南开大学
Nankai University

南　　开　　大　　学

编译系统原理实验报告

完成编译器构造

王思宇

年级：2021 级

专业：信息安全

蒋薇

年级：2021 级

专业：计算机科学与技术

2024 年 1 月 16 日

摘要

本学期通过理论课加实验课，根据助教提供的代码框架，实现构造了一个简单的编译器，主要包括：词法分析、语法分析、类型检查、中间代码生成、目标代码生成。我们构造的编译器除了满足基础要求之外，还实现了一些进阶要求：实现了数组的翻译、浮点类型的翻译、实现了 break、continue 语句的翻译。

关键字：词法分析、语法分析、类型检查、中间代码、目标代码、编译器

目录

一、 实验过程 **1**

（一） 词法分析 . . . 1

1. 实现符号表 . . . 1
2. 整形常量的词法分析 . . . 4
3. 浮点型常量的词法分析 . . . 5
4. 完成单行注释和多行注释的词法分析 . . . 5
5. 完成其他终结符的词法分析 . . . 6
6. 输出信息 . . . 9
7. 实验效果 . . . 10

（二） 语法分析 . . . 10

1. 类型系统 . . . 11
2. 符号表 . . . 13
3. 抽象语法树 . . . 14
4. 语法分析与语法树的创建 . . . 20
5. 实验效果 . . . 25

（三） 类型检查 . . . 26

1. 常量、变量未声明或重复声明 . . . 26
2. 类型转换 . . . 27
3. 数值运算表达式 . . . 28
4. 函数声明、参数 . . . 30
5. 循环检查 . . . 32
6. 数组相关初始化 . . . 32
7. 实验效果 . . . 33

（四） 中间代码生成 . . . 33

1. 关系表达式的翻译 . . . 33
2. 算数表达式的翻译 . . . 35
3. 赋值语句 . . . 36
4. 控制流语句的翻译 . . . 36
5. 数组、浮点数中间代码生成 . . . 40
6. 代码优化合并基本块 . . . 41
7. 实验效果 . . . 42

（五） 目标代码生成 . . . 43

1. 数据访存指令的翻译 . . . 43

2. 二元运算指令的翻译 48
3. 比较指令的翻译 50
4. 控制流指令的翻译 51
5. 函数调用的翻译 53
6. 变量及常量的打印 59
7. 相关 Output() 函数 61
8. 寄存器分配算法 62
9. 数组的翻译 64
10. 浮点类型的翻译 65
11. 实验效果 66
12. 代码链接 67

二、 实验总结 **67**
（一） 小组分工 67
（二） 实验总结 67

一、 实验过程

(一) 词法分析

实验要求：在 lab3 词法分析实验中，需要借助 Flex 完成这样一个程序，它的输入是一个 SysY 语言源程序，它的输出是每一个文法单元的类别、词素、行号、列号，以及必要的属性。即识别程序中所有单词，将其转化为单词流。

我们在助教给定的框架基础上通过修改 lexer.l 文件实现了词法分析器，将给定的 SysY 语言源程序转化为单词流，输出每个单词的类别、词素、行号、列号、属性（DECIMAL 的属性为数值，ID 的属性为符号表项指针）。

1. 实现符号表

对于标识符 (ID)，它的属性为符号表项（Symbol Table Entry），同名标识符在相同作用域可能指向相同的符号表项，也可能因为在不同作用域的重新声明而指向不同符号表项。我们希望词法程序可以对这些情况做区分，这需要设计符号表 (Symbol Table)，虽然目前符号表项还只是词素、作用域等简单内容，但符号表的数据结构，搜索算法，词素的保存，保留字的处理等问题都可以考虑了。

代码

实现符号表

```

  /* Your code here, if desired (lab3). */
    //链表形式组织符号表（此处符号表只记录ID）
    class Node {
        public:
        std::string name;
        Node* prev;
        Node* next;
        Node(const std::string& s) : name(s), prev(nullptr), next(nullptr) {}
};
    //链表
    class LinkedList {
public:
    LinkedList() : head(nullptr), tail(nullptr), count(0) {}
    ~LinkedList() {
        Node* current = head;
        while (current != nullptr) {
            Node* temp = current;
            current = current->next;
            delete temp;
        }
    }
    Node* add(const std::string& s) {
        Node* newNode = new Node(s);

        if (head == nullptr) {
            head = newNode;
```

```
            tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
        count++;
        return newNode;
    }
    Node* find(const std::string& s) {
        Node* current = head;
        while (current != nullptr) {
            if (current->name == s) {
                return current;
            }
            current = current->next;
        }
        return nullptr;
    }

private:
    Node* head;
    Node* tail;
    int count;
};
    LinkedList idlist;
    //自定义stack_element类（符号栈中元素）
    /*
    每个栈元素stack_element会持有一个指向前一元素的指针pre和一个map
    map是无序的，存储的是当前这个{}范围下所有的映射对：<标识符名，对应符号表表项的地址>
    */
    class stack_element {
    private:
        std::unordered_map<std::string, Node*> map;
        stack_element* prev;
    public:
        //初始化空的符号栈的首元素
        stack_element() {
            prev = nullptr;
        }
        //往符号栈里压元素，新压入元素的pre自然就是低他一层的元素
        stack_element(stack_element* back) {
            prev = back;
        }
        void add_into_map(std::string name, Node* id) {
            map[name] = id;
        }
```

```
75          stack_element* get_prev() {
76              return prev;
77          }
78          Node* get_identifier(std::string name) {
79              if (map.empty()) {
80                  return nullptr;
81              }
82              if (map.find(name) != map.end()) {
83                  return map[name];
84              }
85              else return nullptr;
86          }
87          ~stack_element() {
88              prev = nullptr;
89              std::unordered_map<std::string, Node*>().swap(map);
90              malloc_trim(0);
91          }
92      };
93      // symble_stack 类，识别到标识符将其存入符号栈中，最后再一次性输出
94      class symble_stack: public stack_element {
95      private:
96          // 栈的顶部
97          stack_element* top;
98      public:
99          // 初始化，之所以要初始化一个top表项，是为了存储全局标识符，这些标识
                符没有{}标识
100         symble_stack() {
101             stack_element* t = new stack_element();
102             top = t;
103         }
104         //栈中的每一个元素都代表一个{}范围内所有标识符相关的存储
105         // 添加元素
106         void push() {
107             stack_element* t = new stack_element(top);
108             top = t;
109         }
110         // 弹出元素
111         void pop() {
112             stack_element* p;
113             p = top;
114             top = top->get_prev();
115             p->~stack_element();
116         }
117         // 添加标识符
118         void add_map(std::string name, Node* id) {
119             top->add_into_map(name, id);
120         }
121         //查找符号栈中是否曾出现过同样的标识符
```

```
122         Node* lookup(std::string name) {
123         // 1. 未出现: 创建一个新的idlist_entry, 并添加到top指针指向的map中,
                然后返回idlist_entry的地址
124         // 2. 已出现: 返回对应的idlist_entry地址
125             Node* p_entry = nullptr;
126             //搜索: 从栈顶依次向下搜索,因为{}可能层层嵌套
127             stack_element* p;
128             p = top;
129             do {
130                 p_entry = p->get_identifier(name);
131                 if (p_entry) {
132                     //p_entry不为空, 意味着已经找到了
133                     return p_entry;
134                 }
135                 //p_entry为空, 意味着还没找到, 继续向下搜索
136                 p = p->get_prev();
137             } while (p);
138             //搜索不到, 于是这是一个新的标识符
139             p_entry = idlist.add((char*)name.c_str());
140             top->add_into_map(name, p_entry);
141             return p_entry;
142         }
143     };
144     symble_stack stack;
```

2. 整形常量的词法分析

需要定义八进制和十六进制的规则，将其保存为十进制输出：
代码

整形常量的词法分析

```
1  /*十进制*/
2  DECIMIAL ([1-9][0-9]*|0)
3  /*八进制*/
4  OCTAL 0[0-7]+
5  /*十六进制*/
6  HEX 0[xX][0-9a-fA-F]+
7  {OCTAL} {
8      int decimal = strtol(yytext + 1, NULL, 8);
9      dump_tokens("OCTAL\t%s\t%d\n", yytext, decimal);
10 }
11
12
13 {DECIMIAL} {
14     int decimal;
15     if (yytext[0] == '-') {
16         // 如果第一个字符是负号, 使用负数处理
17         decimal = atoi(yytext + 1); // 跳过负号
```

```
        decimal = -decimal; // 负数处理
        dump_tokens("DECIMAL\t%s\t%d\n", yytext, decimal);
    } else {
        // 否则，处理正整数
        decimal = atoi(yytext);
        dump_tokens("DECIMAL\t%s\t%d\n", yytext, decimal);
    }
}

{HEX} {
    int decimal = strtol(yytext + 2, NULL, 16);
    dump_tokens("HEX\t%s\t%d\n", yytext, decimal);
}

{FLOAT} {
    float floatValue = strtof(yytext, NULL);
    dump_tokens("FLOAT\t%s\t%f\n", yytext, floatValue);
}
```

3. 浮点型常量的词法分析

需要定义浮点型常量的规则，将其保存为 float 类型进行输出

代码

浮点型常量的词法分析

```
/*浮点型常量*/
FLOAT ([0-9]+[.][0-9]*|[.][0-9]+)([eE][-+]?[0-9]+)?
{FLOAT} {
    float floatValue = strtof(yytext, NULL);
    dump_tokens("FLOAT\t%s\t%f\n", yytext, floatValue);
}
```

4. 完成单行注释和多行注释的词法分析

代码

单行注释和多行注释的词法分析

```
/*回车换行*/
EOL (\r\n|\n|\r)
/*制表符 空格*/
WHITE [\t ]
/*  Your code here (lab3). */
%x BLOCKCOMMENT
 //块注释
BLOCKCOMMENTBEGIN "/*"
 //通配符+换行
BLOCKCOMMENTELEMENT .|\n
BLOCKCOMMENTEND "*/"
```

```
 //行注释: 以双斜杠开头, 后跟若干个非换行的字符
LINECOMMENT \/\/[^\n]*

{EOL} {
     yylineno++;
     update_position(yyleng);
}

{WHITE} {
    update_position(1);
}

{LINECOMMENT} {}    //处理注释, 使用定义好的独占状态BLOCKCOMMENT
{BLOCKCOMMENTBEGIN} {BEGIN BLOCKCOMMENT;}
<BLOCKCOMMENT>{BLOCKCOMMENTELEMENT} {}
<BLOCKCOMMENT>{BLOCKCOMMENTEND} {BEGIN INITIAL;} //使用宏BEGIN来切换状态, 初始状态默认为INITIAL
```

5. 完成其他终结符的词法分析

代码

其他终结符的词法分析

```

ID [[:alpha:]_][[:alpha:][:digit:]_]*

  %%
    /* rules section */
"int" {
   dump_tokens("INT\t%s\n", yytext);
   update_position(3); // 更新行和列号
}

"void" {

    dump_tokens("VOID\t%s\n", yytext);
    update_position(4);
}

"if" {
    dump_tokens("IF\t%s\n", yytext);
    update_position(2);
}

"else" {
    dump_tokens("ELSE\t%s\n", yytext);
    update_position(4);
}
```

```
26
27  "return" {
28      dump_tokens("RETURN\t%s\n", yytext);
29      update_position(6);
30  }
31
32  "while" {
33      dump_tokens("WHILE\t%s\n", yytext);
34      update_position(5);
35  }
36  "break" {
37      dump_tokens("BREAK\t%s\n", yytext);
38      update_position(5);
39  }
40  "continue" {
41      dump_tokens("CONTINUE\t%s\n", yytext);
42  }
43  "const" {
44      dump_tokens("CONST\t%s\n", yytext);
45      update_position(5);
46  }
47
48  "==" {
49      dump_tokens("EQUAL\t%s\n", yytext);
50      update_position(2);
51  }
52  "!=" {
53      dump_tokens("NOTEQUAL\t%s\n", yytext);
54      update_position(2);
55  }
56  "<=" {
57      dump_tokens("LESSEQUAL\t%s\n", yytext);
58      update_position(2);
59  }
60  ">=" {
61      dump_tokens("GREATEREQUAL\t%s\n", yytext);
62      update_position(2);
63  }
64
65
66  "=" {
67      dump_tokens("ASSIGN\t%s\n", yytext);
68      update_position(yyleng);
69      update_position(1);
70  }
71
72  "<" {
73      dump_tokens("LESS\t%s\n", yytext);
```

```
74      update_position(1);
75  }
76
77  ">" {
78      dump_tokens("GREAT\t%s\n", yytext);
79      update_position(1);
80  }
81
82  "+" {
83      dump_tokens("ADD\t%s\n", yytext);
84      update_position(1);
85  }
86  "-" {
87      dump_tokens("SUB\t%s\n", yytext);
88      update_position(1);
89  }
90  "*" {
91      dump_tokens("MUL\t%s\n", yytext);
92      update_position(1);
93  }
94  "/" {
95      dump_tokens("DIV\t%s\n", yytext);
96      update_position(1);
97  }
98  "%" {
99      dump_tokens("MOD\t%s\n", yytext);
100     update_position(1);
101 }
102 "&&" {
103     dump_tokens("AND\t%s\n", yytext);
104     update_position(2);
105 }
106 "||" {
107     dump_tokens("OR\t%s\n", yytext);
108     update_position(2);
109 }
110 "!" {
111     dump_tokens("NOT\t%s\n", yytext);
112     update_position(1);
113 }
114
115 ";" {
116     dump_tokens("SEMICOLON\t%s\n", yytext);
117     update_position(1);
118 }
119
120 "(" {
121     dump_tokens("LPAREN\t%s\n", yytext);
```

```
122      update_position(1);
123  }
124
125  ")" {
126      dump_tokens("RPAREN\t%s\n", yytext);
127      update_position(1);
128  }
129
130  "{" {
131      dump_tokens("LBRACE\t%s\n", yytext);
132      update_position(1);
133  }
134
135  "}" {
136      dump_tokens("RBRACE\t%s\n", yytext);
137      update_position(1);
138  }
139
140  "[" {
141      dump_tokens("LBRACKET\t%s\n", yytext);
142      update_position(1);
143  }
144
145  "]" {
146      dump_tokens("RBRACKET\t%s\n", yytext);
147      update_position(1);
148  }
149
150  "," {
151      dump_tokens("COMMA\t%s\n", yytext);
152      update_position(1);
153  }
154  {ID} {
155      std::string str = yytext; //yytext = 词素
156      Node *p = stack.lookup(str);
157      print_id(p);
158      //dump_tokens("ID\t%s\n", yytext);
159      update_position(yyleng);
160  }
```

