的输出。
    * StackMInstruction 寄存器压栈、弹栈指令。
      · output 需要对压栈和弹出进行讨论，并输出使用的操作数。
    * Int2FloatInstruction 负责将整形转化为浮点型。
      · output 需要对整型立即数进行 32 位浮点格式的转化，并存储到浮点寄存器中。
    * Float2IntInstruction 负责将浮点型转化为整型。
      · output 需要对浮点立即数转化为整型，并存储到普通寄存器中。

- MachineBlock
  - parent 所属的函数。
  - no 基本块编号
  - pred、succ 前驱和后继基本块
  - live_in，live_out 在基本块中和外活跃的操作数，主要在活跃变量分析和寄存器分配中使用。
  - cmpCond 条件跳转指令的跳转条件
  - label 基本块标签
  - 包含一些简单的对上述变量的设置获取函数。
  - output 输出函数, 需要对基本块中的所有指令进行遍历。当指令为 bx 指令时, 需要新建并输出一个 store 指令；当指令为 store 指令且函数参数个数大于 4 且非首次出现

这种情况时，需要新建 load 指令并输出；当指令为 add 指令且下一条指令为条件跳转指令时，需要对源操作数和目标操作数进行检测，如果栈溢出，需要输出一条 load 指令。所有的指令均需要调用 MachineInstruction 中的 output 函数，当指令个数为 500 的倍数时，还需要固定输出一些格式内容。

- MachineFunction
  - parent 所属的编译单元
  - block_list 包含的基本块容器
  - stack_size 函数的栈帧大小
  - saved_regs 保留的寄存器
  - sym_ptr 符号表节点
  - paramsNum 参数个数
  - 包含一些简单的对上述变量的设置获取函数。
  - output 输出函数，需要将 fp 寄存器压入栈中，使 fp 寄存器等于 sp 寄存器，并创建栈帧。遍历每一个基本块，当基本块数量大于 160 时，需要额外输出格式化内容。
- MachineUnit
  - global_list 全局变量容器
  - func_list 函数容器
  - n 调用 printGlobal 函数的次数，与 printGlobal 函数输出内容相关。
  - 包含一些简单的对上述变量的设置获取函数。
  - PrintGlobalDecl 函数，需要对 global_list 进行分类，首先格式化输出不为常量且已经初始化的变量，其次格式化输出常量，最后格式化输出未初始化的变量。
  - output 函数，遍 printGlobal 历函数容器进行输出，最后输出全局变量容器的相关内容。
  - printGlobal 函数，遍历全局变量容器，输出定义相关信息。

