南開大學
Nankai University

南开大学

计算机学院和网络空间安全学院

编译系统原理实验报告

SysY 编译器设计与实现

朱子轩 2110853 计算机科学与技术

年级：2021 级

指导教师：王刚

https://gitlab.eduxiji.net/nku2023/nku2023_compiler

2024 年 1 月 16 日

摘要

本次编译系统原理工程作业完成了从词法分析、语法分析、类型检查、中间代码生成再到目标代码生成及优化的全过程，实现了自己定义的 SysY 语言特性中的全部内容，完成了对变量作用域的区分、break 和 continue 等语句、非叶函数和数组声明及初始化等功能。

在给定的模板的基础上，我们调整了整体框架，使之能够更好的实现相关功能，最终完整构建了 SysY 语言编译器。

关键字： SysY 语言，ARM，词法分析，语法分析，类型检查，中间代码生成，目标代码生成及优化，编译系统原理

目录

一、 分工

我主要完成的工作如下：

- **词法分析**：不同进制的整数和浮点数正则表达式、部分常规符号的识别
- **语法分析**：符号表、函数、浮点数、条件表达式
- **类型检查**：常量的使用、变量使用、函数
- **中间代码生成**：部分控制流、函数、浮点数
- **目标代码生成**：控制流语句、函数调用、浮点数、寄存器分配
- **代码优化**：不可达代码删除

二、 词法分析

(一) 整型常量

此处，主要考虑定义八进制与十六进制的规则，将其保存为十进制输出。

首先定义其正则表达式：

整型常量的正则表达式

```
/*十进制整数*/
DECIMIAL ([1-9][0-9]*|0)
/*十六进制数*/
HEXNUM 0[xX][0-9]+
/*八进制数*/
OCTNUM 0[1-9][0-9]*

DECIMAL_FLOAT ([+-]?(([0-9]*[.][0-9]*([eE][+-]?[0-9]+)?)|([0-9]+[eE
    ][+-]?[0-9]+))[fL|L]?)
HEXADECIMAL_FLOAT (0[xX](([0-9A-Fa-f]*[.][0-9A-Fa-f]*([pP][+-]?[0-9]+)?)
    |([0-9A-Fa-f]+[pP][+-]?[0-9]+))[fL|L]?)
```

接着在下面的规则段中，完成对其规则的编写：

整型常量的规则

```
{OCTNUM} {
  std::string s(yytext);
  int octnum = std::stoi(s, nullptr, 8);
  dump_tokens("OCTNUM\t%s\t%d\t%d\n", yytext, yylineno - 1, column, octnum);
  column += yyleng;
}
{HEXNUM} {
  std::string s(yytext);
  int hexnum = std::stoi(s, nullptr, 16);
  dump_tokens("HEXNUM\t%s\t%d\t%d\n", yytext, yylineno - 1, column, hexnum);
  column += yyleng;
}
```

还有其他的一些简单的标识定义，在此不再赘述。

三、 语法分析

(一) 浮点数

我们将常数改成 double 类型，以此兼容浮点数，其余增加浮点数相关的必要判断即可。

(二) 符号表

项目中的 lookup 函数实现了符号表的层次性查找：从最内层的符号表开始，逐级向外查找，直到找到匹配的标识符或者达到最外层符号表为止。这种符号表的结构通常用于处理不同作用域中的标识符，以支持作用域嵌套的编程语言特性。如果找到了标识符，它会返回对应的符号表项，否则返回 nullptr，表示标识符未定义或不可见。

lookup 函数

```
SymbolEntry *SymbolTable::lookup(std::string name)
{
    SymbolTable *current = identifiers;
    while (current != nullptr)
        if (current->symbolTable.find(name) != current->symbolTable.end())
            return current->symbolTable[name];//在当前符号表的 symbolTable 中查找名为 name 的标识符
        else
            current = current->prev;//将当前符号表指针移动到上一级符号表，以便在更外层的符号表中查找标识符
    return nullptr;
}
```

(三) 翻译模式

该部分是语法分析部分内容的重中之重，需设计各种语法块的翻译模式。项目中使用 Bison 和 Flex 编写了一个语法分析器。为了避免报告冗长，此处仅作部分展示。

在框架代码之上，为了完善我们的编译器功能，我们又添加了许多终结符与非终结符，如 FLOAT、CONTINUE、FuncDefParams 等。

1. 算符优先级

为了方便后续表达式的书写，我们设计了算符的优先级，这样后续所有的表达式可以直接给出，而不必关心优先级导致的二义性文法。

aa

```
%left OR
%left AND
%left EQUAL NOTEQUAL
%left LESS LESSEQUAL GREATER GREATEREQUAL
%left ADD SUB
%left MUL DIV MOD
%precedence UMINUS
```