6. 输出信息

输出

```
1  inline void dump_tokens(const char* format, ...) {
2          va_list args;
3          va_start(args, format);
4          if (dump_type == TOKENS){
```

```
            fprintf(yyout, "Line %d, Column %d: ", yylineno, column_number);//输出行号和列号
            vfprintf(yyout, format, args);
        }

        va_end(args);
    }
    // 打印ID:单词, 词素, 行号, 符号表项的地址
    void print_id(Node* id_elem) {
    // 输出行号和列号
    fprintf(yyout, "Line %d, Column %d: ", yylineno, column_number);
    // 打印标识符的信息
    int num_spaces = 15 - strlen(yytext);  // Calculate the number of spaces needed for alignment
    fprintf(yyout, "ID\t%s", yytext);
    for (int i = 0; i < num_spaces; ++i)
        fprintf(yyout, " ");
    fprintf(yyout, "Node Address:\t\t%p\n", (void*)id_elem);
```

7. 实验效果

```
Line 2, Column 1: INT   int
Line 2, Column 5: ID    test_hexoct     Node Address:       0x55bc2d8e0360
Line 2, Column 16: LPAREN   (
Line 2, Column 17: RPAREN   )
Line 2, Column 19: LBRACE   {
Line 3, Column 5: INT   int
Line 3, Column 9: ID    a               Node Address:       0x55bc2d8e0450
Line 3, Column 10: COMMA    ,
Line 3, Column 12: ID   b               Node Address:       0x55bc2d8e04d0
Line 3, Column 13: SEMICOLON    ;
Line 4, Column 5: ID    a               Node Address:       0x55bc2d8e0450
Line 4, Column 7: ASSIGN    =
Line 4, Column 10: HEX  0xf0aC  61612
Line 4, Column 10: SEMICOLON    ;
Line 5, Column 5: ID    b               Node Address:       0x55bc2d8e04d0
Line 5, Column 7: ASSIGN    =
Line 5, Column 10: HEX  0XcBeF  52207
Line 5, Column 10: SEMICOLON    ;
Line 6, Column 5: RETURN    return
Line 6, Column 12: ID   a               Node Address:       0x55bc2d8e0450
Line 6, Column 14: ADD  +
Line 6, Column 16: ID   b               Node Address:       0x55bc2d8e04d0
Line 6, Column 18: ADD  +
Line 6, Column 20: OCTAL    075 61
```

(二) 语法分析

实验要求：完善预备工作 2 中所设计的 SysY 语言的上下文无关文法，借助 Yacc 工具实现语法分析器：

- 语法树数据结构的设计：结点类型的设计，不同类型的节点应保存的信息。
- 扩展上下文无关文法，设计翻译模式。
- 设计 Yacc 程序，实现能构造语法树的分析器。
- 以文本方式输出语法树结构，验证语法分析器实现的正确性。

1. 类型系统

我们实现了 int、void 、FLOAT、函数、数组、CONST、布尔这些类型。主要是在 Type.h/c 文件中

类型 Type.h

```
class Type
{
private:
    int kind;
protected:
    enum {INT,VOID,FLOAT,CONST_INT, CONST_FLOAT,BOOL,FUNC,INT_ARRAY,
        FLOAT_ARRAY, CONST_INT_ARRAY, CONST_FLOAT_ARRAY};
public:
    Type(){};
    Type(int kind) : kind(kind) {};
    virtual ~Type() {};
    virtual std::string toStr() = 0;
    bool isInt() const {return kind == INT;};
    bool isVoid() const {return kind == VOID;};
    bool isFloat() const {return kind == FLOAT;};
    bool isConstInt() const {return kind == CONST_INT;}
    bool isConstFloat() const {return kind == CONST_FLOAT;}
    bool isBool() const {return kind == BOOL;}
    bool isFunc() const {return kind == FUNC;}
    bool isIntArray() const {return kind == INT_ARRAY;}
    bool isFloatArray() const {return kind == FLOAT_ARRAY;}
    bool isConstIntArray() const {return kind == CONST_INT_ARRAY;}
    bool isConstFloatArray() const {return kind == CONST_FLOAT_ARRAY;}
    bool isArray() const {return kind == INT_ARRAY || kind == FLOAT_ARRAY ||
        kind == CONST_FLOAT_ARRAY || kind == CONST_INT_ARRAY;}

};
```

类型 Type.c

```
std::string IntType::toStr()
{
    return "int";
}
std::string FloatType::toStr()
{
    return "float";

}
std::string VoidType::toStr()
{
    return "void";
}
```

```
14 std::string BoolType::toStr()
15 {
16     return "bool";
17 }
18 std::string ConstIntType::toStr()
19 {
20     return "const␣int";
21 }
22
23 std::string ConstFloatType::toStr()
24 {
25     return "const␣float";
26 }
27
28 void FunctionType::setparamsType(std::vector<Type*> in)
29 {
30     paramsType = in;
31 }
32
33 std::string FunctionType::toStr()
34 {
35     std::ostringstream buffer;
36     buffer << returnType->toStr() << "(";
37    for(int i = 0;i < (int)paramsType.size();i++){
38         if(i!=0) buffer << ",␣";
39         buffer << paramsType[i]->toStr();
40     }
41     buffer << ')';
42     return buffer.str();
43 }
44
45 void IntArrayType::pushBackDimension(int dim)
46 {
47     dimensions.push_back(dim);
48 }
49
50 std::vector<int> IntArrayType::getDimensions()
51 {
52     return dimensions;
53 }
54
55 std::string IntArrayType::toStr()
56 {
57     return "int␣array";
58 }
59
60 void FloatArrayType::pushBackDimension(int dim)
61 {
```

```
62      dimensions.push_back(dim);
63  }
64
65  std::vector<int> FloatArrayType::getDimensions()
66  {
67      return dimensions;
68  }
69
70  std::string FloatArrayType::toStr()
71  {
72      return "float array";
73  }
74  void ConstIntArrayType::pushBackDimension(int dim)
75  {
76      dimensions.push_back(dim);
77  }
78
79  std::vector<int> ConstIntArrayType::getDimensions()
80  {
81      return dimensions;
82  }
83
84  std::string ConstIntArrayType::toStr()
85  {
86      return "const int array";
87  }
88
89  void ConstFloatArrayType::pushBackDimension(int dim)
90  {
91      dimensions.push_back(dim);
92  }
93
94  std::vector<int> ConstFloatArrayType::getDimensions()
95  {
96      return dimensions;
97  }
98
99  std::string ConstFloatArrayType::toStr()
100 {
101     return "const float array";
102 }
```


2. 符号表

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。在 SymbolTable.cpp 中实现符号表的查找函数。

SymbolTale.cpp

```
//实现符号表的查找函数
SymbolEntry* SymbolTable::lookup(std::string name)
{
    // Todo
    if(symbolTable.find(name)!=symbolTable.end()){
        return symbolTable[name];// 如果在当前符号表中找到了符号，则返回该符号的指针
    }else{// 如果在当前符号表中没有找到符号
        if(prev != nullptr){// 如果存在父符号表（前一个符号表），则递归在父符号表中查找
            return prev->lookup(name);
        }else{
            return nullptr;// 如果当前符号表是最顶层的符号表且在其中也找不到符号，则返回nullptr表示未找到
        }
    }
}
```

3. 抽象语法树

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。结点的类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。

我们根据在预备工作 2 中定义的 SysY 语言特性设计其他结点类型，如 while 语句、函数调用等

Ast.h

```
//二元表达式节点
class BinaryExpr : public ExprNode
{
private:
    int op;
    ExprNode *expr1, *expr2;
public:
    enum {ADD, SUB, AND, OR, LESS,MUL,DIV,MOD,LESSEQUAL,GREATER,GREATEREQUAL,EQUAL,NOTEQUAL};//增加了一些运算符
    BinaryExpr(SymbolEntry *se, int op, ExprNode*expr1, ExprNode*expr2) : ExprNode(se), op(op), expr1(expr1), expr2(expr2){};
    void output(int level);
    int getValue();
};
//一元表达式节点
class UnaryExpr : public ExprNode
```

```
{
private:
    int op;
    ExprNode *expr;
public:
    enum {ADD, SUB, NOT};//一元表达式运算符
    UnaryExpr(SymbolEntry *se, int op, ExprNode*expr) : ExprNode(se), op(op),
        expr(expr){};
    void output(int level);
};
//常量表达式节点
class Constant : public ExprNode
{
public:
    Constant(SymbolEntry *se) : ExprNode(se){};
    void output(int level);
};
//函数调用表达式节点
class FuncCallExp : public ExprNode
{
private:
    ExprNode *param;//参数

public:
    FuncCallExp(SymbolEntry *se, ExprNode *param = nullptr)
        : ExprNode(se), param(param){};
    void output(int level);
};

//语句节点
class StmtNode : public Node
{};
//表达式语句节点
class ExprStmtNode : public StmtNode//注意: 该类由ExprStmt与ArrayIndices共
    享, 二者的行为完全一致
{
private:
    std::vector<ExprNode*> exprList;
public:
    ExprStmtNode(){};
    void addNext(ExprNode* next);//增加节点
    void output(int level);
};
//标识符节点
class Id : public ExprNode
{
private:
    ExprStmtNode* indices;//标识符相关联的索引表达式。这通常在处理数组时会用
```

```
        到，例如 a[0] 中的 [0] 部分。
public:
    Id(SymbolEntry *se) : ExprNode(se), indices(nullptr){};
    SymbolEntry* getSymbolEntry() {return symbolEntry;}
    bool isArray();      //必须配合indices!=nullptr使用（a[]的情况）
    void addIndices(ExprStmtNode* idx) {indices = idx;}//将索引表达式与标识符
        关联起来。这可以在处理数组时使用，例如 a[0] 中的 [0] 部分。
    void output(int level);
};
//表达式语句
class ExprStmt : public StmtNode
{
private:
    ExprNode *expr;

public:
    ExprStmt(ExprNode *expr) : expr(expr){};
    void output(int level);
};
//复合语句
class CompoundStmt : public StmtNode
{
private:
    StmtNode *stmt;
public:
    CompoundStmt(StmtNode *stmt) : stmt(stmt) {};
    void output(int level);
};
//序列节点
class SeqNode : public StmtNode
{
private:
    std::vector<StmtNode*> stmtList;
public:
    SeqNode(){};
    void addNext(StmtNode* next);
    void output(int level);
};
//初始化值节点，用于表示初始化语句中的值。
class InitValNode : public StmtNode
{
private:
    bool isConst;
    ExprNode* leafNode; //可能为空，j即使是叶节点（考虑{}）
    std::vector<InitValNode*> innerList;//为空则为叶节点，这是唯一判断标准
public:
    InitValNode(bool isConst) :
        isConst(isConst), leafNode(nullptr){};
```

```
107     void addNext(InitValNode* next);
108     void setLeafNode(ExprNode* leaf);
109     bool isLeaf();
110     void output(int level);
111 };
112 //表示变量或常量的声明语句
113 class DefNode : public StmtNode
114 {
115 private:
116     bool isConst;
117     bool isArray;
118     Id* id;//表示声明的标识符
119     Node* initVal;//对于非数组，是ExprNode；对于数组，是InitValueNode。获取赋
        值给标识符的表达式,如果为空，代表只声明，不赋初值
120 public:
121     DefNode(Id* id, Node* initVal, bool isConst, bool isArray) :
122         isConst(isConst), isArray(isArray), id(id), initVal(initVal){};
123     Id* getId() {return id;}
124     void output(int level);
125 };
126 //声明语句节点
127 class DeclStmt : public StmtNode
128 {
129 private:
130     bool isConst;
131     std::vector<DefNode*> defList;
132 public:
133     DeclStmt(bool isConst) : isConst(isConst){};
134     void addNext(DefNode* next);
135     void output(int level);
136 };
137 //IF语句
138 class IfStmt : public StmtNode
139 {
140 private:
141     ExprNode *cond;
142     StmtNode *thenStmt;
143 public:
144     IfStmt(ExprNode *cond, StmtNode *thenStmt) : cond(cond), thenStmt(
        thenStmt){};
145     void output(int level);
146 };
147 //IfElse语句
148 class IfElseStmt : public StmtNode
149 {
150 private:
151     ExprNode *cond;
152     StmtNode *thenStmt;
```

```
153     StmtNode *elseStmt;
154 public:
155     IfElseStmt(ExprNode *cond, StmtNode *thenStmt, StmtNode *elseStmt) : cond
            (cond), thenStmt(thenStmt), elseStmt(elseStmt) {};
156     void output(int level);
157 };
158 //While语句
159 class WhileStmt:public StmtNode
160 {
161 private:
162     ExprNode *cond;
163     StmtNode *stmt;
164 public:
165     WhileStmt(ExprNode *cond,StmtNode *stmt):cond(cond),stmt(stmt){};
166     void output(int level);
167 };
168 //Break语句
169 class BreakStmt : public StmtNode
170 {
171 public:
172     BreakStmt(){};
173     void output(int level);
174 };
175 //Continue语句
176 class ContinueStmt : public StmtNode
177 {
178 public:
179     ContinueStmt(){};
180     void output(int level);
181 };
182 //Return语句
183 class ReturnStmt : public StmtNode
184 {
185 private:
186     ExprNode *retValue;
187 public:
188     ReturnStmt(ExprNode*retValue) : retValue(retValue) {};
189     void output(int level);
190 };
191 //空语句
192 class EmptyStmt : public StmtNode
193 {
194 public:
195     EmptyStmt(){};
196     void output(int level);
197 };
198 //赋值语句
199 class AssignStmt : public StmtNode
```

```
{
private:
    ExprNode *lval;//左值
    ExprNode *expr;//表达式
public:
    AssignStmt(ExprNode *lval, ExprNode *expr) : lval(lval), expr(expr) {};
    void output(int level);
};
//函数调用参数节点
class FuncCallParamsNode : public StmtNode
{
private:
    std::vector<ExprNode*> paramsList;//参数列表
public:
    FuncCallParamsNode(){};
    void addNext(ExprNode* next);
    void output(int level);
};
//函数调用节点
class FuncCallNode : public ExprNode
{
private:
    Id* funcId;
    FuncCallParamsNode* params;
public:
    FuncCallNode(SymbolEntry *se, Id* id, FuncCallParamsNode* params) :
        ExprNode(se), funcId(id), params(params){};
    void output(int level);
};
//函数声明参数节点
class FuncDefParamsNode : public StmtNode
{
private:
    std::vector<Id*> paramsList;
public:
    FuncDefParamsNode() {};
    void addNext(Id* next);
    std::vector<Type*> getParamsType();
    void output(int level);
};
//函数声明节点
class FunctionDef : public StmtNode
{
private:
    SymbolEntry *se;
    FuncDefParamsNode *params;
    StmtNode *stmt;
public:
```

```
247     FunctionDef(SymbolEntry *se, FuncDefParamsNode *params, StmtNode *stmt) :
            se(se), params(params), stmt(stmt){};
248     void output(int level);
249 };
```

