实验报告

一、 实验目标

在语法实验的基础上，修改语义动作，输出 SysY 代码对应的 x86 AT&T 汇编代码。

二、 实现功能

能按要求翻译出汇编代码（并进行类型检查），但是存在部分缺陷：

1. 未进行寄存器等的优化
2. 定义数组时只能使用常数来定义数组的维度，不能使用变量或常量。比如样例中的 const int N = 10; int a[N]; 就不能编译通过，需要改成 int a[10];
   （但调用数组元素时可以令下标为变量）
3. 没有保证 16 对齐

三、 具体实现

1. 词法和语法部分

这两部分与之前相比改动不大，只是由于改用了 c++的语法，所以某些函数的形式发生了改变。另外为了配合汇编代码生成，对语法规则也进行了适量修改。

2. 语义规则

利用之前语法实验时构建的语法树，我选择的实现方式是对构建好的语法树进行类型检查，然后将其翻译为汇编代码。(在“test.cpp”中)

（1）Type 类来指示类型，包括 int、数组、函数等等，并且为数组和函数类型设置了相应的成员变量。

```
42  class Type
43  {
44  public:
45      bool constvar;
46      ValueType type;
47      Type(ValueType valueType); //初始化为valueType的类型
48      void copy(Type* a); //复制a
49
50  public:
51      unsigned short paramNum; //函数类型需要使用
52      Type* paramType[MAX_PARAM];
53      Type* retType;
54      void addParam(Type* t); //添加参数类型
55      void addRet(Type* t); //添加返回值类型
56
57      unsigned int dim;   //数组类型需要使用
58      ValueType elementType;
59      int dimSize[MAX_ARRAY_DIM];
60
61      unsigned int visitDim = 0; // 下一次使用下标运算符会访问的维度
62
63      int getSize();
64
```

```
65    public:
66        string getTypeInfo();
67        string getTypeInfo(ValueType type);
68    };
```

成员函数中，Type(ValueType valueType)和 copy 函数进行初始化，getTypeInfo()返回相应的类型信息，addParam 为函数类型添加参数的类型，addRet 为函数类型添加返回值的类型，getSize 则返回该类型所占空间大小。

（2）TreeNode 类，语法树节点，存储相应类型信息，以及运算符、表达式等的类型等等

```
struct TreeNode {
public:
    int num_lines; //行数


    TreeNode* child = nullptr;
    TreeNode* brother = nullptr;

    NodeType nodeType;
    OperatorType optype;// 运算符类型
    StmtType stype;     // 表达式类型
    Type* type;         // 变量、类型、表达式结点，有类型。
    int int_val;
    char ch_val;
    bool b_val;
    string str_val;
    string var_name;
    string var_scope;   // 变量作用域标识符

    TreeNode(int num_lines, NodeType type);
    TreeNode(TreeNode* node);   // 仅用于叶节点拷贝
    TreeNode(int num_lines, NodeType type, int val);
    void addChild(TreeNode*);
    void addBrother(TreeNode*);
    int getLength();
    int getVal();
```

```
// -------------- 类型检查 ----------------

void typeCheck();
void findReturn(vector<TreeNode *> &retList);

// ------------- asm 代码生成 -------------

int node_seq = 0;
int temp_var_seq = 0;
List list;

void gen_var_list();
void gen_str();

string new_list();
void get_list();

void genCode();


string getVarPos(TreeNode* p);
```

初始化函数以及 addChild 和 addBrother 函数都与语法实验中相似，进行初始化和子节点构建。getLength 返回子节点代表的链的长度，getVal 返回为常数的节点的值。typeCheck 进行类型检查。genCode 将语法树翻译成汇编代码，gen_var_list、gen_str、new_list、get_list、getVarPos 辅助进行汇编代码生成（详细生成过程之后介绍）。

（3）首先设置全局的变量列表

```
//<标识符名称， 作用域> 变量名列表（可重名）
extern multimap<string, string> idNameList;
//<<标识符名称， 作用域>，结点指针> 变量列表
extern map<pair<string, string>, TreeNode*> idList;

// 用于检查continue和break是否在循环内部
bool inCycle = false;
```