2. FuncDef

此部分主要用于解析函数定义，由三个主要的产生式组成。第一个产生式处理函数定义的开始部分，包括函数返回类型、函数名字和左括号。它创建了一个新的函数类型（FunctionType），创建一个函数标识符并将其安装到符号表，然后创建一个新的符号表作为函数的参数符号表。第二个产生式在处理函数参数列表之后创建了一个新的符号表。第三个产生式的动作部分从之前的符号表中获取函数的参数类型，并使用这些信息创建一个新的函数定义对象。然后弹出两个符号表，一个是函数体内的，一个是函数参数，并删除它们。最后回到全局作用域。

FuncDef

```
FuncDef
    :
    Type ID LPAREN {
        Type *funcType;
        funcType = new FunctionType($1,vector<Type*>());
        if(identifiers->checkExist($2))
        {
            fprintf(stderr, "redefined function\n");
            exit(-1);
        }
        SymbolEntry *se = new IdentifierSymbolEntry(funcType, $2, identifiers
            ->getLevel());
        identifiers->install($2, se);
        identifiers = new SymbolTable(identifiers);
    }
    FuncDefParams RPAREN {
        identifiers = new SymbolTable(identifiers);
        DeclStmt* curr = (DeclStmt*)$5;
        int i = 0;
        while(curr != nullptr)
        {
            ((IdentifierSymbolEntry*)curr->getId()->getSymPtr())->setParamNo
                (++i);
            std::string name = ((IdentifierSymbolEntry*)curr->getId()->
                getSymPtr())->getName();
            SymbolEntry *se = new IdentifierSymbolEntry(curr->getId()->
                getSymPtr()->getType(), name, identifiers->getLevel());
            identifiers->install(name, se);
            curr = (DeclStmt*)(curr->getNext());
        }
    }
    funcBlock {
        vector<Type*> paramsType;
        DeclStmt* curr = (DeclStmt*)$5;
        DeclStmt* paramDecl = nullptr;
        //......
    }
    ;
```

（四） 抽象语法树

在语法树的构建部分较为简单，将重要的参数存入类中即可。在此不多赘述。

四、 语义分析（类型检查）

(一) 常量和变量

在类型检查时，需要在赋值语句中检查目标是否为常量，如果是，则报错。
在表达式中，需要检查运算对象是否为 void，是则报错。

(二) 函数相关

在遍历函数调用节点时，对函数的变量个数和返回类型进行检查，如果不匹配则报错。

FuncCallExp

```
void FuncCallExp::typeCheck()
{
    FunctionType* t = (FunctionType*)funcSym->getType();
    ExprNode* curr = param;
    int i = 0;
    while(curr != nullptr)
    {
        curr->typeCheck();
        Type* p = t->getParamsType(i);
        if(p == nullptr)
        {
            fprintf(stderr, "FuncCallExp param number error\n");
            exit(-1);
        }
        Type* currType = curr->getSymPtr()->getType();
        if(currType->isFunc())
            currType = ((FunctionType*)currType)->getRetType();
        if(!Type::isValid(p, currType))
        {
            fprintf(stderr, "FuncCallExp param type error\n");
            exit(-1);
        }
        i++;
        curr = (ExprNode*)curr->getNext();
    }
    if(t->getParamsType(i) != nullptr)
    {
        fprintf(stderr, "FuncCallExp param number error\n");
            exit(-1);
    }
}
```

(三) 循环语句

为了检查 break 和 continue 是否出现在循环体内，我们设置全局变量 isInwhile，每当进入 while 则保存该变量，将其值置为 false，当前 while 检查结束后将其值恢复。

WhileStmt

```
void WhileStmt::typeCheck()
{
    cond->typeCheck();
    bool tmp = isInwhile;
    isInwhile = true;
    stmt->typeCheck();
    isInwhile = tmp;
}
```

五、 中间代码生成

(一) 浮点数

浮点数主要问题是输出。使用强制类型转换将其输出:

```
uint64_t val = reinterpret_cast<uint64_t &>(temp);
```

(二) 中间代码指令

完善了函数调用指令、函数定义、全局变量的定义。在这里展示一条函数调用指令的输出。

```
GlobalVarDefInstruction::GlobalVarDefInstruction(Operand *dst,
    ConstantSymbolEntry *se, BasicBlock *insert_bb) : Instruction(GLOBALVAR,
    insert_bb), dst(dst)
{
    type = ((PointerType *)dst->getType())->getValueType();
    if (se == nullptr)
    {

        if (type->isInt())
            value.intValue = 0;
        else if (type->isFloat())
            value.floatValue = 0;
    }
    else
    {
        if (se->getType()->isInt())
        {
            if (type->isInt())
                value.intValue = se->getInt();
            else if (type->isFloat())
                value.intValue = se->getFloat();
        }
        else if (se->getType()->isFloat())
        {
            if (type->isInt())
                value.floatValue = se->getInt();
            else if (type->isFloat())
                value.floatValue = se->getFloat();
        }
    }
    dst->setDef(this);
}
```