4. 语法分析与语法树的创建

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：

由于代码较多，下面是简化版本的，只有定义，没有具体的实现代码。

Parser.y

```
1  %start Program
2  %token <strtype> ID
3  %token <itype> INTEGER
4  %token <ftype> FLOATYPE
5  %token IF ELSE WHILE RETURN BREAK CONTINUE
6  %token INT VOID FLOAT
7  %token LPAREN RPAREN LBRACE RBRACE SEMICOLON LBRACKET RBRACKET COMMA
8  %token ADD SUB MUL DIV MOD OR AND NOT LESS LESSEQUAL GREATER GREATEREQUAL
       ASSIGN EQUAL NOTEQUAL
9  %token CONST
10
11 %type <stmttype> Stmts Stmt AssignStmt ExpStmt BlockStmt IfStmt WhileStmt
       BreakStmt ContinueStmt ReturnStmt
12 %type <stmttype> DeclStmt ConstDefList ConstDef ConstInitVal VarDefList
       VarDef VarInitVal FuncDef FuncParams FuncParam FuncRParams
13 %type <stmttype> ArrConstIndices ArrValIndices ConstInitValList
       VarInitValList
14 %type <exprtype> Exp ConstExp AddExp MulExp UnaryExp PrimaryExp LVal Cond
       LOrExp LAndExp EqExp RelExp
15 %type <type> Type
16
17 %precedence THEN
18 %precedence ELSE    //让else的优先级更高，即else优先与最近的if匹配
19 %%
20
21 //------------------------------编译单元------------------------------
22 Program
23     : Stmts
24     ;
25 //------------------------------类型------------------------------
26 Type
27     : INT
28     | VOID
29     | FLOAT
```

```
30      ;
31  //————————————————————————————语句————————————————————————————
32  Stmts
33      : Stmts Stmt
34      |Stmt
35      ;
36  Stmt
37      : AssignStmt {$$=$1;}
38      | BlockStmt {$$=$1;}
39      | IfStmt {$$=$1;}
40      | WhileStmt {$$=$1;}
41      | BreakStmt {$$=$1;}
42      | ContinueStmt {$$=$1;}
43      | ReturnStmt {$$=$1;}
44      | DeclStmt {$$=$1;}
45      | FuncDef {$$=$1;}
46      | ExpStmt SEMICOLON{$$=$1;}
47      | SEMICOLON {$$ = new EmptyStmt();}
48      ;
49
50  //左值
51  LVal
52      : ID
53      |ID ArrValIndices //数组左值
54      ;
55  // 赋值语句
56  AssignStmt
57      :
58      LVal ASSIGN Exp SEMICOLON {}
59      ;
60  // 表达式语句
61  ExpStmt
62      :    ExpStmt COMMA Exp
63      |    Exp
64      ;
65  //语句块
66  BlockStmt
67      :    LBRACE
68           |Stmts RBRACE
69           |LBRACE RBRACE
70      ;
71  //if语句
72  IfStmt
73      : IF LPAREN Cond RPAREN Stmt %prec THEN
74      | IF LPAREN Cond RPAREN Stmt ELSE Stmt
75      ;
76  //While语句
77  WhileStmt
```

```
78      : WHILE LPAREN Cond RPAREN Stmt
79      ;
80  BreakStmt    //Break语句
81      : BREAK SEMICOLON
82      ;
83  ContinueStmt    //Continue语句
84      : CONTINUE SEMICOLON
85      ;
86  ReturnStmt     // 返回语句
87      :
88      RETURN Exp SEMICOLON
89      | RETURN SEMICOLON
90      ;
91  Exp    // 算数表达式
92      :
93      AddExp {$$ = $1;}
94      ;
95  // 常量表达式
96  ConstExp
97      :    AddExp
98      ;
99  AddExp   // 加法级表达式
100     : MulExp {$$ = $1;}
101     | AddExp ADD MulExp
102     | AddExp SUB MulExp
103     ;
104 MulExp   // 乘法级表达式
105     : UnaryExp {$$ = $1;}
106     | MulExp MUL UnaryExp
107     | MulExp DIV UnaryExp
108     | MulExp MOD UnaryExp
109     ;
110 UnaryExp   //一元表达式
111     :
112     PrimaryExp {$$ = $1;}
113     |ID LPAREN FuncRParams RPAREN
114     |ADD UnaryExp   //'+'号
115     |SUB UnaryExp   //'-'号
116     |NOT UnaryExp   //'!'号
117     ;
118 // 基本表达式   基本数字/ID
119 PrimaryExp
120     :    LVal
121     |    LPAREN Exp RPAREN
122     |    INTEGER
123     |    FLOATYPE
124     ;
125 // 函数参数列表
```

```
126 FuncRParams
127     :    FuncRParams COMMA Exp
128     |    Exp
129     |    %empty
130     ;
131 Cond    // 条件表达式: 关系+逻辑运算（补）
132     :
133     LOrExp {$$ = $1;}
134     ;
135 // 或运算表达式
136 LOrExp
137     : LAndExp {$$ = $1;}
138     | LOrExp OR LAndExp
139     ;
140 LAndExp
141     : EqExp {$$ = $1;}
142     | LAndExp AND EqExp
143     ;
144 EqExp
145     : RelExp {$$ = $1;}
146     | EqExp EQUAL RelExp
147     | EqExp NOTEQUAL RelExp
148     ;
149 // 关系表达式
150 RelExp
151     : AddExp {$$ = $1;}
152     | RelExp LESS AddExp
153     | RelExp LESSEQUAL AddExp
154     | RelExp GREATER AddExp
155     | RelExp GREATEREQUAL AddExp
156     ;
157 // 数组的常量下标表示
158 ArrConstIndices
159     :    ArrConstIndices LBRACKET ConstExp RBRACKET
160     |    LBRACKET ConstExp RBRACKET
161     ;
162 // 数组的变量下标表示
163 ArrValIndices
164     :    ArrValIndices LBRACKET Exp RBRACKET
165     |    LBRACKET Exp RBRACKET
166     ;
167
168 //————————————————————————————声明————————————————————————————
169 DeclStmt
170     :    CONST Type ConstDefList SEMICOLON //常量声明
171
172     |    Type VarDefList SEMICOLON //变量声明
173     ;
```

```
174 // 常量定义列表
175 ConstDefList
176     :   ConstDefList COMMA ConstDef
177     |   ConstDef
178     ;
179 // 常量定义
180 ConstDef
181     :   ID ASSIGN ConstExp
182     |   ID ArrConstIndices ASSIGN ConstInitVal//数组常量的定义
183     ;
184 // 常量初始化值
185 ConstInitVal
186     :   ConstExp //不能直接赋值，否则根本无法判断是list还是expr
187 
188     // todo 常量数组的初始化值
189     |   LBRACE ConstInitValList RBRACE
190     |   LBRACE RBRACE
191     ;
192 // 数组常量初始化列表
193 ConstInitValList
194     :   ConstInitValList COMMA ConstInitVal
195     |   ConstInitVal
196     ;
197 // 变量定义列表
198 VarDefList
199     :   VarDefList COMMA VarDef
200     |   VarDef
201     ;
202 // 变量定义
203 VarDef
204     :   ID
205     |   ID ASSIGN Exp
206     // todo 数组变量的定义
207     |   ID ArrConstIndices
208     |   ID ArrConstIndices ASSIGN VarInitVal
209     ;
210 // 变量初始化值
211 VarInitVal
212     :   Exp
213     // todo 数组变量的初始化值
214     |   LBRACE VarInitValList RBRACE
215     |   LBRACE RBRACE
216     ;
217 // 数组变量初始化列表
218 VarInitValList
219     :   VarInitValList COMMA VarInitVal
220     |   VarInitVal
221     ;
```

```
// 函数定义
FuncDef
    :   Type ID
        |LPAREN FuncParams
        |RPAREN BlockStmt
    ;
// 函数参数列表
FuncParams
    :   FuncParams COMMA FuncParam
    |   FuncParam
    |   %empty
    ;
// 函数参数
FuncParam
    :   Type ID
    //数组函数参数
    |   Type ID LBRACKET RBRACKET ArrConstIndices
    |   Type ID LBRACKET RBRACKET
    ;
```

5. 实验效果

```
program
    Sequence
        FunctionDefine function name: deepWhileBr, type: int(int, int)
            FuncDefParamsNode
                Id  name: a scope: 1    type: int
                Id  name: b scope: 1    type: int
            CompoundStmt
                Sequence
                    DeclStmt
                        DefNode isConst:false   isArray:false
                            Id  name: c scope: 2    type: int
                            null
                    AssignStmt
                        Id  name: c scope: 2    type: int
                        BinaryExpr  op: add
                            Id  name: a scope: 1    type: int
                            Id  name: b scope: 1    type: int
                    WhileStmt
                        BinaryExpr  op: less
                            Id  name: c scope: 2    type: int
                            IntegerLiteral  value: 75   type: int
                        CompoundStmt
                            Sequence
                                DeclStmt
                                    DefNode isConst:false   isArray:false
                                        Id  name: d scope: 3    type: int
                                        null
                                AssignStmt
                                    Id  name: d scope: 3    type: int
                                    IntegerLiteral  value: 42   type: int
                                IfStmt
                                    BinaryExpr  op: less
                                        Id  name: c scope: 2    type: int
                                        IntegerLiteral  value: 100  type: int
                                    CompoundStmt
                                        Sequence
                                            AssignStmt
```

(三) 类型检查

实验要求：在语法分析实验的基础上，遍历语法树，进行简单的类型检查，对于语法错误的情况简单打印出提示信息。

基础要求：
- 检查未声明变量，及在同一作用域下重复声明的变量；
- 条件判断表达式：int 至 bool 隐式类型转换;
- 数值运算表达式：运算数类型是否正确 (如，返回值为 void 的函数调用结果是否参与了其他表达式的计算)
- 检查未声明函数，及函数形参是否与实参类型及数目匹配；
- 检查 return 语句操作数和函数声明的返回值类型是否匹配；
- 对 break、continue 语句进行静态检查，判断是否仅出现在 while 语句中。

1. 常量、变量未声明或重复声明

可以直接在构造语法分析树时进行检查 Parser.y 文件中实现：检查未声明变量，及在同一作用域下重复声明的变量或常量；以变量为例

未声明:

Parser.y

```
LVal
    :   ID {
            SymbolEntry *se;
            se = identifiers->lookup($1);
            if(se == nullptr)
            {
                fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1
                    );
                delete [](char*)$1;
                assert(se != nullptr);
            }
            $$ = new Id(se);
            delete []$1;
        }
    |   ID ArrValIndices {
            SymbolEntry *se;
            se = identifiers->lookup($1);
            if(se == nullptr)
            {
                fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1
                    );
                delete [](char*)$1;
                assert(se != nullptr);
            }
        }
    ;
```

重复声明:

Parser.y

```
// 变量定义
VarDef
    :   ID {
            if(identifiers->isRedefine($1)) {// 首先判断是否重定义
                fprintf(stderr, "identifier %s redefine\n", $1);
                exit(EXIT_FAILURE);
            }
        }
    |   ID ASSIGN Exp {
            if(identifiers->isRedefine($1)) {
                fprintf(stderr, "identifier %s redefine\n", $1);
                exit(EXIT_FAILURE);
            }
        }
    |   ID ArrConstIndices { // 数组变量的定义
            if(identifiers->isRedefine($1)) {
                fprintf(stderr, "identifier %s redefine\n", $1);
                exit(EXIT_FAILURE);
            }
        }
    |   ID ArrConstIndices ASSIGN VarInitVal{
            if(identifiers->isRedefine($1)) {
                fprintf(stderr, "identifier %s redefine\n", $1);
                exit(EXIT_FAILURE);
            }
        }
    ;
```

2. 类型转换

主要有两种类型：对于与和或运算，其两个运算数必须为 bool 类型，如果此时传入一个 int 类型的值，则需要进行类型转换。在 if 的条件判断时，经常会传入一个 int 类型的值作为不为 0 的判断标准，在类型检查时，当发现需要一个 bool 类型的值接受了一个 int 或者 float 类型的值，增加了一个该 int 值和 0 不相等的比较节点，实现了从 int 到 bool 类型的转化。

Ast.cpp

```
// 操作数类型不为 bool，或者se是一个常量bool
    // 则说明此时的情况为 a || 1 或者 a && a + b
    // 增加一个和1的EQ判断
    if(op == AND || op == OR) {
        if(!expr1->getSymPtr()->getType()->isBool() || expr1->getSymPtr()->
            isConstant()) {
            Constant* zeroNode = new Constant(new ConstantSymbolEntry(
                TypeSystem::constIntType, 0));
            TemporarySymbolEntry* tmpSe = new TemporarySymbolEntry(TypeSystem
                ::boolType, SymbolTable::getLabel());
```

```
            BinaryExpr* newCond = new BinaryExpr(tmpSe, BinaryExpr::NEQ,
                zeroNode, expr1);
            expr1 = newCond;
        }
        if(!expr2->getSymPtr()->getType()->isBool() || expr2->getSymPtr()->
            isConstant()) {
            Constant* zeroNode = new Constant(new ConstantSymbolEntry(
                TypeSystem::constIntType, 0));
            TemporarySymbolEntry* tmpSe = new TemporarySymbolEntry(TypeSystem
                ::boolType, SymbolTable::getLabel());
            BinaryExpr* newCond = new BinaryExpr(tmpSe, BinaryExpr::NEQ,
                zeroNode, expr2);
            expr2 = newCond;
        }
    }
```

