南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理大作业报告

SysY 编译器

刘嗣旸

学号：2013458

专业：计算机科学与技术

指导教师：王刚 李忠伟

2023 年 1 月 14 日

目录

一、 设计简述

(一) 报告说明

因为实现过程中细节太多，无法在报告中逐一说明具体实现，报告主要阐述我负责完成的重要核心的部分的具体实现，其余一些由框架提供的、比较简单或代码重复类似的内容就不在报告中详细说明。

在报告全文中展示的代码均为说明实现逻辑，因为原本代码较长，报告中会省略细节用“.....”代替。

(二) SysY 编译器总体设计

大作业实现的 SysY 编译器，功能上总体分为五个模块：词法分析器、语法分析器、类型检查、llvm 中间代码生成、armv8-a 汇编代码生成。按照先后次序依次完成了这五个部分的实现。

实现的编译器，支持 SysY 语言的子集，实现了基本要求的全部语言特性，实现了进阶部分的一部分语言特性。

基本数据类型支持 int、void 类型；支持常量、变量的声明和初始化，区分常量变量作用域；支持赋值、表达式、语句块、if、while、return、break、continue 等语句；支持算术运算（+、-、*、/、%，其中 +、-都可以是单目运算符）、关系运算（(==，>，<，>=，<=，!=)、逻辑运算（&&，||, !）等；支持对注释的识别；支持 SysY 运行时库；支持函数，函数调用，有参函数等。

(三) 各模块设计

词法分析器：完成了对注释，八进制、十六进制整数，各种终结符，运行时库的读取处理。

语法分析器：完成了符号表的相关功能、对空语句、控制流语句、while 语句、break 语句、continue 语句、二元表达式、一元表达式、函数调用、有参函数定义等内容的语法树结点的结构设计，并且通过 yacc 程序进行语法分析，生成语法树，同时完成打印语法树的功能。

类型检查：一部分错误通过构建语法树的过程中检查，一部分错误通过语法树建立后遍历语法树进行检查。

中间代码生成：类型检查完成后，对抽象语法树作一次遍历，遍历的过程中根据综合属性和继承属性来生成各结点的中间代码，通过指令对象的输出函数打印输出中间代码。

目标代码生成：类似于中间代码生成，对语法树自顶向下遍历，生成目标代码，然后完成寄存器分配工作。

二、 个人负责实现的工作

(一) 词法分析

1. 数制转换

通过正则表达式匹配十六进制、八进制、十进制数，在识别到十六进制和八进制数时，直接转换成十进制数存储。正则表达式和转换代码如下示例：

数制转换

```
OCTAL (0([1-7][0-7]*|0))
HEXADECIMAL (0[xX][0-9a-fA-f]+)
DECIMIAL ([1-9][0-9]*|0)

// 八进制转换
{OCTAL} {
    int octal;
    sscanf(yytext, "%o", &octal);
    if(dump_tokens)
        DEBUG_FOR_LAB4(yytext);
    yylval.itype = octal;
    return INTEGER;
}
```

2. 注释

注释的实现如下，当读到注释的起始标记，就会加入下面定义的规则，可以认为什么也不做，忽略注释内容，直到读到注释结束符才停止忽略：

注释

```

LINECOMMENT \/\/[^\n]*
COMMENTBEGIN "/*"
COMMENTELEMENT .|\n
COMMENTEND "*/"
%x BLOCKCOMMENT

```

```
%%
{COMMENTBEGIN} {BEGIN BLOCKCOMMENT;}
<BLOCKCOMMENT>{COMMENTELEMENT} {}
<BLOCKCOMMENT>{COMMENTEND} {BEGIN INITIAL;}
```

LINECOMMENT

3. 其他终结符

对于其他终结符如 int、void、+-*/等等，读到以后返回对应单词即可，如下所示：

其他终结符

```
"void" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("VOID\tvoid");
    return VOID;
}
```

(二) 语法分析

1. 二元表达式结点

二元表达式结点包含三个成员：两个表达式结点、一个运算符，结点的输出函数负责打印表达式结点的语法树结果，打印过程是一个递归的过程，通过 switch 语句根据运算符类型进行不同的操作，然后递归打印两个表达式的结构，递归过程中记录语法树的层次 level，通过 level 体现打印出的树的层次结构。结点的设计代码如下所示：