将 YYSTYPE 设置为 TreeNode*类型，在语义动作中构建语法树。

大多数语法规则的语义动作都是利用 TreeNode(int num_lines, NodeType type)和 addChild 来为待规约元素建立结点，并添加规约式中的非终结符结点作为子节点，此外还要根据规约式的性质（函数声明、表达式、语句等）来标记对应节点的 Type 类型。

但是对于变量的声明（Decl），要在识别出对应的标识符时检查其是否已经在与当前相交的作用域内被声明，并将未声明的标识符添加到变量列表中，比如：

```
DeclIdentifier: IDENT {
    $$ = $1;
    $$->var_scope = presentScope;
    $$->type = new Type(NOTYPE);
    if (idList.count(make_pair($$->var_name, $$->var_scope)) != 0) {
        string t = "Redeclared identifier : " + $$->var_name;
        yyerror(t.c_str());
    }
    idNameList.insert(make_pair($$->var_name, $$->var_scope));
    idList[make_pair($$->var_name, $$->var_scope)] = $$;
}
```

而对于变量的调用，也需要在识别出对应标识符时检查变量列表，看是否有可用作用域范围内的同名变量，并且如果存在的话需要选最近的定义，从变量列表中读取出，比如：

```
Identifiers :IDENT {
    $$ = $1;
    int idNameCount = idNameList.count($$->var_name);
    int declCnt = 0;
    int minDefDis = MAX_SCOPE_STACK;

    // 搜索变量是否已经声明
    auto it = idNameList.find($$->var_name);
    int resScoptCmp;
    while (idNameCount--) {
        resScoptCmp = scopeCmp(presentScope, it->second);
        if (resScoptCmp >= 0){
            // 寻找最近的定义
            if (resScoptCmp < minDefDis) {
                minDefDis = resScoptCmp;
                $$ = idList[make_pair(it->first, it->second)];
            }
            declCnt++;
        }
        it++;
    }
    if (declCnt == 0) {
        string t = "Undeclared identifier :'" + $1->var_name + "', scope
        yyerror(t.c_str());
    }
};
```

由于语法规则众多，其他语义动作不再一一说明。

3. 汇编代码生成

（1）gen_var_list()函数，生成对应区域变量列表

在根节点，即整个程序处，只处理全局变量的声明，用如下方式输出汇编代码

```
// 遍历常变量列表
int varsize = t->type->getSize();
if (t->type->dim > 0) {
    t->type->elementType = p->child->type->type;
    t->type->type = VALUE_ARRAY;
    varsize = t->type->getSize();
}
cout << "\t.globl\t" << t->var_name << endl
     << "\t.type\t" << t->var_name << ", @object" << endl
     << "\t.size\t" << t->var_name << ", " << varsize << endl
     << t->var_name << ":" << endl;
if (q->nodeType == NODE_OP && q->optype == OP_DECLASSIGN) {
```

而对于局部变量，并不需要直接输出汇编代码，只需要计算所占空间和每个变量的相对位置即可。对函数声明语句，需要递归查找局部变量的声明，分配空间；对于局部变量声明语句，则需要根据其类型判断分配空间的大小

```
else if (nodeType == NODE_STMT && stype == STMT_FUNCDECL) {
    // 对于函数声明语句，递归查找局部变量声明
    LocalVarList.clear();
    stackSize = -12;
    int paramSize = 8;
    // 遍历参数定义列表
    TreeNode *p = child->brother->brother->child;
    while (p) {
        // 只能是基本数据类型，简便起见一律分配4字节
        LocalVarList[p->child->brother->var_scope + p->child->brothe
        paramSize += 4;
        p = p->brother;
    }
    // 遍历代码段，查找函数内声明的局部变量
    p = child->brother->brother->brother->child;
    while (p) {
        p->gen_var_list();
        p = p->brother;
    }
}
```

（2）gen_str()函数，检测是否出现 scanf 和 printf，输出对应模式串

在 scanf 和 printf 被调用的情况下，输出其对应的模式串

```
while (p) {
    if (p->nodeType == NODE_VAR && p->var_name == "scanf") {
        if (!print_rodata) {
            print_rodata = true;
            cout << "\t.section\t.rodata" << endl;
        }
        if (!print_scanf) {
            print_scanf = true;
            cout << ".LC0:" << endl
                 << "\t.string\t" << "\"" << "%d" << "\"" << endl;
        }
    }
```