Ast.cpp

```
void IfStmt::typeCheck(Node** parentToChild)
{
    cond->typeCheck((Node**)&(this->cond));
    // 如果cond中的se的类型不为 bool，或者se是一个常量bool
    // 则说明此时的情况为 if(a) 或者 if(1) 或者 if(a+1)
    // 增加一个和1的EQ判断
    if(!cond->getSymPtr()->getType()->isBool() || cond->getSymPtr()->
        isConstant()) {
        Constant* zeroNode = new Constant(new ConstantSymbolEntry(TypeSystem
            ::constIntType, 0));
        TemporarySymbolEntry* tmpSe = new TemporarySymbolEntry(TypeSystem::
            boolType, SymbolTable::getLabel());
        BinaryExpr* newCond = new BinaryExpr(tmpSe, BinaryExpr::NEQ, zeroNode
            , cond);
        cond = newCond;
    }
}
```

3. 数值运算表达式

以二元运算节点为例：首先对两个运算数执行类型检查，然后根据类型检查的结果，判断两个运算数是否为常量，如果均为常量，则根据运算符对该表达式节点进行计算，并且生成新的节点保存计算结果。这里实现了 int 和 float 两种类型。

Ast.cpp

```
void BinaryExpr::typeCheck(Node** parentToChild)
{
    expr1->typeCheck((Node**)&(this->expr1));
    expr2->typeCheck((Node**)&(this->expr2));
    //检查是否void函数返回值参与运算
```

```
6       Type* realTypeLeft = expr1->getType()->isFunc() ?
7           ((FunctionType*)expr1->getType())->getRetType() :
8           expr1->getType();
9       if(!realTypeLeft->calculatable()){
10          fprintf(stderr, "type %s is not calculatable!\n", expr1->getType()->
                toStr().c_str());
11          exit(EXIT_FAILURE);
12      }
13      Type* realTypeRight = expr2->getType()->isFunc() ?
14          ((FunctionType*)expr2->getType())->getRetType() :
15          expr2->getType();
16      if(!realTypeRight->calculatable()){
17          fprintf(stderr, "type %s is not calculatable!\n", expr2->getType()->
                toStr().c_str());
18          exit(EXIT_FAILURE);
19      }
20      //左右子树均为常数，计算常量值，替换节点
21      if(realTypeLeft->isConst() && realTypeRight->isConst()){
22          SymbolEntry *se;
23          // 如果该节点结果的目标类型为bool
24          if(this->getType()->isBool()) {
25              bool val = 0;
26              float leftValue = expr1->getSymPtr()->isConstant() ?
27                  ((ConstantSymbolEntry*)(expr1->getSymPtr()))->getValue() :
28                  ((IdentifierSymbolEntry*)(expr1->getSymPtr()))->value;
29              float rightValue = expr2->getSymPtr()->isConstant() ?
30                  ((ConstantSymbolEntry*)(expr2->getSymPtr()))->getValue() :
31                  ((IdentifierSymbolEntry*)(expr2->getSymPtr()))->value;
32              switch(op)
33              {
34              case AND:
35                  val = leftValue && rightValue;
36              break;
37              case OR:
38                  val = leftValue || rightValue;
39              break;
40              case LESS:
41                  val = leftValue < rightValue;
42              break;
43              case LESSEQ:
44                  val = leftValue <= rightValue;
45              break;
46              case GREAT:
47                  val = leftValue > rightValue;
48              break;
49              case GREATEQ:
50                  val = leftValue >= rightValue;
51              break;
```

```
            case EQ:
                val = leftValue == rightValue;
            break;
            case NEQ:
                val = leftValue != rightValue;
            break;
            }
            se = new ConstantSymbolEntry(TypeSystem::constBoolType, val);
        }
        Constant* newNode = new Constant(se);
        *parentToChild = newNode;
    }
}
```

4. 函数声明、参数

检查未声明的函数，函数的形参与实参的类型及数目是否匹配。return 语句操作数和函数声明的返回值类型是否匹配

Ast.cpp

```
//函数形参是否与实参类型及数目匹配
void FuncCallNode::typeCheck(Node** parentToChild)
{
    std::vector<Type*> funcParamsType = (dynamic_cast<FunctionType*>(this->
        funcId->getSymPtr()->getType()))->getParamsType();
    // 首先对于无参的进行检查
    if(this->params==nullptr && funcParamsType.size() != 0){
        fprintf(stderr, "function␣%s␣call␣params␣number␣is␣not␣consistent\n",
            this->funcId->getSymPtr()->toStr().c_str());
        exit(EXIT_FAILURE);
    }
    else if(this->params==nullptr) {
        return;
    }
    // 先对FuncCallParamsNode进行类型检查，主要是完成常量计算
    this->params->typeCheck(nullptr);
    std::vector<ExprNode*> funcCallParams = this->params->getParamsList();
    // 如果数量不一致直接报错
    if(funcCallParams.size() != funcParamsType.size()) {
        fprintf(stderr, "function␣%s␣call␣params␣number␣is␣not␣consistent\n",
            this->funcId->getSymPtr()->toStr().c_str());
        exit(EXIT_FAILURE);
    }
    // 然后进行类型匹配
    // 依次匹配类型
    for(std::vector<Operand*>::size_type i = 0; i < funcParamsType.size(); i
        ++){
        Type* needType = funcParamsType[i];
```

```
25          Type* giveType = funcCallParams[i]->getSymPtr()->getType();
26          // 暂时不考虑类型转化的问题 所有的类型转化均到IR生成再做
27          // 除了void类型都可以进行转化
28          if((!needType->calculatable() && giveType->calculatable()) ||(
                needType->calculatable() && !giveType->calculatable())){
29              fprintf(stderr, "function %s call params type is not consistent\n
                    ",this->funcId->getSymPtr()->toStr().c_str());
30              exit(EXIT_FAILURE);
31          }
32          // 检查数组是否匹配
33          if((!needType->isArray() && giveType->isArray()) || (needType->
                isArray() && !giveType->isArray())){
34              fprintf(stderr, "function %s call params type is not consistent\n
                    ",this->funcId->getSymPtr()->toStr().c_str());
35              exit(EXIT_FAILURE);
36          }
37          //TODO: 检查数组维度是否匹配
38          if(needType->isArray() && giveType->isArray()){
39
40          }
41      }
42  }
```

Ast.cpp

```
1   void ReturnStmt::typeCheck(Node** parentToChild)
2   {
3       if(returnType == nullptr){//返回语句在函数之外
4           fprintf(stderr, "return statement outside functions\n");
5           exit(EXIT_FAILURE);
6       }
7       else if(returnType->isVoid() && retValue!=nullptr){//void 函数中尝试返回
            值
8           fprintf(stderr, "value returned in a void() function\n");
9           exit(EXIT_FAILURE);
10      }
11      else if(!returnType->isVoid() && retValue==nullptr){//在一个非 void 函数
            中没有返回值
12          fprintf(stderr, "expected a %s type to return, but returned nothing\n
                ", returnType->toStr().c_str());
13          exit(EXIT_FAILURE);
14      }
15      if(!returnType->isVoid()){//函数的返回类型不是 void
16          retValue->typeCheck((Node**)&(retValue));
17      }
18      this->retType = returnType;
19      funcReturned = true;//设置一个标志 funcReturned, 表示函数已经执行了返回语
            句
20  }
```

5. 循环检查

对 break、continue 语句进行静态检查，判断是否仅出现在 while 语句中。

Ast.cpp

```
void ContinueStmt::typeCheck(Node** parentToChild)
{
    if(!inIteration){
        fprintf(stderr, "continue statement outside iterations\n");
        exit(EXIT_FAILURE);
    }
}
void BreakStmt::typeCheck(Node** parentToChild)
{
    if(!inIteration){
        fprintf(stderr, "break statement outside iterations\n");
        exit(EXIT_FAILURE);
    }
}
```

6. 数组相关初始化

对数组维度进行相应的类型检查

Ast.cpp

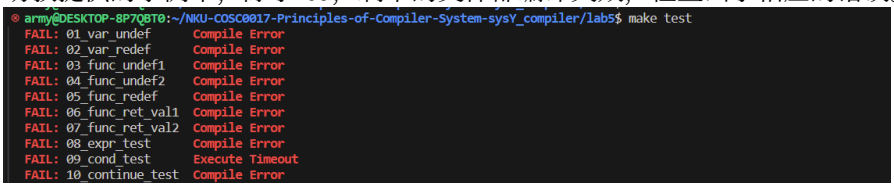
```
void Id::typeCheck(Node** parentToChild)
{
    // 如果是一个普通变量就什么也不做.如果是数组 要看看维度信息有没有初始化
    // 由于在语法解析阶段已经判断了标识符先定义再使用.所以如果维度信息还未初始化则说明当前是数组定义阶段
    if(isArray() && indices!=nullptr){
        indices->typeCheck(nullptr);
        // 检查indices下的exprList(私有域)中的每个exprNode的类型, 若不为自然数则报错
        if(((IdentifierSymbolEntry*)getSymPtr())->arrayDimension.empty()){
            indices->initDimInSymTable((IdentifierSymbolEntry*)getSymPtr());
        }
    }
}

void ExprStmtNode::initDimInSymTable(IdentifierSymbolEntry* se)
{
    for(auto expr :exprList){
        if(!(expr->getSymPtr()->isConstant() || expr->getType()->isConst())){
            fprintf(stderr, "array dimensions must be constant! %d %d\n",
                expr->getSymPtr()->isConstant(), expr->getType()->isConst());
```

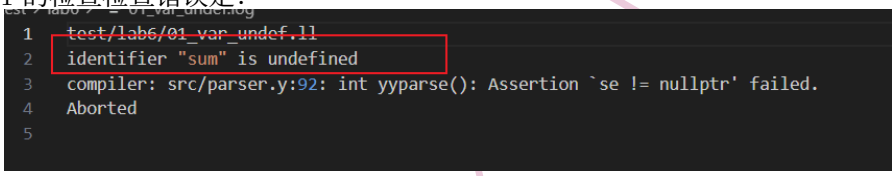
```
19              fprintf(stderr, "%d␣%d\n", (int)((ConstantSymbolEntry*)(expr->
                    getSymPtr()))->getValue(), (int)((IdentifierSymbolEntry*)(
                    expr->getSymPtr()))->value);
20              exit(EXIT_FAILURE);
21          }
22      }
23  }
```

7. 实验效果

助教提供的示例中，除了 09，剩下的文件都编译失败，检查出了相应的错误。



如 01 的检查检查错误是：



(四) 中间代码生成

中间代码生成旨在前继词法分析、语法分析实验的基础上，将 SysY 源代码翻译为中间代码。中间代码生成主要包含对数据流和控制流两种类型语句的翻译，数据流包括表达式运算、变量声明与赋值等，控制流包括 if、while、break、continue 等语句。

可以用于多种中间表示，包括抽象语法树和三地址代码。之所以命名为三地址代码，主要是因为这些指令的一般形式 x = y op z 具有三个地址：两个运算分量 y 和 z，以及结果变量 x。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。我们将生成 llvm IR。

1. 关系表达式的翻译

关系表达式

```
1  void BinaryExpr::genCode()
2  {
3      BasicBlock *bb = builder->getInsertBB();
4      Function *func = bb->getParent();
5      Type* maxType = TypeSystem::getMaxType(expr1->getSymPtr()->getType(),
           expr2->getSymPtr()->getType());
```

```
    if (op == AND)
    {
        BasicBlock *trueBB = new BasicBlock(func);  // if the result of lhs is true, jump to the trueBB.
        genBr = 1;
        expr1->genCode();
        backPatch(expr1->trueList(), trueBB);
        builder->setInsertBB(trueBB);               // set the insert point to the trueBB so that intructions generated by expr2 will be inserted into it.
        expr2->genCode();
        true_list = expr2->trueList();
        false_list = merge(expr1->falseList(), expr2->falseList());
    }
    else if(op == OR)
    {
        BasicBlock *falseBB = new BasicBlock(func);
        genBr = 1;
        expr1->genCode();
        backPatch(expr1->falseList(), falseBB);
        builder->setInsertBB(falseBB);
        expr2->genCode();
        true_list = merge(expr1->trueList(), expr2->trueList());
        false_list = expr2->falseList();
    }
    else if(op >= LESS && op <= NEQ)
    {
        genBr--;
        expr1->genCode();
        expr2->genCode();
        genBr++;
        Operand *src1 = typeCast(maxType, expr1->getOperand());
        Operand *src2 = typeCast(maxType, expr2->getOperand());
        int opcode;
        switch (op)
        {
        case LESS:
            opcode = CmpInstruction::L;
            break;
        case LESSEQ:
            opcode = CmpInstruction::LE;
            break;
        case GREAT:
            opcode = CmpInstruction::G;
            break;
        case GREATEQ:
            opcode = CmpInstruction::GE;
            break;
```

```
        case EQ:
            opcode = CmpInstruction::E;
            break;
        case NEQ:
            opcode = CmpInstruction::NE;
            break;
        }
        if(maxType->isFloat()) {
            new FCmpInstruction(opcode, dst, src1, src2, bb);
        }
        else {
            new CmpInstruction(opcode, dst, src1, src2, bb);
        }
        if(genBr > 0){
            // 跳转目标block
            BasicBlock* trueBlock, *falseBlock, *mergeBlock;
            trueBlock = new BasicBlock(func);
            falseBlock = new BasicBlock(func);
            mergeBlock = new BasicBlock(func);
            true_list.push_back(new CondBrInstruction(trueBlock, falseBlock,
                dst, bb));
            false_list.push_back(new UncondBrInstruction(mergeBlock,
                falseBlock));
        }
    }
```