二元表达式结点

```
class BinaryExpr : public ExprNode
{
private:
    int op;
    ExprNode *expr1, *expr2;
public:
    enum {ADD, SUB, MUL, DIV, MOD, AND, OR, LESS, LEQU, EQUAL, NOTEQUAL,
        GREAT, GEQU};
         ……
    void output(int level);
};

void BinaryExpr::output(int level){
```

```
    std::string op_str;
    switch(op)
    {
        case ADD:
            op_str = "add";
            break;
                ……
        case MUL:
            op_str = "mul";
            break;
                ……
        case GREAT:
            op_str = "great";
            break;
                ……
    }
    fprintf(yyout, "%*cBinaryExpr\top: %s\n", level, ' ', op_str.c_str());
    expr1->output(level + 4);
    expr2->output(level + 4);
}
```

2. 一元表达式结点

一元表达式的结点设计和输出函数极为类似，只不过只有一个表达式，不再赘述。

3. 函数调用节点

函数调用结点包含一个表达式成员变量，是函数的参数。结点的输出函数首先通过符号表获取函数的名字和类型信息，打印函数信息，然后依次调用函数参数列表中参数的输出函数，打印每个参数信息。逻辑代码如下：

函数调用结点

```
class FunctionCallExpr : public ExprNode {
private:
    ExprNode* param;
public:
    FunctionCallExpr(SymbolEntry* se, ExprNode* param = nullptr) : ExprNode(
        se), param(param){};
    void output(int level);
};

void FunctionCallExpr::output(int level) {
    std::string name, type;
    int scope;
```

```
    name = symbolEntry->toStr();
    type = symbolEntry->getType()->toStr();
    scope = dynamic_cast<IdentifierSymbolEntry*>(symbolEntry)->getScope();
    fprintf(yyout, "%*cCallee\tfunction name: %s\ttype: %s\tscope:  %d\n",
            level, ' ', name.c_str(), type.c_str(), scope);
    Node* tmp = param;
    while (tmp) {
        tmp->output(level + 4);
        tmp = tmp->brother();
    }
}
```

4. 函数定义结点

函数定义的结点包含三个成员变量：函数对应的符号表项、函数体语句结点、函数声明语句结点，输出函数通过符号表项获取函数信息，打印函数信息然后递归调用语句结点的输出函数打印函数体内的语句。代码如下：

二元表达式结点

```
class FunctionDef : public StmtNode{
private:
    SymbolEntry *se;
    StmtNode *stmt;
    DeclStmt *decl;
public:
    FunctionDef(SymbolEntry *se, StmtNode *stmt, DeclStmt *decl = nullptr) :
        se(se), stmt(stmt), decl(decl){};
    void output(int level);
};

void FunctionDef::output(int level)
{
    std::string name, type;
    name = se->toStr();
    type = se->getType()->toStr();
    fprintf(yyout, "%*cFunctionDefine function name: %s, type: %s\n", level,
        ' ',
            name.c_str(), type.c_str());
    stmt->output(level + 4);
}
```

5. yacc 语法分析器构建

设计了语法树的结点以后，还要补充相应的 yacc 程序，yacc 程序在语法分析的过程中利用设计好的结点类型构建语法树，这里以函数定义语句的语法分析为例，语法分析器读到函数定义后，会创建相应的符号表，在递归创建完参数的符号表项和结点后，创建函数的符号表项，然后生成函数体的结点，最后生成上一小节函数定义的结点，代码逻辑如下：

函数定义语法分析

```
FuncDef
    :
    Type ID {
        identifiers = new SymbolTable(identifiers);
    }
    LPAREN FuncFParams RPAREN {
                // 创建函数的符号表项
                ……
    }
    BlockStmt
    {
                // 生成函数定义的结点
                ……
    }
    ;
```

(三) 类型检查

1. 总体执行逻辑简述

在生成语法树后，从 root 开始遍历语法树，依次调用结点的类型检查函数检查每个结点，有些错误在构建语法树的时候检查。

2. 函数声明调用部分问题

在语法分析构建语法树的阶段检查函数的未定义和重定义问题。检查未定义的逻辑如下只要查找符号表项，如果没有对应的符号表项，说明函数未定义，而对应重定义问题，只需要在插入符号表时进行检查，插入函数首先查找符号表，如果发现函数已经定义的表项，则插入失败，打印函数重定义的错误。逻辑代码如下：