（3）new_list()函数，生成并返回新的基本块的标签

用静态变量统计已有标签的个数，生成新的标签

```
string TreeNode::new_list() {  //生成并返回新的基本块的标签（不断累加）
    static int list_seq = 0;
    string listStr = ".L";
    listStr += list_seq++;
    return listStr;
}
```

（4）get_list() 函数，分配并处理标签号

利用课上学过的知识，对 IF-ELSE、WHILE 语句及布尔表达式等的 true-list 和 false-list 进行处理，连接同样的 list，比如对 IF-ELSE 语句

```
case STMT_IFELSE:
    this->list.begin_list = new_list();
    this->list.true_list = new_list();
    this->list.false_list = new_list();
    this->list.next_list = new_list();
    this->child->list.true_list = this->list.true_list;
    this->child->list.false_list = this->list.false_list;
    break;
```

（5）getVarPos(TreeNode* p) 函数，返回访问对应变量的地址

全局变量直接返回其标签地址，局部变量则利用局部变量列表求得它的相对地址。

```
003  string TreeNode::getVarPos(TreeNode* p) { //返回对应变量名的访问
004      string varCode = "";
005      if (p->nodeType == NODE_VAR) {
006          // 标识符
007          if (p->var_scope == "1") {
008              // 全局变量
009              varCode = p->var_name;
010          }
011          else {
012              // 局部变量（不要跨定义域访问）
013              varCode += LocalVarList[p->var_scope + p->var_name];
014              varCode += "(%rbp)";
015          }
016      }
017      else {
018          // 数组
019          if (p->child->var_scope == "1") {
020              varCode = p->child->var_name + "(,%rax,4)";
021          }
022          else {
023              varCode += LocalVarList[p->child->var_scope + p->child->var_name];
024              varCode += "(%rbp,%rax,4)";
025          }
026      }
027      return varCode;
028  }
```

（6）genCode()函数，生成汇编代码

运用以上函数，对函数调用、STMT 语句、Exp 表达式、运算操作等不同的结点分别递归处

理。（以下通过举少数例子来说明，详见“test.cpp”）
比如函数调用要先将参数从右向左压栈，再 call 调用，如下：

```
    N = p->brother->getLength();

    q = new TreeNode *[N];  //为了从右向左压栈，存放反向的参数列表
    p = p->brother->child;
    while (p) {
        q[N - n++] = p;
        p = p->brother;
    }
    // 从右向左压栈
    for (int i = 0; i < N; i++) {
        q[i]->genCode();
        cout << "\tpushq\t%rax" << endl;
        pSize += this->child->type->paramType[i]->getSize();
    }
    // call和参数栈清理
    cout << "\tcall\t" << child->var_name << endl
        << "\taddq\t$" << pSize << ", %rsp" << endl;
}
break;
```

语句也要分类处理，如函数声明和 IF 语句，都要递归处理内部代码

```
case STMT_FUNCDECL:  //函数声明
    cycleStackTop = -1;
    pFunction = this;
    get_list(); //
    cout << "\t.globl\t" << p->brother->var_name << endl
        << "\t.type\t" << p->brother->var_name << ", @function" << endl
        << p->brother->var_name << ":" << endl;
    gen_var_list(); //生成变量列表
    cout << "\tpushq\t%rbp" << endl
        << "\tmovl\t%rsp, %rbp" << endl;
    // 在栈上分配局部变量
    cout << "\tsubq\t$" << -stackSize << ", %rsp" << endl;
    // 内部代码递归生成
    p->brother->brother->brother->genCode();
    // 产生返回标签代码
    cout << this->list.next_list << ":" << endl;
    // 清理局部变量栈空间
    cout << "\taddq\t$" << -stackSize << ", %rsp" << endl;
    cout << "\tpopq\t%ebp" << endl
        << "\tret" << endl;
    pFunction = nullptr;
    break;
```

```
case STMT_IF:
    get_list();
    cout << list.begin_list << ":" << endl;
    this->child->genCode();
    cout << list.true_list << ":" << endl;
    this->child->brother->genCode();
    cout << list.false_list << ":" << endl;
    break;
```