2. 算数表达式的翻译

算术表达式

```
else if(op >= ADD && op <= MOD)
    {
        expr1->genCode();
        expr2->genCode();
        Operand *src1 = typeCast(maxType, expr1->getOperand());
        Operand *src2 = typeCast(maxType, expr2->getOperand());
        int opcode;
        switch (op)
        {
        case ADD:
            opcode = BinaryInstruction::ADD;
            break;
        case SUB:
            opcode = BinaryInstruction::SUB;
            break;
        case MUL:
            opcode = BinaryInstruction::MUL;
            break;
```

```
        case DIV:
            opcode = BinaryInstruction::DIV;
            break;
        case MOD:
            opcode = BinaryInstruction::MOD;
            break;
        }
        if(maxType->isFloat()) {
            new FBinaryInstruction(opcode, dst, src1, src2, bb);
        }
        else {
            new BinaryInstruction(opcode, dst, src1, src2, bb);
        }
```

builder 是 IRBuilder 类对象，用于传递继承属性，如新生成的指令要插入的基本块，辅助我们进行中间代码生成。在上面的例子中，我们首先通过 builder 得到后续生成的指令要插入的基本块 bb，然后生成子表达式的中间代码，通过 getOperand 函数得到子表达式的目的操作数，设置指令的操作码，最后生成相应的二元运算指令并插入到基本块 bb 中。

3. 赋值语句

赋值语句

```

void AssignStmt::genCode()
{
    BasicBlock *bb = builder->getInsertBB();
    expr->genCode();
    Operand *addr = dynamic_cast<IdentifierSymbolEntry*>(lval->getSymPtr())->
        getAddr();
    Operand *src = typeCast(dynamic_cast<PointerType*>(addr->getType())->
        getValueType(), expr->getOperand());
    new StoreInstruction(addr, src, bb);
```

4. 控制流语句的翻译

if-else 语句

if-else 控制流

```
  void IfElseStmt::genCode()
{
    Function *func;
    BasicBlock *then_bb, *else_bb, *end_bb;

    func = builder->getInsertBB()->getParent();
    then_bb = new BasicBlock(func);
    else_bb = new BasicBlock(func);
    end_bb = new BasicBlock(func);

```

```
    genBr = 1;
    cond->genCode();
    backPatch(cond->trueList(), then_bb);
    backPatch(cond->falseList(), else_bb);

    // 先处理then分支
    builder->setInsertBB(then_bb);
    thenStmt->genCode();
    then_bb = builder->getInsertBB();
    new UncondBrInstruction(end_bb, then_bb);

    // 再处理else分支
    builder->setInsertBB(else_bb);
    elseStmt->genCode();
    else_bb = builder->getInsertBB();
    new UncondBrInstruction(end_bb, else_bb);

    builder->setInsertBB(end_bb);
}
```

通过回填技术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基本块的列表，true_list 中的基本块为无条件跳转指令跳转到的目标基本块与条件跳转指令条件为真时跳转到的目标基本块，false_list 中的基本块为条件跳转指令条件为假时跳转到的目标基本块，这些目标基本块在翻译当前结点时尚不能确定，等到翻译其祖先结点能确定这些目标基本块时进行回填。

while 语句

while 语句

```
void WhileStmt::genCode()
{
    // 将当前的whileStmt压栈
    whileStack.push(this);
    Function* func = builder->getInsertBB()->getParent();
    BasicBlock* stmt_bb, *cond_bb, *end_bb, *bb = builder->getInsertBB();
    stmt_bb = new BasicBlock(func);
    cond_bb = new BasicBlock(func);
    end_bb = new BasicBlock(func);

    this->condBlock = cond_bb;
    this->endBlock = end_bb;

    // 先从当前的bb跳转到cond_bb进行条件判断
    new UncondBrInstruction(cond_bb, bb);

    // 调整插入点到cond_bb，对条件判断部分生成中间代码
    builder->setInsertBB(cond_bb);
    genBr = 1;
    cond->genCode();
```

```
    backPatch(cond->trueList(), stmt_bb);
    backPatch(cond->falseList(), end_bb);

    // 调整插入点到stmt_bb，对循环体部分生成中间代码
    builder->setInsertBB(stmt_bb);
    bodyStmt->genCode();
    // 循环体完成之后，增加一句无条件跳转到cond_bb
    stmt_bb = builder->getInsertBB();
    new UncondBrInstruction(cond_bb, stmt_bb);

    // 重新调整插入点到end_bb
    builder->setInsertBB(end_bb);


    // 将当前的whileStmt出栈
    whileStack.pop();
}
```

return 语句

return 语句

```
void ReturnStmt::genCode()
{
    BasicBlock *bb = builder->getInsertBB();
    if(retValue != nullptr) {
        retValue->genCode();
        Operand* operand = typeCast(this->retType, retValue->getOperand());
        new RetInstruction(operand, bb);
    }
    else {
        new RetInstruction(nullptr, bb);
    }
}
```

break 语句

break 语句

```
void BreakStmt::genCode()
{
    assert(whileStack.size()!=0);
    Function* func = builder->getInsertBB()->getParent();
    BasicBlock* bb = builder->getInsertBB();
    // 首先获取当前所在的while
    WhileStmt* whileStmt = whileStack.top();
    // 获取条件判断block
    BasicBlock* end_bb = whileStmt->getEndBlock();
    // 在当前基本块中生成一条跳转到条件判断的语句
    new UncondBrInstruction(end_bb, bb);
    // 声明一个新的基本块用来插入后续的指令
```

```
    BasicBlock* nextBlock = new BasicBlock(func);
    builder->setInsertBB(nextBlock);
}
```

continue 语句

continue 语句

```
void ContinueStmt::genCode()
{
    assert(whileStack.size()!=0);
    Function* func = builder->getInsertBB()->getParent();
    BasicBlock* bb = builder->getInsertBB();
    // 首先获取当前所在的while
    WhileStmt* whileStmt = whileStack.top();
    // 获取条件判断block
    BasicBlock* cond_bb = whileStmt->getCondBlock();
    // 在当前基本块中生成一条跳转到条件判断的语句
    new UncondBrInstruction(cond_bb, bb);
    // 声明一个新的基本块用来插入后续的指令
    BasicBlock* nextBlock = new BasicBlock(func);
    builder->setInsertBB(nextBlock);

}
```

函数

函数

```
void FuncCallNode::genCode()
{
    //找到对应function的符号表项
    IdentifierSymbolEntry* actualSE = dynamic_cast<IdentifierSymbolEntry*>(
        funcId->getSymPtr());
    if(actualSE->isLibFunc()){//若为库函数，则输出declare语句
        builder->getUnit()->insertDecl(actualSE);
    }
    //输出call语句
    //TODO: 内联函数
    BasicBlock *bb = builder->getInsertBB();
    //void 型函数不能返回
    if(params==nullptr){
        std::vector<Operand*> emptyList;
        new CallInstruction(dst, emptyList, dynamic_cast<
            IdentifierSymbolEntry*>(funcId->getSymPtr()), bb);
    }
    else{
        // 生成计算各个实参的中间代码
        params->genCode();
        // 完成实参形参之间的类型转换
```

```
        IdentifierSymbolEntry* funcSe = dynamic_cast<IdentifierSymbolEntry*>(
            funcId->getSymPtr());
        std::vector<Type*> paramsType = dynamic_cast<FunctionType*>(funcSe->
            getType())->getParamsType();
        std::vector<Operand*> passParams = params->getOperandList();
        std::vector<Operand*> realParams;
        for(int i = 0; i < passParams.size(); i++) {
            realParams.push_back(typeCast(paramsType[i], passParams[i]));
        }
        new CallInstruction(dst, realParams, dynamic_cast<
            IdentifierSymbolEntry*>(funcId->getSymPtr()), bb);
    }
}
```

5. 数组、浮点数中间代码生成

浮点数

```
Operand* Node::typeCast(Type* targetType, Operand* operand) {
    // 首先判断是否真的需要类型转化
    if(!TypeSystem::needCast(operand->getType(), targetType)) {
        return operand;
    }
    BasicBlock *bb = builder->getInsertBB();
    Function *func = bb->getParent();
    Operand* retOperand = new Operand(new TemporarySymbolEntry(targetType,
        SymbolTable::getLabel()));
    // 先实现bool扩展为int
    if(operand->getType()->isBool() && targetType->isInt()) {
        // 插入一条符号扩展指令
        new ZextInstruction(operand, retOperand, bb);
    }
    // 实现 int 到 float 的转换
    else if(operand->getType()->isInt() && targetType->isFloat()) {
        // 插入一条类型转化指令
        new IntFloatCastInstructionn(IntFloatCastInstructionn::I2F, operand,
            retOperand, bb);
    }
    // 实现 float 到 int 的转换
    else if(operand->getType()->isFloat() && targetType->isInt()) {
        // 插入一条类型转化指令
        new IntFloatCastInstructionn(IntFloatCastInstructionn::F2I, operand,
            retOperand, bb);
    }
    return retOperand;
}
```

数组

```
// 数组变量的定义
    |   ID ArrConstIndices {
            // 首先判断是否重定义
            if(identifiers->isRedefine($1)) {
                fprintf(stderr, "identifier %s redefine\n", $1);
                exit(EXIT_FAILURE);
            }
            Type* type;
            if(currentType->isInt()){
                type = new IntArrayType();
            }
            else{
                type = new FloatArrayType();
            }
            SymbolEntry *se;
            se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel())
                ;
            identifiers->install($1, se);
            Id* id = new Id(se);
            id->addIndices(dynamic_cast<ExprStmtNode*>($2));
            $$ = new DefNode(id, nullptr, false, true);//类型向上转换
        }
```

6. 代码优化合并基本块

通过检查块内是否存在条件跳转指令以及后继块的情况来确定是否可以进行合并。

基本快合并

```
// 查找可以合并的块
void BlockMerge::findBLocks(Function *func) {
    for (auto bb : func->getBlockList()) {
        // 1. 检查块内是否存在条件跳转指令, 后继块数目是否为1
        BasicBlock *block = bb;
        int succ_num = block->getNumOfSucc();
        bool hasConditionalJump = false;
        for (auto inst : block->getInstructionList()) {
            if (inst->isConditionalJump()) {
                hasConditionalJump = true;
                break;
            }
        }
        if (hasConditionalJump || succ_num > 1) {
            continue;
        }

        mergeList.clear();

```

```
        // 依据控制流持续向后合并，直至存在块不可合并
        while (true) {
            //  2. 检查后继块是否可以合并，包括后继块的前驱块数目等
            bool can_merge = false;
            BasicBlock* succ = block->getSuccessor(0); // 假设只有一个后继块
            int pred_num = succ->getNumOfPred();
            if (pred_num == 1) {
                can_merge = true;
            }

            if (can_merge) {
                mergeList.push_back(succ);
                block = succ;
            } else {
                break;
            }
        }
        //3. 合并基本块
        if (mergeList.size() > 0) {
            for (auto mergeBlock : mergeList) {
                // 将mergeBlock的指令移动到bb中
                bb->merge(mergeBlock);
                // 更新控制流图等
                func->updateControlFlowGraph();
            }
        }
    }
}
```

7. 实验效果

```
define i32 @main(){
B1:
  %t4 = alloca i32, align 4
  %t3 = alloca i32, align 4
  %t2 = alloca i32, align 4
  store i32 1, i32* %t2, align 4
  store i32 2, i32* %t3, align 4
  store i32 3, i32* %t4, align 4
  %t5 = load i32, i32* %t3, align 4
  %t6 = load i32, i32* %t4, align 4
  %t0 = add i32 %t5, %t6
  ret i32 %t0
}
```

图 1: Caption

（五） 目标代码生成

我们要将中间代码转化为目标代码。

AsmBuilder.h 是汇编代码构造辅助类，其主要作用就是在中间代码向目标代码进行自顶向下的转换过程中，记录当前正在翻译的函数、基本块，以便于函数、基本块及指令的插入。

MachineCode.h 是汇编代码构造相关类, –MachineUnit –MachineFunction –MachineBlock –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。–MachineOperand IMM 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。