函数声明调用部分问题

```
        // 未定义错误
    ID LPAREN FuncRParams RPAREN {
        SymbolEntry* se;
        se = identifiers->lookup($1);
        if(se == nullptr)
        {
            fprintf(stderr, "function \"%s\" is undefined\n", (char*)$1);
            delete [](char*)$1;
            assert(se != nullptr);
        }
        $$ = new FunctionCallExpr(se, $3);
    }

        // 函数重定义错误
            LPAREN FuncFParams RPAREN {
                ......
        bool success = identifiers->getPrev()->install($2, se);
        if(!success){
            fprintf(stderr, "redefinition of \'%s %s\'\n", $2, ((
                IdentifierSymbolEntry*)se)->getName().c_str()/*se->getType()
                ->toStr().c_str()*/);
        }
        $<se>$ = se;
    }
```

3. 函数调用部分问题

这里说明调用时参数不一致问题的检查，这个问题应当在构建函数调用结点的时候就进行检查，检查的思路是，获取函数的参数类型列表，遍历参数类型列表，将当前传入的实参类型列表和函数的参数类型列表逐一进行比较，如果不符合，则是传入的类型错误，遍历过程中如果传入的实参提前变成空，说明传入的参数太少，如果遍历结束传入的实参列表仍然未遍历完，说明传入的参数太多。逻辑代码如下：

函数调用参数检查

```
        for (auto i = params.begin(); i != params.end(); ++i) {
            if (temp == nullptr) {
                fprintf(stderr, "too few arguments to function %s %s\n",
                    symbolEntry->toStr().c_str(), type->toStr().c_str());
                break;
            } else if ((*i)->getKind() != temp->getType()->getKind()){
                fprintf(stderr, "parameter's type %s can't convert to %s\n",
                    temp->getType()->toStr().c_str(), (*i)->toStr().c_str());
            }
```

```
8             temp = (ExprNode*)(temp->brother());
9         }
10        // too many
11        if (temp != nullptr) {
12            fprintf(stderr, "too many arguments to function %s %s\n",
                  symbolEntry->toStr().c_str(), type->toStr().c_str());
13        }
```

4. 返回语句部分问题

返回语句的检查，只需要获取返回值和返回类型，然后分情况检查 void 类型有返回值、非 void 类型没有返回值、返回类型不匹配等问题打印错误信息即可。

(四) 中间代码生成

1. 总体执行逻辑简述

从 root 开始遍历语法树，调用结点 genCode 函数生成中间代码。基本上按照分支划分基本块，遍历到这个结点时，先生成基本块，再往基本块中插入指令，生成指令对象后就相当于生成了中间代码，指令对象中的输出函数最后会打印相应的中间代码。每个 genCode 函数结束以后，都要设置接下来要代码生成的辅助操作对象。

2. 函数定义和流图翻译

函数定义结点的中间代码生成逻辑比较简单，只需要递归地生成声明语句和函数体内的语句的中间代码即可，重点是基本块的控制流图的生成。关于流图的生成，遍历函数的所有基本块，因为基本块按照分支指令划分，所以检查每个基本块的最后一条指令，如果是条件分支指令，就要获取其对应的 true 和 false 分支块，如果块为空，就要插入返回指令结束，然后要把当前基本块和 true、false 块前后链接，形成流图，无条件跳转指令类似，如果当前基本块最后一条指令不是分支指令，就要检查是不是返回指令，根据判断结果，有必要时生成相应的返回指令。

具体的逻辑结构如下：

基本块流图

```
for (auto block = func->begin(); block != func->end(); block++){
    Instruction *last = (*block)->rbegin();
    ……
    if (last->isCond()) {
        ……
    }
    else if (last->isUncond()){
        ……
    }
    else {
        if (last->isRet() == false){
        ……
        }
    }
}
```

3. 声明语句翻译

声明语句的中间代码生成分为三种情况，检查是不是全局声明、是不是局部声明或者函数参数声明、如果是声明列表，继续调用下一个变量的 genCode 函数。如果是全局声明，需要加入编译单元维护的全局符号表中；局部声明，创建临时符号表项，生成分配空间的指令，对于参数和表达式，进行相应处理，生成 store 指令等，表达式还要递归调用生成代码的函数，最后如果是声明列表，就获取下一个声明变量，调用其中间代码生成函数。