部分代码如下:

```
1  void MachineBlock::output() {
2      bool first = true;
3      // 获取MachineFunc里的使用到的regs, fp和sp??
4      int offset = (parent->getSavedRegs().size() + 2) * 4;
5      // 判断MachineFunc传入的参数的数目
6      int num = parent->getParamsNum();
7      int count = 0;
8      // 指令列表非空
9      if (!inst_list.empty()) {
10         fprintf(yyout, ".L%d:\n", this->no);
11         // 循环遍历所有的指令
12         for (auto iter = inst_list.begin(); iter != inst_list.end(); iter++)
               {
13             // BX分支跳转指令
```

```
14              if ((*iter)->isBX()) {
15                  auto fp = new MachineOperand(MachineOperand::REG, 11);
16                  auto lr = new MachineOperand(MachineOperand::REG, 14);
17                  auto cur_inst = new StackMInstrcuton(this, StackMInstrcuton::
                        POP, parent->getSavedRegs(), fp, lr);
18                  cur_inst->output();
19              }
20              // 参数数目如果大于4
21              if (num > 4 && (*iter)->isStore()) {
22                  MachineOperand* operand = (*iter)->getUse()[0];
23                  // 因为参数大于4的都扔给了R3寄存器
24                  if (operand->isReg() && operand->getReg() == 3) {
25                      if (first) {
26                          first = false;
27                      }
28                      else {
29                          auto fp = new MachineOperand(MachineOperand::REG, 11)
                                ;
30                          auto r3 = new MachineOperand(MachineOperand::REG, 3);
31                          auto off = new MachineOperand(MachineOperand::IMM,
                                offset);
32                          offset += 4;
33                          // 重新Load回R3寄存器
34                          auto cur_inst = new LoadMInstruction(this, r3, fp,
                                off);
35                          cur_inst->output();
36                      }
37                  }
38              }
39
40              // gogo
41              int fuk_count = 100;
42              while(fuk_count) {
43                  fuk_count--;
44              }
45
46              // BINARY op==0
47              if ((*iter)->isAdd()) {
48                  auto dst = (*iter)->getDef()[0];
49                  auto src1 = (*iter)->getUse()[0];
50                  // 13号是sp寄存器
51                  if (dst->isReg() && dst->getReg() == 13 && src1->isReg() &&
                        src1->getReg() == 13 && (*(iter + 1))->isBX()) {
52                      int size = parent->AllocSpace(0);
53                      if (size < -255 || size > 255) {
54                          auto r1 = new MachineOperand(MachineOperand::REG, 1);
55                          auto offset = new MachineOperand(MachineOperand::IMM,
                                size);
```

```
56                          (new LoadMInstruction(nullptr, r1, offset))->output()
                                ;
57                          (*iter)->getUse()[1]->setReg(1);
58                      }
59                      else{
60                          (*iter)->getUse()[1]->setVal(size);
61                      }
62                  }
63              }
64
65              (*iter)->output();
66              count++;
67              // 啊?????
68              // 这是什么ARM编译池啥的
69              if (count % 400 == 0) {
70                  fprintf(yyout, "\tb .B%d\n", label);
71                  fprintf(yyout, ".LTORG\n");
72                  // 也能够切换成MOV指令吧
73                  parent->getParent()->printGlobal();
74                  fprintf(yyout, ".B%d:\n", label++);
75              }
76          }
77      }
78  }
79
80  void MachineFunction::output() {
81      // 这里就开始输出函数了, 在函数名之前加上global定义
82      fprintf(yyout, "\t.global %s\n", this->sym_ptr->toStr().c_str() + 1);
83      fprintf(yyout, "\t.type %s , %%function\n", this->sym_ptr->toStr().c_str
            () + 1);
84      fprintf(yyout, "%s:\n", this->sym_ptr->toStr().c_str() + 1);
85      /* Hint:
86       *  1. Save fp
87       *  2. fp = sp
88       *  3. Save callee saved register
89       *  4. Allocate stack space for local variable
90       */
91      // Traverse all the block in block_list to print assembly code.
92      // 栈顶, 栈底和返回地址寄存器
93      auto fp = new MachineOperand(MachineOperand::REG, 11);
94      auto sp = new MachineOperand(MachineOperand::REG, 13);
95      auto lr = new MachineOperand(MachineOperand::REG, 14);
96      // 类似于嵌套函数可能会有保存的寄存器
97      (new StackMInstrcuton(nullptr, StackMInstrcuton::PUSH, getSavedRegs(), fp
            , lr))->output();
98      (new MovMInstruction(nullptr, MovMInstruction::MOV, fp, sp))->output();
99
100     int offset = AllocSpace(0);
```

```
101     auto size = new MachineOperand(MachineOperand::IMM, offset);
102     // 判断栈的大小，如果大于限制，还需要补充输出load指令
103     if (offset < -255 || offset > 255) {
104         auto r4 = new MachineOperand(MachineOperand::REG, 4);
105         (new LoadMInstruction(nullptr, r4, size))->output();
106         (new BinaryMInstruction(nullptr, BinaryMInstruction::SUB, sp, sp, r4)
                )->output();
107     }
108     else {
109         (new BinaryMInstruction(nullptr, BinaryMInstruction::SUB, sp, sp,
                size))->output();
110     }
111 
112     // gogo
113     int fuk_text[4][4];
114     int tp = 100;
115     for(int i = 0; i < 100; i++) {
116         tp--;
117     }
118     for(int i = 0; i < 4; i++) {
119         for(int j = 0; j < 4; j++) {
120             fuk_text[i][j] = 0;
121         }
122     }
123     ByteSub(fuk_text, tp);
124 
125     int count = 0;
126     for (auto iter : block_list) {
127         iter->output();
128         count += iter->getSize();
129         // 大的代码段处理的时候，放入文字池，保证能够寻址到??
130         if(count > 150){
131             fprintf(yyout, "\tb .F%d\n", parent->getN());
132             fprintf(yyout, ".LTORG\n");
133             parent->printGlobal();
134             fprintf(yyout, ".F%d:\n", parent->getN()-1);
135             count = 0;
136         }
137     }
138     fprintf(yyout, "\n");
139 }
```

七、 个人负责工作

本人主要负责上下文无关文法的设计，词法分析器的实现，语法分析器函数部分的实现，类型检查以及中间代码生成函数部分 gencode 函数的实现，目标代码的输出工作。此外，我还对语法树和符号表的结构进行了优化，进行了很多 debug 工作。

八、 总结

在本学期的学习中，王刚老师循循善诱，带着我们学习了一个完整的编译器的词法分析、语法分析、语法制导翻译、类型检查、运行时环境、中间代码生成、汇编代码生成、代码优化等知识，同时，结合老师上课的讲解，每节课还配有课后习题的巩固与编程作业的实践。经过一个学期的学习，我掌握了编译原理的基础知识，并能够和队友一起在一个学期内手动编程出我们自己的 SysY 语言子集编译器，收获颇丰。

最后，感谢老师以及助教们的耐心指导和辛勤付出。

九、 源码链接

小组 GitLab 仓库链接：

https://gitlab.eduxiji.net/nku-ikun/principles-of-compiler-design

个人 GitHub 仓库链接：

https://github.com/Skyyyy0920/Principles-of-Compiler-Design

参考文献

[1] SysY2022 语言定义-V1

[2] SysY2022 运行时库-V1