LinvearScan.h 是线性扫描寄存器分配相关类，为虚拟寄存器分配物理寄存器。

LiveVariableAnalysis.h 是活跃变量分析相关类，用于寄存器分配过程。

对中间代码进行自顶向下的遍历，从而生成使用虚拟寄存器的目标代码。

1. 数据访存指令的翻译

storeInstruction

```
1  // 存储指令的生成
2  void StoreInstruction::genMachineCode(AsmBuilder *builder)
3  {
4      auto cur_block = builder->getBlock();   // 获取当前代码块
5      MachineInstruction *cur_inst = nullptr; // 指令
6      // 如果存储的是函数参数（参数传递至少前四个参数使用寄存器）
7      if (operands[1]->getEntry()->isVariable())
8      {

           // 检查操作数1是否为一个变量
9          auto id_se = dynamic_cast<IdentifierSymbolEntry *>(operands[1]->
               getEntry()); // 将操作数1的符号表条目转换为标识符类型。
10         if (id_se->isParam())
11         {

               // 检查该标识符是否是一个函数参数
12             int param_id = this->getParent()->getParent()->getParamId(
                   operands[1]); // 获取该参数在函数参数列表中的索引
13             if (param_id >= 4)
14             {

                   // 检查参数ID是否大于等于4，因为前四个参数通常通过寄存器传
                   递，而不是通过栈。
15                 int offset = 4 * (param_id - 4);
                                                          // 计算
```

```
                参数在栈上的偏移量。偏移量是相对于前四个参数在栈上的位置
                而言的，每个参数占据4个字节的空间。
16              dynamic_cast<TemporarySymbolEntry *>(operands[0]->getEntry())
                    ->setOffset(offset); // 将计算得到的偏移量设置给存储指令
                    中的目标操作数（通常是一个临时变量）。
17              return;
18          }
19      }
20  }
21
22  MachineOperand *src = nullptr;
23  // 如果源操作数是浮点数类型
24  if (operands[1]->getType()->isFloat())
25  {
26      src = genMachineOperand(operands[1], true);
27      // 如果源操作数是常数，需要先加载到寄存器
28      if (src->isImm())
29      {
30          auto tmp_dst = genMachineVReg(true);
31          auto internal_reg = genMachineVReg();
32          cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
33          cur_block->InsertInst(cur_inst);
34          internal_reg = new MachineOperand(*internal_reg);
35          cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
                tmp_dst, internal_reg);
36          cur_block->InsertInst(cur_inst);
37          src = new MachineOperand(*tmp_dst);
38      }
39  }
40  else
41  {
42      src = genMachineOperand(operands[1]);
43      // 如果源操作数是常数，需要先加载到寄存器
44      if (src->isImm())
45      {
46          auto internal_reg = genMachineVReg();
47          cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
48          cur_block->InsertInst(cur_inst);
49          src = new MachineOperand(*internal_reg);
50      }
51  }
52  // 如果存储的是全局变量
53  if (operands[0]->getEntry()->isVariable() && dynamic_cast<
        IdentifierSymbolEntry *>(operands[0]->getEntry())->isGlobal())
54  {
55      auto internal_reg1 = genMachineVReg();
56      auto internal_reg2 = new MachineOperand(*internal_reg1);
57      auto dst = genMachineOperand(operands[0]);
```

```
58          if (operands[1]->getType()->isFloat())
59          {
60              // example: load r0, addr_a
61              cur_inst = new LoadMInstruction(cur_block, internal_reg1, dst);
62              cur_block->InsertInst(cur_inst);
63              // example: store r1, [r0]
64              cur_inst = new StoreMInstruction(cur_block, src, internal_reg2,
                    nullptr, StoreMInstruction::VSTR);
65              cur_block->InsertInst(cur_inst);
66          }
67          else
68          {
69              // example: load r0, addr_a
70              cur_inst = new LoadMInstruction(cur_block, internal_reg1, dst);
71              cur_block->InsertInst(cur_inst);
72              // example: store r1, [r0]
73              cur_inst = new StoreMInstruction(cur_block, src, internal_reg2);
74              cur_block->InsertInst(cur_inst);
75          }
76      }
77      // 如果存储的是局部变量
78      else if (operands[0]->getEntry()->isTemporary() && operands[0]->getDef()
            && operands[0]->getDef()->isAlloc())
79      {
80          // example: store r1, [r0, #4]
81          auto dst1 = genMachineReg(11);
82          int offset = dynamic_cast<TemporarySymbolEntry *>(operands[0]->
                getEntry())->getOffset();
83          auto dst2 = genMachineImm(offset);
84          if (offset > 255 || offset < -255)
85          {
86              auto internal_reg = genMachineVReg();
87              cur_inst = new LoadMInstruction(cur_block, internal_reg, dst2);
88              cur_block->InsertInst(cur_inst);
89              dst2 = new MachineOperand(*internal_reg);
90          }
91          if (operands[1]->getType()->isFloat())
92          {
93              // 对于浮点数 超过范围的 offset 需要再做处理
94              if (offset > 255 || offset < -255)
95              {
96                  auto reg = genMachineVReg();
97                  cur_inst = new BinaryMInstruction(cur_block,
                        BinaryMInstruction::ADD, reg, dst1, dst2);
98                  cur_block->InsertInst(cur_inst);
99                  dst2 = reg;
100             }
101             cur_inst = new StoreMInstruction(cur_block, src, dst1, dst2,
```

```
                StoreMInstruction::VSTR);
            cur_block->InsertInst(cur_inst);
        }
        else
        {
            cur_inst = new StoreMInstruction(cur_block, src, dst1, dst2);
            cur_block->InsertInst(cur_inst);
        }
    }
    // 如果存储的是临时变量的操作数
    else
    {
        // example: store r1, [r0]
        if (operands[0]->getEntry()->getType()->isArray())
        {
            // 如果是全局数组访问，不需要将offset与fp相加（丑）
            // 如果是函数参数传进来的，同理
            std::vector<int> dimensions;
            if (operands[0]->getEntry()->getType()->isIntArray())
            {
                dimensions = dynamic_cast<IntArrayType *>(operands[0]->
                    getEntry()->getType())->getDimensions();
            }
            else if (operands[0]->getEntry()->getType()->isConstIntArray())
            {
                dimensions = dynamic_cast<ConstIntArrayType *>(operands[0]->
                    getEntry()->getType())->getDimensions();
            }
            else if (operands[0]->getEntry()->getType()->isFloatArray())
            {
                dimensions = dynamic_cast<FloatArrayType *>(operands[0]->
                    getEntry()->getType())->getDimensions();
            }
            else
            {
                dimensions = dynamic_cast<ConstFloatArrayType *>(operands
                    [0]->getEntry()->getType())->getDimensions();
            }
            if (dynamic_cast<TemporarySymbolEntry *>(operands[0]->getEntry())
                ->getGlobalArray() ||
                dimensions[0] == -1)
            {
                auto dst_addr = genMachineOperand(operands[0]);
                if (operands[1]->getType()->isFloat())
                {
                    cur_inst = new StoreMInstruction(cur_block, src, dst_addr
                        , nullptr, StoreMInstruction::VSTR);
                    cur_block->InsertInst(cur_inst);
```

```
143                 }
144                 else
145                 {
146                     cur_inst = new StoreMInstruction(cur_block, src, dst_addr
                        );
147                     cur_block->InsertInst(cur_inst);
148                 }
149             }
150             else
151             {
152                 auto dst_addr = genMachineVReg();
153                 auto fp = genMachineReg(11);
154                 auto offset = genMachineOperand(operands[0]);
155                 if (offset->isImm())
156                 {
157                     if (((ConstantSymbolEntry *)(operands[0]->getEntry()))->
                        getValue() > 255 ||
158                         ((ConstantSymbolEntry *)(operands[0]->getEntry()))->
                            getValue() < -255)
159                     {
160                         auto internal_reg = genMachineVReg();
161                         cur_inst = new LoadMInstruction(cur_block,
                            internal_reg, offset);
162                         cur_block->InsertInst(cur_inst);
163                         offset = new MachineOperand(*internal_reg);
164                     }
165                 }
166                 cur_inst = new BinaryMInstruction(cur_block,
                    BinaryMInstruction::ADD, dst_addr, fp, offset);
167                 cur_block->InsertInst(cur_inst);
168 
169                 if (operands[1]->getType()->isFloat())
170                 {
171                     cur_inst = new StoreMInstruction(cur_block, src, new
                        MachineOperand(*dst_addr), nullptr, StoreMInstruction
                        ::VSTR);
172                     cur_block->InsertInst(cur_inst);
173                 }
174                 else
175                 {
176                     cur_inst = new StoreMInstruction(cur_block, src, new
                        MachineOperand(*dst_addr));
177                     cur_block->InsertInst(cur_inst);
178                 }
179             }
180         }
181         else
182         {
```

```
            auto dst = genMachineOperand(operands[0]);
            if (operands[1]->getType()->isFloat())
            {
                cur_inst = new StoreMInstruction(cur_block, src, dst, nullptr
                    , StoreMInstruction::VSTR);
                cur_block->InsertInst(cur_inst);
            }
            else
            {
                cur_inst = new StoreMInstruction(cur_block, src, dst);
                cur_block->InsertInst(cur_inst);
            }
        }
    }
}
```

LoadInstruction 指令

1. 加载一个全局变量或者常量
2. 加载一个栈中的临时变量
3. 加载一个数组元素数组元素的地址存放

StoreInstruction 同理

2. 二元运算指令的翻译

二元运算指令的翻译

```
// 二元运算指令翻译
void BinaryInstruction::genMachineCode(AsmBuilder *builder)
{
    // complete other instructions
    auto cur_block = builder->getBlock();
    auto dst = genMachineOperand(operands[0]);
    auto src1 = genMachineOperand(operands[1]);
    auto src2 = genMachineOperand(operands[2]);
    /* HINT:
     * The source operands of ADD instruction in ir code both can be
        immediate num.
     * However, it's not allowed in assembly code.
     * So you need to insert LOAD/MOV instrucrion to load immediate num into
        register.
     * As to other instructions, such as MUL, CMP, you need to deal with this
        situation, too.*/
    MachineInstruction *cur_inst = nullptr;
    if (src1->isImm())
    {
        auto internal_reg = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, internal_reg, src1);
        cur_block->InsertInst(cur_inst);
        src1 = new MachineOperand(*internal_reg);
```

```
    }
    if (opcode == MUL || opcode == DIV || opcode == MOD)
    {
        if (src2->isImm())
        {
            auto internal_reg = genMachineVReg();
            cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
            cur_block->InsertInst(cur_inst);
            src2 = new MachineOperand(*internal_reg);
        }
    }
    if (src2->isImm() && (src2->getVal() > 255 || src2->getVal() < -255))
    {
        auto internal_reg = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
        cur_block->InsertInst(cur_inst);
        src2 = new MachineOperand(*internal_reg);
    }
    switch (opcode)
    {
    case ADD:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
            dst, src1, src2);
        break;
    case SUB:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::SUB,
            dst, src1, src2);
        break;
    case MUL:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::MUL,
            dst, src1, src2);
        break;
    case DIV:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::DIV,
            dst, src1, src2);
        break;
    case MOD:
    {
        // a % b = a - a / b * b
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::DIV,
            dst, src1, src2);
        MachineOperand *dst1 = new MachineOperand(*dst);
        src1 = new MachineOperand(*src1);
        src2 = new MachineOperand(*src2);
        cur_block->InsertInst(cur_inst);
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::MUL,
            dst1, dst, src2);
        cur_block->InsertInst(cur_inst);
```

```
63          dst = new MachineOperand(*dst1);
64          cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::SUB,
                dst, src1, dst1);
65          break;
66      }
67      case AND:
68          cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::AND,
                dst, src1, src2);
69          break;
70      case OR:
71          cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::OR,
                dst, src1, src2);
72          break;
73      default:
74          break;
75      }
76      cur_block->InsertInst(cur_inst);
77  }
```

需要注意的是，中间代码中二元运算指令的两个操作数都可以是立即数，但这一点在汇编指令中是不被允许的。

根据汇编指令的规则，提前插入 LOAD 汇编指令，来将其中的某个操作数加载到寄存器中。需要注意的是，当第二个源操作数是立即数时，其数值范围有一定限制。

3. 比较指令的翻译

比较指令的翻译

```
1   // 比较指令的翻译
2   void CmpInstruction::genMachineCode(AsmBuilder *builder)
3   {
4       MachineBlock *cur_block = builder->getBlock();
5       MachineOperand *src1 = genMachineOperand(operands[1]);
6       MachineOperand *src2 = genMachineOperand(operands[2]);
7       MachineInstruction *cur_inst = nullptr;
8       if (src1->isImm())
9       {
10          MachineOperand *internal_reg = genMachineVReg();
11          cur_inst = new LoadMInstruction(cur_block, internal_reg, src1);
12          cur_block->InsertInst(cur_inst);
13          src1 = new MachineOperand(*internal_reg);
14      }
15      if (src2->isImm())
16      {
17          MachineOperand *internal_reg = genMachineVReg();
18          cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
19          cur_block->InsertInst(cur_inst);
20          src2 = new MachineOperand(*internal_reg);
21      }
```

```
    cur_inst = new CmpMInstruction(cur_block, src1, src2, opcode);
    cur_block->InsertInst(cur_inst);
    cur_block->setCurrentBranchCond(opcode);
    // 采用条件存储的方式将1/0存储到dst中
    MachineOperand *dst = genMachineOperand(operands[0]);
    MachineOperand *trueOperand = genMachineImm(1);
    MachineOperand *falseOperand = genMachineImm(0);
    cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
        trueOperand, opcode);
    cur_block->InsertInst(cur_inst);
    if (opcode == CmpInstruction::E || opcode == CmpInstruction::NE)
    {
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
            falseOperand, 1 - opcode);
    }
    else
    {
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
            falseOperand, 7 - opcode);
    }
    cur_block->InsertInst(cur_inst);
}
```

4. 控制流指令的翻译

UncondBrInstruction 的翻译

```
// 无条件跳转指令
void UncondBrInstruction::genMachineCode(AsmBuilder *builder)
{
    MachineBlock *cur_block = builder->getBlock();
    std::stringstream label;
    label << ".L" << branch->getNo();
    MachineOperand *dst = new MachineOperand(label.str());
    MachineInstruction *cur_inst = new BranchMInstruction(cur_block,
        BranchMInstruction::B, dst);
    cur_block->InsertInst(cur_inst);
}
```

要生成一条无条件跳转指令即可，至于跳转目的操作数的生成，大家只需要调用 genMachineLabel() 函数即可，参数为目的基本块号。

CondBrInstruction 的翻译

```
// 条件跳转指令
void CondBrInstruction::genMachineCode(AsmBuilder *builder)
{
    MachineBlock *cur_block = builder->getBlock();
    std::stringstream true_label, false_label;
```

```
    true_label << ".L" << true_branch->getNo();
    false_label << ".L" << false_branch->getNo();
    MachineOperand *true_dst = new MachineOperand(true_label.str());
    MachineOperand *false_dst = new MachineOperand(false_label.str());
    // 符合当前块跳转条件有条件跳转到真分支
    MachineInstruction *cur_inst = new BranchMInstruction(cur_block,
        BranchMInstruction::B, true_dst, cur_block->getCurrentBranchCond());
    cur_block->InsertInst(cur_inst);
    // 不符合当前块跳转条件无条件跳转到假分支
    cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::B,
        false_dst);
    cur_block->InsertInst(cur_inst);
}
```

于 CondBrInstruction，首先明确在中间代码中该指令一定位于 CmpInstruction 之后，对 CmpInstruction 的翻译比较简单，相信大家都能完成。对 CondBrInstruction，同学们首先需要在 AsmBuilder 中添加成员以记录前一条 CmpInstruction 的条件码，从而在遇到 CondBrInstruction 时生成对应的条件跳转指令跳转到 True Branch，之后需要生成一条无条件跳转指令跳转到 False Branch。

RetInstruction 的翻译

```
// 条件跳转指令
// return指令
void RetInstruction::genMachineCode(AsmBuilder *builder)
{
    auto cur_block = builder->getBlock();
    MachineInstruction *cur_inst = nullptr;
    // 1. Generate mov instruction to save return value in r0
    if (!operands.empty())
    {
        if (operands[0]->getType()->isFloat())
        {
            auto src = genMachineOperand(operands[0], true);
            if (src->isImm())
            {
                auto internal_reg = genMachineVReg();
                cur_inst = new LoadMInstruction(cur_block, internal_reg, src)
                    ;
                cur_block->InsertInst(cur_inst);
                src = internal_reg;
            }
            auto dst = new MachineOperand(MachineOperand::REG, 16, true);
            cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
                dst, src);
            cur_block->InsertInst(cur_inst);
        }
        else
        {
```

```
            auto src = genMachineOperand(operands[0]);
            // 立即数->寄存器
            if (src->isImm())
            {
                auto internal_reg = genMachineVReg();
                cur_inst = new LoadMInstruction(cur_block, internal_reg, src)
                    ;
                cur_block->InsertInst(cur_inst);
                src = new MachineOperand(*internal_reg);
            }
            auto dst = new MachineOperand(MachineOperand::REG, 0); // r0
            cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
                dst, src);
            cur_block->InsertInst(cur_inst);
        }
    }
    // 生成一条跳转到结尾函数栈帧处理的无条件跳转语句
    auto dst = new MachineOperand(".L" + this->getParent()->getParent()->
        getSymPtr()->toStr().erase(0, 1) + "_END");
    cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::B, dst);
    cur_block->InsertInst(cur_inst);
    // 接下来的工作放到MachineCode.cpp: void MachineFunction::output()完成
}
```