对表达式，令其值一律存放在%rax 寄存器中返回，以方便运算操作时处理

```
case NODE_EXPR:
    if (child->nodeType == NODE_VAR) {
        // 内存变量（全局/局部）
        string varCode = getVarPos(this->child);
        cout << "\tmovl\t" << varCode << ", %rax" << endl;
    }
    else if (child->nodeType == NODE_OP && child->optype == OP_INDEX) {
        // 数组
        child->genCode();
    }
    else {
        cout << "\tmovl\t$" << child->getVal() << ", %rax" << endl;
    }
    break;
```

对运算操作，分类处理，比如对大于关系和求和运算

```
case OP_GRA:
    p->genCode();
    cout << "\tpushq\t%rax" << endl;
    p->brother->genCode();
    cout << "\tpopq\t%rbx" << endl
         << "\tcmpq\t%rax, %rbx" << endl
         << "\tsetg\t%al" << endl;
    if (list.true_list != "") {
        cout << "\tjg\t\t" << list.true_list << endl
             << "\tjmp\t\t" << list.false_list << endl;
    }
    break;
```

```
case OP_ADD:
    p->genCode();
    cout << "\tpushq\t%rax" << endl;
    p->brother->genCode();
    cout << "\tpopq\t%rbx" << endl;
    cout << "\taddq\t%rbx, %rax" << endl;
    break;
```

四、 使用方法及样例展示

1. 使用方法（以 test.sy 为例）

   flex sysy.l

   yacc -d yacc.y

   g++ -c lex.yy.c

   g++ -c y.tab.c

   g++ -c test.cpp

   g++ lex.yy.o y.tab.o test.o -o main

   ./main < test.sy > test.out

2. 样例展示

   应用于 test.sy，得到输出如下（存放在 test.out）中

```
        .text
        .data
        .align  4
        .globl  N
        .type   N, @object
        .size   N, 4
N:
        .long   100
        .globl  a
        .type   a, @object
        .size   a, 400
a:
        .zero   100
        .text
        .globl  func
        .type   func, @function
func:
        pushq   %rbp
        movl    %rsp, %rbp
        subq    $12, %rsp
.L0:
        movl    8(%rbp), %rax
        pushq   %rax
        movl    $1, %rax
        popq    %rbx
        cmpq    %rax, %rbx
        sete    %al
        je              .L1
        jmp             .L4
        pushq   %rax
.L4:
        movl    8(%rbp), %rax
        pushq   %rax
        movl    $0, %rax
        popq    %rbx
        cmpq    %rax, %rbx
        sete    %al
        je              .L1
        jmp             .L2
        popq    %rbx
        orq     %rax, %rbx
        setne   %al
.L1:
        movl    $1, %rax
        jmp             .LRET_func
        jmp             .L3
.L2:
        movl    8(%rbp), %rax
        pushq   %rax
        movl    $1, %rax
        movl    %rax, %rbx
        popq    %rax
        subq    %rbx, %rax
        pushq   %rax
        call    func
        addq    $4, %rsp
        pushq   %rax
        movl    8(%rbp), %rax
        pushq   %rax
        movl    $2, %rax
        movl    %rax, %rbx
        popq    %rax
        subq    %rbx, %rax
        pushq   %rax
        call    func
        addq    $4, %rsp
        popq    %rbx
        addq    %rbx, %rax
        jmp             .LRET_func
.L3:
.LRET_func:
        addq    $12, %rsp
        popq    %ebp
        ret
        .globl  main
        .type   main, @function
main:
        pushq   %rbp
        movl    %rsp, %rbx
        subq    $16, %rsp
        movl    $10, %rax
        movl    %rax, -12(%rbp)
        movl    -12(%rbp), %rax
        pushq   %rax
        movl    $2, %rax
        popq    %rbx
        addq    %rbx, %rax
        pushq   %rax
        call    func
        addq    $4, %rsp
        pushq   %rax
        movl    $0, %rax
        popq    %rbx
        movl    %rbx, a(,%rax,4)
        movl    $0, %rax
        jmp             .LRET_main
.LRET_main:
        addq    $16, %rsp
        popq    %ebp
        ret
```

另外四个样例在此不再展示，存放在对应的 out 文件中