代码逻辑框架如下：

声明语句翻译

```
IdentifierSymbolEntry *se = dynamic_cast<IdentifierSymbolEntry *>(id->
    getSymPtr());
if(se->isGlobal())
{
    ……
    unit.insertGlobal(se);
}
else if(se->isLocal() || se->isParam())
{
    ……
    alloca = new AllocaInstruction(addr, se);
    ……
    if (se->isParam() == true){
        ……
    }
    ……
```

```
        if (expr != nullptr){
            ……
        }
        if (se->isParam() == true){
            ……
        }
    }
    if (this->brother()) {
        this->brother()->genCode();
    }
```

4. 函数调用翻译

函数的调用语句，只需要遍历参数列表，调用参数表达式的生成函数依次生成中间代码，保存对应的操作数，最后利用参数的操作数列表生成函数调用指令即可。

5. 返回语句翻译

返回语句的翻译比较简单，如果有返回值，就递归调用返回值表达式结点的中间代码生成函数生成代码，然后保留返回值的操作数，生成一条返回指令即可。

（五） 目标代码生成

1. 总体执行逻辑简述

目标代码生成和中间代码生成类似，通过对中间代码进行自顶向下的遍历从而生成目标代码。通过已经建立的中间代码的函数、基本块等的结构，进行一一对应的构建相应目标代码函数基本块等的结构。然后同样递归地生成目标代码。例如，基本块的生成逻辑如下所示，注释部分解释了对应的含义：

基本块目标代码生成

```
void BasicBlock::genMachineCode(AsmBuilder* builder)
{
    auto cur_func = builder->getFunction(); // 获取当前操作函数
    auto cur_block = new MachineBlock(cur_func, no); // 创建当前操作基本块
    builder->setBlock(cur_block);
    for (auto i = head->getNext(); i != head; i = i->getNext())
    {
        i->genMachineCode(builder); // 给当前操作基本块内指令生成代码
    }
    cur_func->InsertBlock(cur_block);// 生成完的基本块插入其对应函数
}
```

2. 函数调用指令翻译

函数调用指令的翻译，如同中间代码生成，首先要获取当前操作的对象。然后要对参数情况分类讨论，如果实参数量小于 4 个，通过传参寄存器 r0 r3 保存，生成一条 mov 指令，同时，在目标代码所以涉及到立即数的地方都要进行合法性检查，因为 arm 汇编对立即数的要求，如果是非法立即数，要生成一条 load 指令装入寄存器。对于超过四个的寄存器，则通过栈传参，需要生成一条 push 指令压栈，最后，生成跳转链接指令切换到函数入口，函数执行完成后，如果有入栈的参数，还要恢复栈顶指针 sp 的位置，插入一条 add 指令，如果函数有返回值，还要将返回值保存到 call 指令的目的操作数中。

具体实现逻辑如下:

函数调用指令翻译

```
void FuncCallInstruction::genMachineCode(AsmBuilder *builder)
{
    auto cur_block = builder->getBlock();
    MachineInstruction* cur_inst;
    MachineOperand* operand;
        ……
    for (auto it = operands.begin(); (it != operands.end()) && (i < 5); ++it,
        ++i) { // 前四个参数
        ……
        cur_block->InsertInst(cur_inst);
    }

    for (int i = operands.size() - 1; i > 4; --i) { // 从右至左入栈
        operand = genMachineOperand(operands[i]);
        if (operand->isImm()) {
          ……
        }
        ……
        cur_inst = new StackMInstrcuton(cur_block, StackMInstrcuton::PUSH,
            vec, operand);
        cur_block->InsertInst(cur_inst);
    }

   ……
    cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::BL, L);
    cur_block->InsertInst(cur_inst);
    if (operands.size() > 5) {
        ……
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
            sp, sp, genMachineImm((operands.size() - 5) * 4));
```

```
28          cur_block->InsertInst(cur_inst);
29      }
30      if (dst != nullptr) {
31                  ......
32          cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
                operand, r0);
33          cur_block->InsertInst(cur_inst);
34      }
35  }
```

3. 返回指令翻译

函数的返回指令主要分为三个步骤，如果有返回值，将返回值保存在 r0 寄存器中，然后恢复现场，最后生成 bx 指令返回。

4. store 指令翻译

Store 指令考虑三种情况，操作对象是立即数，保存全局变量，保存局部变量。对于操作数是立即数，就要先把立即数 load 到虚拟寄存器；对于全局变量的保存，要先通过 load 把全局变量地址加载到寄存器，然后生成 store 指令保存；对于局部变量，要注意其寻址，通过基址寄存器和符号表中的栈内偏移获取其地址，然后生成 store 指令。