当函数有返回值时，我们需要生成 MOV 指令，将返回值保存在 R0 寄存器中；其次，我们需要生成 MOV 指令来恢复栈帧；如果该函数保存了被调用者保存寄存器，我们还需要生成 POP 指令恢复这些寄存器；最后再生成跳转指令来返回到 Caller。

5. 函数调用的翻译

函数调用的翻译

```
// 函数调用指令
void CallInstruction::genMachineCode(AsmBuilder *builder)
{
    int saved_reg_cnt = 0;
    auto cur_block = builder->getBlock();
    MachineInstruction *cur_inst = nullptr;
    std::vector<MachineOperand *> additional_args;
    // for(unsigned int i = 1;i < operands.size();i++){
    int iparam_cnt = 0;
    int fparam_cnt = 0;
    for (int i = 1; i < int(operands.size()); i++)
    {
        if (operands[i]->getType()->isInt())
        {
            iparam_cnt++;
        }
        else if (operands[i]->getType()->isFloat())
```

```
18          {
19              fparam_cnt++;
20          }
21          else if (operands[i]->getType()->isArray())
22          {
23              bool isPointer = false;
24              bool is_float = false;
25              if (operands[i]->getEntry()->getType()->isIntArray())
26              {
27                  isPointer = dynamic_cast<IntArrayType *>(operands[i]->
                        getEntry()->getType())->getPointer();
28                  is_float = false;
29              }
30              else if (operands[i]->getEntry()->getType()->isFloatArray())
31              {
32                  isPointer = dynamic_cast<FloatArrayType *>(operands[i]->
                        getEntry()->getType())->getPointer();
33                  is_float = true;
34              }
35              else if (operands[i]->getEntry()->getType()->isConstIntArray())
36              {
37                  isPointer = dynamic_cast<ConstIntArrayType *>(operands[i]->
                        getEntry()->getType())->getPointer();
38                  is_float = false;
39              }
40              else if (operands[i]->getEntry()->getType()->isConstFloatArray())
41              {
42                  isPointer = dynamic_cast<ConstFloatArrayType *>(operands[i]->
                        getEntry()->getType())->getPointer();
43                  is_float = true;
44              }
45              if (isPointer)
46              {
47                  iparam_cnt++;
48              }
49              else
50              {
51                  if (is_float)
52                  {
53                      fparam_cnt++;
54                  }
55                  else
56                  {
57                      iparam_cnt++;
58                  }
59              }
60          }
61      }
```

```
62      for (unsigned int i = operands.size() - 1; i > 0; i--)
63      {
64          // 需要保证不是值而是数组指针
65          bool isPointer = false;
66          // 如果类型是数组，需要考虑局部数组指针的情况
67          if (operands[i]->getEntry()->getType()->isArray())
68          {
69              if (operands[i]->getEntry()->getType()->isIntArray())
70              {
71                  isPointer = dynamic_cast<IntArrayType *>(operands[i]->
                        getEntry()->getType())->getPointer();
72                  // dynamic_cast<IntArrayType*>(operands[i]->getEntry()->
                        getType())->setPointer(false);
73                  // 如果第一维为-1，表明其为指针，传参时需要注意不加fp
74                  if (dynamic_cast<IntArrayType *>(operands[i]->getEntry()->
                        getType())->getDimensions()[0] == -1)
75                  {
76                      isPointer = false;
77                  }
78              }
79              else if (operands[i]->getEntry()->getType()->isFloatArray())
80              {
81                  isPointer = dynamic_cast<FloatArrayType *>(operands[i]->
                        getEntry()->getType())->getPointer();
82                  if (dynamic_cast<FloatArrayType *>(operands[i]->getEntry()->
                        getType())->getDimensions()[0] == -1)
83                  {
84                      isPointer = false;
85                  }
86              }
87              else if (operands[i]->getEntry()->getType()->isConstIntArray())
88              {
89                  isPointer = dynamic_cast<ConstIntArrayType *>(operands[i]->
                        getEntry()->getType())->getPointer();
90                  if (dynamic_cast<ConstIntArrayType *>(operands[i]->getEntry()
                        ->getType())->getDimensions()[0] == -1)
91                  {
92                      isPointer = false;
93                  }
94              }
95              else if (operands[i]->getEntry()->getType()->isConstFloatArray())
96              {
97                  isPointer = dynamic_cast<ConstFloatArrayType *>(operands[i]->
                        getEntry()->getType())->getPointer();
98                  if (dynamic_cast<ConstFloatArrayType *>(operands[i]->getEntry
                        ()->getType())->getDimensions()[0] == -1)
99                  {
100                     isPointer = false;
```

```
101                 }
102             }
103         }
104         // 表示传入的是一个数组并且是指针
105         if (isPointer)
106         {
107             --iparam_cnt;
108             MachineOperand *dst_addr = nullptr;
109             // 情况1 必须保证是局部数组, 而且不是传进来的参数 此时需要加fp
110             if (!dynamic_cast<TemporarySymbolEntry *>(operands[i]->getEntry()
                    )->getGlobalArray())
111             {
112                 auto fp = genMachineReg(11);
113                 auto offset = genMachineOperand(operands[i]);
114                 if (offset->isImm())
115                 {
116                     if (((ConstantSymbolEntry *)(operands[i]->getEntry()))->
                            getValue() > 255 ||
117                         ((ConstantSymbolEntry *)(operands[i]->getEntry()))->
                                getValue() < -255)
118                     {
119                         auto internal_reg = genMachineVReg();
120                         cur_inst = new LoadMInstruction(cur_block,
                                internal_reg, offset);
121                         cur_block->InsertInst(cur_inst);
122                         offset = new MachineOperand(*internal_reg);
123                     }
124                 }
125                 dst_addr = genMachineVReg();
126                 cur_inst = new BinaryMInstruction(cur_block,
                        BinaryMInstruction::ADD, dst_addr, fp, offset);
127                 cur_block->InsertInst(cur_inst);
128             }
129             else
130             {
131                 dst_addr = genMachineOperand(operands[i]);
132             }
133             // 全局数组或传入的数组指针参数, 此时一律按int处理, 但不需要加fp
134             // 而对于局部数组需要添加fp的已经在上面处理完
135             // 左起前4个参数通过r0-r3传递
136             if (iparam_cnt < 4)
137             {
138                 auto dst = new MachineOperand(MachineOperand::REG, iparam_cnt
                        ); // r0-r3
139                 cur_inst = new MovMInstruction(cur_block, MovMInstruction::
                        MOV, dst, dst_addr);
140                 cur_block->InsertInst(cur_inst);
141             }
```

```
142             else
143             {
144                 additional_args.clear();
145                 additional_args.push_back(dst_addr);
146                 cur_inst = new StackMInstruction(cur_block, StackMInstruction
                        ::PUSH, additional_args);
147                 cur_block->InsertInst(cur_inst);
148                 saved_reg_cnt++;
149             }
150         }
151         else
152         {
153             if (operands[i]->getType()->isFloat())
154             {
155                 --fparam_cnt;
156                 // 左起前4个参数通过s0-s3传递
157                 if (fparam_cnt < 4)
158                 {
159                     auto dst = new MachineOperand(MachineOperand::REG,
                            fparam_cnt + 16, true);
160                     auto src = genMachineOperand(operands[i], true);
161                     if (src->isImm())
162                     {
163                         auto internal_reg = genMachineVReg();
164                         cur_inst = new LoadMInstruction(cur_block,
                                internal_reg, src);
165                         cur_block->InsertInst(cur_inst);
166                         internal_reg = new MachineOperand(*internal_reg);
167                         cur_inst = new MovMInstruction(cur_block,
                                MovMInstruction::VMOV, dst, internal_reg);
168                         cur_block->InsertInst(cur_inst);
169                     }
170                     else
171                     {
172                         cur_inst = new MovMInstruction(cur_block,
                                MovMInstruction::VMOV, dst, src);
173                         cur_block->InsertInst(cur_inst);
174                     }
175                 }
176                 else
177                 {
178                     additional_args.clear();
179                     MachineOperand *operand = genMachineOperand(operands[i],
                            true);
180                     if (operand->isImm())
181                     {
182                         MachineOperand *internal_reg = genMachineVReg();
183                         cur_inst = new LoadMInstruction(cur_block,
```

```
                        internal_reg, operand);
184                     cur_block->InsertInst(cur_inst);
185                     operand = genMachineVReg(true);
186                     cur_inst = new MovMInstruction(cur_block,
                            MovMInstruction::VMOV, operand, internal_reg);
187                     cur_block->InsertInst(cur_inst);
188                 }
189                 additional_args.push_back(operand);
190                 cur_inst = new StackMInstruction(cur_block,
                        StackMInstruction::VPUSH, additional_args);
191                 cur_block->InsertInst(cur_inst);
192                 saved_reg_cnt++;
193             }
194         }
195         else
196         {
197             --iparam_cnt;
198             // 左起前4个参数通过r0-r3传递
199             if (iparam_cnt < 4)
200             {
201                 auto dst = new MachineOperand(MachineOperand::REG,
                        iparam_cnt); // r0-r3
202                 cur_inst = new MovMInstruction(cur_block, MovMInstruction
                        ::MOV, dst, genMachineOperand(operands[i]));
203                 cur_block->InsertInst(cur_inst);
204             }
205             else
206             {
207                 additional_args.clear();
208                 MachineOperand *operand = genMachineOperand(operands[i]);
209                 if (operand->isImm())
210                 {
211                     MachineOperand *internal_reg = genMachineVReg();
212                     cur_inst = new LoadMInstruction(cur_block,
                            internal_reg, operand);
213                     cur_block->InsertInst(cur_inst);
214                     operand = new MachineOperand(*internal_reg);
215                 }
216                 additional_args.push_back(operand);
217                 cur_inst = new StackMInstruction(cur_block,
                        StackMInstruction::PUSH, additional_args);
218                 cur_block->InsertInst(cur_inst);
219                 saved_reg_cnt++;
220             }
221         }
222     }
223 }
224 cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::BL, new
```

```
        MachineOperand(funcSE->getName(), true));
225     cur_block->InsertInst(cur_inst);
226     // 对于有返回值的函数调用 需要提供一条从mov r0, dst的指令
227     if (dynamic_cast<FunctionType *>(this->funcSE->getType())->getRetType()
            == TypeSystem::intType)
228     {
229         auto dst = genMachineOperand(operands[0]);
230         auto src = new MachineOperand(MachineOperand::REG, 0); // r0
231         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
                src);
232         cur_block->InsertInst(cur_inst);
233     }
234     else if (dynamic_cast<FunctionType *>(this->funcSE->getType())->
            getRetType() == TypeSystem::floatType)
235     {
236         auto dst = genMachineOperand(operands[0], true);
237         auto src = new MachineOperand(MachineOperand::REG, 16, true); // s0
238         cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV, dst,
                src);
239         cur_block->InsertInst(cur_inst);
240     }
241     // 恢复栈帧 调整sp
242     if (saved_reg_cnt)
243     {
244         auto src1 = genMachineReg(13);
245         auto src2 = genMachineImm(saved_reg_cnt * 4);
246         if (saved_reg_cnt * 4 > 255 || saved_reg_cnt * 4 < -255)
247         {
248             auto internal_reg = genMachineVReg();
249             cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
250             cur_block->InsertInst(cur_inst);
251             src2 = new MachineOperand(*internal_reg);
252         }
253         auto dst = genMachineReg(13);
254         cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
                dst, src1, src2);
255         cur_block->InsertInst(cur_inst);
256     }
257 }
```

在进行函数调用时，对于含参函数，需要使用 R0-R3 寄存器传递参数，如果参数个数大于四个还需要生成 PUSH 指令来传递参数；之后生成跳转指令来进入 Callee 函数；在此之后，需要进行现场恢复的工作，如果之前通过压栈的方式传递了参数，需要恢复 SP 寄存器；最后，如果函数执行结果被用到，还需要保存 R0 寄存器中的返回值。

6. 变量及常量的打印

变量及常量的打印

```
void MachineUnit::PrintGlobalDecl()
{
    if (global_var_list.empty())
    {
        return;
    }
    fprintf(yyout, "\t.data\n");
    for (auto var : global_var_list)
    {
        if (var->getType()->isArray())
        {
            if (var->arrayValues.empty())
            {
                fprintf(yyout, "\t.comm\t%s,%d,4\n", var->toStr().erase(0, 1)
                    .c_str(), var->getType()->getSize());
            }
            else
            {
                fprintf(yyout, "\t.global␣%s\n", var->toStr().erase(0, 1).
                    c_str());
                fprintf(yyout, "\t.align␣4\n");
                fprintf(yyout, "\t.size␣%s,␣%d\n", var->toStr().erase(0, 1).
                    c_str(), var->getType()->getSize());
                fprintf(yyout, "%s:\n", var->toStr().erase(0, 1).c_str());
                if (var->getType()->isIntArray() || var->getType()->
                    isConstIntArray())
                {
                    for (auto value : var->arrayValues)
                    {
                        fprintf(yyout, "\t.word␣%d\n", int(value));
                    }
                }
                else
                {
                    for (auto value : var->arrayValues)
                    {
                        auto tmp_value = float(value);
                        // uint32_t temp = reinterpret_cast<uint32_t&>(
                            tmp_value);
                        uint32_t temp;
                        std::memcpy(&temp, &tmp_value, sizeof(uint32_t));
                        fprintf(yyout, "\t.word␣%u\n", temp);
                    }
                }
            }
        }
        else
```

```
        {
            fprintf(yyout, "\t.global %s\n", var->toStr().erase(0, 1).c_str()
                );
            fprintf(yyout, "\t.align 4\n");
            fprintf(yyout, "\t.size %s, %d\n", var->toStr().erase(0, 1).c_str
                (), var->getType()->getSize());
            fprintf(yyout, "%s:\n", var->toStr().erase(0, 1).c_str());
            if (var->getType()->isInt())
            {
                fprintf(yyout, "\t.word %d\n", int(var->value));
            }
            else
            {
                auto value = float(var->value);
                // uint32_t temp = reinterpret_cast<uint32_t&>(value);
                uint32_t temp;
                std::memcpy(&temp, &value, sizeof(uint32_t));

                fprintf(yyout, "\t.word %u\n", temp);
            }
        }
    }
}
```

7. 相关 Output() 函数

BinaryMIstruction::Output() 函数

```
void BinaryMInstruction::output()
{
    switch (this->op)
    {
    case BinaryMInstruction::ADD:
        fprintf(yyout, "\tadd ");
        break;
    case BinaryMInstruction::SUB:
        fprintf(yyout, "\tsub ");
        break;
    case BinaryMInstruction::MUL:
        fprintf(yyout, "\tmul ");
        break;
    case BinaryMInstruction::DIV:
        fprintf(yyout, "\tsdiv ");
        break;
    case BinaryMInstruction::AND:
        fprintf(yyout, "\tand ");
        break;
    case BinaryMInstruction::OR:
```

```
        fprintf(yyout, "\tor ");
        break;
    case BinaryMInstruction::VADD:
        fprintf(yyout, "\tvadd.f32 ");
        break;
    case BinaryMInstruction::VSUB:
        fprintf(yyout, "\tvsub.f32 ");
        break;
    case BinaryMInstruction::VMUL:
        fprintf(yyout, "\tvmul.f32 ");
        break;
    case BinaryMInstruction::VDIV:
        fprintf(yyout, "\tvdiv.f32 ");
        break;
    default:
        break;
    }
    this->PrintCond();
    this->def_list[0]->output();
    fprintf(yyout, ", ");
    this->use_list[0]->output();
    fprintf(yyout, ", ");
    this->use_list[1]->output();
    fprintf(yyout, "\n");
}
//storeMInstruction ,MovMInstruction ,BranchMInstruction 等同理
```

8. 寄存器分配算法

linearScanRegisterAllocation() 函数

```
/线性扫描寄存器分配算法
bool LinearScan::linearScanRegisterAllocation()
{
    bool retValue = true;
    // 清空活跃和寄存器列表
    active.clear();
    regs.clear();
    fregs.clear();
     // 初始化整数寄存器列表
    for (int i = 4; i < 11; i++)
        regs.push_back(i);
     // 初始化浮点寄存器列表
    for (int i = 21; i < 48; i++)
        fregs.push_back(i);
    // 遍历每个变量的活跃区间
    for(auto &interval : intervals){
        expireOldIntervals(interval);// 处理过期的活跃区间
```

```
        //判断 active 列表中 interval 的数目和可用的物理寄存器数目是否相等
        if(interval->freg){
            if(fregs.size() == 0){//溢出
                spillAtInterval(interval);
                retValue = false;
            }
            else{//当前有可用于分配的物理寄存器
                interval->rreg = fregs[fregs.size()-1];// 为未处理的活跃区间
                    分配物理寄存器
                fregs.pop_back();
                active.push_back(interval);
                sort(active.begin(), active.end(), insertComp);
            }
        }
        else{
            if(regs.size() == 0){//溢出
                spillAtInterval(interval);
                retValue = false;
            }
            else{//当前有可用于分配的物理寄存器
                interval->rreg = regs[regs.size()-1];//为未处理的活跃区间分配
                    物理寄存器
                regs.pop_back();
                //再按照活跃区间结束位置, 将其插入到 active 列表中
                active.push_back(interval);
                sort(active.begin(), active.end(), insertComp);
            }
        }
    }
    return retValue;
}
```

算法遍历 intervals 列表，对遍历到的每一个活跃区间 i 都进行如下的处理：1. 遍历 active 列表，看该列表中是否存在结束时间早于区间 i 开始时间的 interval（即与活跃区间 i 不冲突），若有，则说明此时为其分配的物理寄存器可以回收，可以用于后续的分配，需要将其在 active 列表删除；

2. 判断 active 列表中 interval 的数目和可用的物理寄存器数目是否相等，

(a) 若相等，则说明当前所有物理寄存器都被占用，需要进行寄存器溢出操作。具体为在 active 列表中最后一个 interval 和活跃区间 i 中选择一个 interval 将其溢出到栈中，选择策略就是看哪个活跃区间结束时间更晚，如果是活跃区间 i 的结束时间更晚，只需要置位其 spill 标志位即可，如果是 active 列表中的活跃区间结束时间更晚，需要置位其 spill 标志位，并将其占用的寄存器分配给区间 i，再将区间 i 插入到 active 列表中。

(b) 若不相等，则说明当前有可用于分配的物理寄存器，为区间 i 分配物理寄存器之后，再按照活跃区间结束位置，将其插入到 active 列表中即可。

spillAtInterval() 函数

```
  /处理需要溢出到内存的活跃区间
void LinearScan::spillAtInterval(Interval *interval)
```

```
{
    // 如果 unhandled interval 的结束时间更晚
    if(active[active.size()-1]->end <= interval->end){
        interval->spill = true;//只需要置位其 spill 标志位
    }
    else{
        // active 列表中的 interval 结束时间更晚
        active[active.size()-1]->spill = true;//置位其spill标志位
        interval->rreg = active[active.size()-1]->rreg;//将其占用的寄存器分配
            给 unhandled interval
        // 将 unhandled interval 插入到 active 列表中
        active.push_back(interval);
        sort(active.begin(), active.end(), insertComp);
    }
}
```

1. 为其在栈内分配空间，获取当前在栈内相对 FP 的偏移;

2. 遍历其 USE 指令的列表，在 USE 指令前插入 LoadMInstruction，将其从栈内加载到目前的虚拟寄存器中;

3. 遍历其 DEF 指令的列表，在 DEF 指令后插入 StoreMInstruction，将其从目前的虚拟寄存器中存到栈内;

插入结束后，会迭代进行以上过程，重新计算活跃区间，进行寄存器分配，直至没有溢出情况出现。

9. 数组的翻译

数组的相关实现

```
class ArrayUtil {
    static Type* currentArrayType;
    static std::vector<int> arrayDims;
    static int currentArrayDim;
    static Operand* arrayAddr;// 数组的首地址
    static int currentOffset;// 当前数组偏移量
    static std::vector<ExprNode*> initVals;// 数组的初始化值
public:
    static void init();
    static void setArrayType(Type* type);
    static Type* getArrayType();
    static Type* getElementType();
    static void setCurrentArrayDim(int dim);
    static int getCurrentArrayDim();
    static void incCurrentDim();// 增加当前数组的维度
    static void decCurrentDim();
    static int getDimSize(int i);
    static int getCurrentDimCapacity();
    static void setArrayAddr(Operand* dst); // 设置数组的首地址
    static Operand* getArrayAddr();
```

```
    static void incCurrentOffset(); // 增加当前数组的偏移量
    static int getCurrentOffset();
    static void insertInitVal(ExprNode* val);// 插入数组的初始化值
    static void paddingInitVal(int size);// 对初始化值进行填充，使其达到指定大小
    static std::vector<ExprNode*>& getInitVals(); // 获取数组的初始化值列表
};
```

10. 浮点类型的翻译

浮点类型的翻译

```
// 浮点数二元运算指令
class FBinaryInstruction : public Instruction
{
public:
    FBinaryInstruction(unsigned opcode, Operand *dst, Operand *src1, Operand *src2, BasicBlock *insert_bb = nullptr);
    ~FBinaryInstruction();
    void output() const;
    void genMachineCode(AsmBuilder*);
    enum {ADD, SUB, MUL, DIV};
};

// 浮点数的比较指令
class FCmpInstruction : public Instruction
{
public:
    FCmpInstruction(unsigned opcode, Operand *dst, Operand *src1, Operand *src2, BasicBlock *insert_bb = nullptr);
    ~FCmpInstruction();
    void output() const;
    void genMachineCode(AsmBuilder*);
    enum {L, LE, G, GE, E, NE};
};
//整数和浮点数转换指令
class IntFloatCastInstructionn : public Instruction
{
public:
    IntFloatCastInstructionn(unsigned opcode, Operand *src, Operand *dst, BasicBlock *insert_bb = nullptr);
    ~IntFloatCastInstructionn();
    void output() const;
    void genMachineCode(AsmBuilder*);
    enum {I2F, F2I};
};
```

11. 实验效果

```
 7 main:
 8         push {r8, r9, r10, fp, lr}
 9         mov fp, sp
10         sub sp, sp, #8
11 .L4:
12         ldr r10, =0
13         str r10, [fp, #-4]
14         ldr r10, =0
15         str r10, [fp, #-8]
16         b .L8
17 .L8:
18         ldr r10, [fp, #-8]
19         ldr r9, =21
20         cmp r10, r9
21         movlt r10, #1
22         movge r10, #0
23         blt .L7
24         b .L12
25 .L7:
26         ldr r10, [fp, #-4]
27         ldr r9, [fp, #-8]
28         mul r8, r10, r9
29         str r8, [fp, #-4]
30         ldr r10, [fp, #-8]
31         add r9, r10, #1
32         str r9, [fp, #-8]
33         b .L8
34 .L12:
35         b .L9
36 .L9:
37         ldr r10, [fp, #-4]
38         mov r0, r10
39         bl putint
40         ldr r10, =0
41         mov r0, r10
42         b .Lmain_END
43 .Lmain_END:
44         add sp, sp, #8
45         pop {r8, r9, r10, fp, lr}
46         bx lr
47
```

图 2: 目标代码生成

测试样例的执行效果:

```
fufu@fufu-virtual-machine:/mnt/hgfs/share/Compile/final1/final$ make test TEST_PATH=test/level1-2
PASS: 012_func_defn
PASS: 013_var_defn_func
PASS: 028_if_test1
PASS: 029_if_test2
PASS: 030_if_test3
PASS: 031_if_test4
PASS: 032_if_test5
PASS: 035_while_test2
PASS: 036_while_test3
PASS: 039_while_if_test1
PASS: 040_while_if_test2
PASS: 041_while_if_test3
PASS: 069_greatest_common_divisor
PASS: 100_int_literal
PASS: 102_short_circuit3
PASS: 1074_itera_sqrt
PASS: 1076_int_factor_sum
PASS: 1078_decbinoct
PASS: 1080_lcm
PASS: 1083_enc_dec
Total: 20       Accept: 20      Fail: 0

make: *** [Makefile:95: test] 错误 1
fufu@fufu-virtual-machine:/mnt/hgfs/share/Compile/final1/final$ make test TEST_PATH=test/level2-1
PASS: 002_var_defn2
PASS: 060_scope
PASS: 096_side_effect
PASS: 101_scope2
Total: 4        Accept: 4       Fail: 0

make: *** [Makefile:95: test] 错误 1
fufu@fufu-virtual-machine:/mnt/hgfs/share/Compile/final1/final$ make test TEST_PATH=test/level2-2
PASS: 037_break
PASS: 038_continue
PASS: 052_comment1
PASS: 053_comment2
PASS: 090_int_io
Total: 5        Accept: 5       Fail: 0

make: *** [Makefile:95: test] 错误 1
fufu@fufu-virtual-machine:/mnt/hgfs/share/Compile/final1/final$ make test TEST_PATH=test/level2-3
```

图 3: 测试样例

希冀平台测试结果:

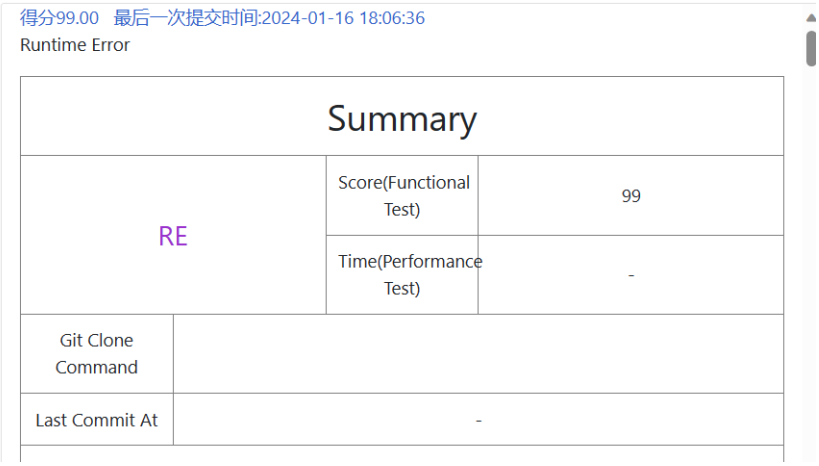


图 4: 测试样例

12. 代码链接

gitlab 代码链接：https://gitlab.eduxiji.net/nku20234/lab6final

二、 实验总结

(一) 小组分工

在了解实验指导书要求后讨论基础问题和提高部分，共同合作，互相交流讨论完成。

文档两人共同完成，负责不同代码部分如下说明。

词法分析阶段：

完成整形常量的词法分析；完成其他终结符的词法分析；完成单行注释和多行注释的词法分析；

语法分析阶段：

实现类型系统, 编写 Type.h(cpp); 合作完成抽象语法树构建，实现 Ast.h(cpp); 实现部分表达式的语法分析，构建正则表达式，完成 parser.y。

中间代码生成：

完成部分的类型检查，如：常量变量的未声明和重复声明，数值运算表达式，循环检查。

目标代码生成：完成生成部分指令的机器码，如：访存指令 load 和 store，二元运算指令和控制流指令。在寄存器分配阶段，对生成的溢出代码进行分析处理，处理溢出到栈的临时变量。

(二) 实验总结

学习了编译器的构造和各个阶段的具体实现，也在学习编程语言背后的原理和逻辑。词法分析、语法分析、类型检查、中间代码生成和目标代码生成这些环节，是编译器实现的步骤，编程语言设计的基石。

词法分析是编译器中的第一个阶段，它负责将源代码转换成标记（token），识别关键字、运算符、标识符等，并去除空格和注释。语法分析则负责将标记序列转换成抽象语法树（AST），以便后续的处理。类型检查是确保程序中的类型使用是正确的过程，它涉及到类型推断、类型转换和类型一致性检查等。中间代码生成阶段将 AST 转换成中间表示形式，通常是三地址码或者四地址码，这样可以方便后续的优化和目标代码生成。目标代码生成是将中间代码转换成目标机器的机器代码，这个阶段需要考虑目标机器的指令集和寄存器分配等问题。