代码的逻辑结构如下所示：

store 指令翻译

```
1   void StoreInstruction::genMachineCode(AsmBuilder* builder)
2   {
3       ......
4       // Store imm
5       if (operands[1]->getEntry()->isConstant()) {
6           ......
7       }
8
9       if (operands[0]->getEntry()->isVariable()
10      && dynamic_cast<IdentifierSymbolEntry*>(operands[0]->getEntry())->
            isGlobal())
11      {
12          ......
13          // example: load r0, addr_a
14          cur_inst = new LoadMInstruction(cur_block, internal_reg1, dst);
15          cur_block->InsertInst(cur_inst);
16          // example: store r1, [r0]
17          cur_inst = new StoreMInstruction(cur_block, src, internal_reg1);
```

```
        cur_block->InsertInst(cur_inst);
    }

    else if (operands[0]->getEntry()->isTemporary()
    && operands[0]->getDef()
    && operands[0]->getDef()->isAlloc())
    {
        ……
        cur_inst = new StoreMInstruction(cur_block, src, src1, src2);
        cur_block->InsertInst(cur_inst);
    }
}
```

5. 二元指令翻译

二元运算指令的翻译，主要考虑如下问题即可：

首先二元指令不允许两个操作数都是立即数，所以第一个源操作数如果是立即数，就生成 load 指令加载到寄存器，同时对第二个立即数进行合法性检查，注意乘除指令等操作的都是寄存器；然后，只需要通过 switch 语句根据运算符生成对应的二元运算指令即可。

代码结构如下：

二元运算指令翻译

```
void BinaryInstruction::genMachineCode(AsmBuilder* builder)
{
    auto cur_block = builder->getBlock();
    auto dst = genMachineOperand(operands[0]);
    auto src1 = genMachineOperand(operands[1]);
    auto src2 = genMachineOperand(operands[2]);

    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);
    }
    /**
     * 判断立即数是否合法：在255之内，它一定是合法的
    **/
    if (src2->isImm()) {
        ……
    }
```

```
    switch (opcode)
    {
        ……
    }
    cur_block->InsertInst(cur_inst);
}
```

6. cmp 指令翻译

比较指令的翻译就要注意检查两个操作数，如果是立即数就先 load 到寄存器中。

7. store 指令打印

Store 指令只要获取相应的操作数和类型，然后按照汇编指令格式打印即可。

8. 二元指令和 cmp 指令打印

二元指令和 cmp 指令打印同样借助 switch 结构判断操作码类型，然后打印相应格式的指令即可。

9. 函数调用指令打印

函数调用指令只要注意遍历参数列表，依次打印参数部分即可。如下所示：

函数调用指令打印

```
void FuncCallInstruction::output() const
{
    fprintf(yyout, "  ");
    if (operands[0]){
        fprintf(yyout, "%s = ", operands[0]->toStr().c_str());
    }
    FunctionType* type = (FunctionType*)(func->getType());
    fprintf(yyout, "call %s %s(", type->getRetType()->toStr().c_str(), func->
        toStr().c_str());
    for (uintmax_t i = 1; i < operands.size(); ++i) {
        if (i != 1){
            fprintf(yyout, ", ");
        }
        fprintf(yyout, "%s %s", operands[i]->getType()->toStr().c_str(),
            operands[i]->toStr().c_str());
    }
    fprintf(yyout, ")\n");
}
```

三、 总结

这学期的编译器大作业，对我而言无疑是历次作业中最有挑战性的任务，虽然实现编译器的过程困难重重，甚至不得不去请教同学，不得不参考学习成熟的代码，虽然最终尽自己所能只能实现基本要求和一小部分的进阶要求内容，但是最后能看到自己和队友通过一学期的努力，完成一个如此庞大的工程项目，能自己亲手完成一个编译器，编译出正确的汇编代码文件，就有一种成就感，虽然编译器没有实现大多数的进阶要求的功能，功能还很有限，但对我来说，这个过程已经足够充满挑战性，我已经从中收获了足够多的知识、经验、乐趣和成就感。

四、 源码 GitLab 链接

代码 GitLab 链接:https://gitlab.eduxiji.net/nku-jojo/compiler.git