(三) 控制流分析

我主要实现的是 while 的控制流语句的翻译。将 while 语句分为起始块、条件块、循环体块和末尾块，据此写出控制流。

```
void WhileStmt::genCode()
{
    Function *func;
    BasicBlock *while_bb, *then_bb, *end_bb;

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

    new UncondBrInstruction(while_bb, builder->getInsertBB());
    builder->setInsertBB(while_bb);
    cond->genCode();
    if(cond->getOperand()->getType() != TypeSystem::boolType)
        cond->toBool(func);
    backPatch(cond->trueList(), then_bb);
    backPatch(cond->falseList(), end_bb);
    while_bb = builder->getInsertBB();

    builder->setInsertBB(then_bb);
    stmt->genCode();
    backPatch(builder->getWhileList(true), while_bb);
    backPatch(builder->getWhileList(false), end_bb);
    then_bb = builder->getInsertBB();

    new UncondBrInstruction(while_bb, then_bb);

    builder->setInsertBB(end_bb);

}
```

我们为 builder 新增 truewhileList、falseWhileList 项，用于维护过程中 while 语句的真假分支。这样在遇到 break 和 continue 的时候能找到正确的路径。

```
void BreakStmt::genCode()
{
    BasicBlock* bb = builder->getInsertBB();
    Function *func = bb->getParent();
    BasicBlock* breakBB = new BasicBlock(func);
    new UncondBrInstruction(breakBB, bb);
    // builder->setInsertBB(breakBB);
    BasicBlock* tempBB = new BasicBlock(func);
    builder->addWhileList(new UncondBrInstruction(tempBB, breakBB), false);
    // builder->setInsertBB(bb);
}

void ContinueStmt::genCode()
{
```

```
    BasicBlock* bb = builder->getInsertBB();
    Function *func = bb->getParent();
    BasicBlock* continueBB = new BasicBlock(func);
    new UncondBrInstruction(continueBB, bb);
    // builder->setInsertBB(continueBB);
    BasicBlock* tempBB = new BasicBlock(func);
    builder->addWhileList(new UncondBrInstruction(tempBB, continueBB), true);
    // builder->setInsertBB(bb);
}
```

(四) 函数

在分析完成一个函数后，需要为各个块建立前驱后继关系。由于该部分代码过长，所以只在这里列出执行的一些操作：

- 删除无条件跳转指令之后的指令。
- 删除 ret 之后的指令。
- 删除前面的条件跳转指令（其实是生成的时候生成多了）。
- 根据块末尾的跳转关系建立块之间的控制流图。

六、 目标代码生成

(一) 控制流与浮点数

跳转指令的翻译比较简单，在此不再赘述。

与浮点数相关的部分，需要调用专门的函数处理，如 ___aeabi_fcmplt 等。

(二) 函数

在进入函数时，需要做以下操作:

- 保存函数中改变且需要保存的寄存器、栈顶指针 fp 和返回地址 lr。
- 将栈底指针 sp 移动到 fp 的位置。
- 获取栈上要分配的空间，移动栈顶指针。

```
void MachineFunction::output()
{
    fprintf(yyout, "\t.global %s\n", this->sym_ptr->toStr().c_str() + 1);
    fprintf(yyout, "\t.type %s , %%function\n", this->sym_ptr->toStr().c_str
        () + 1);
    fprintf(yyout, "%s:\n", this->sym_ptr->toStr().c_str() + 1);

    auto fp = new MachineOperand(MachineOperand::REG, 11);
    auto sp = new MachineOperand(MachineOperand::REG, 13);
    auto lr = new MachineOperand(MachineOperand::REG, 14);
    (new StackMInstrcuton(nullptr, StackMInstrcuton::PUSH, getSavedRegs(), fp
        , lr))->output();
    (new MovMInstruction(nullptr, MovMInstruction::MOV, fp, sp))->output();
    int off = AllocSpace(0);
    auto size = new MachineOperand(MachineOperand::IMM, off);
    (new BinaryMInstruction(nullptr, BinaryMInstruction::SUB, sp, sp, size))
       ->output();
    int count = 0;
    for (auto iter : block_list)
    {
        iter->output();
        count += iter->getSize();
    }
    fprintf(yyout, "\n");
}
```

在生成机器操作符的时候也要对参数进行考虑，如果参数个数小于 4，就直接使用 r0 到 r3，如果超过了，就先用 r3 代替，后续在基本块的输出部分处理:

```
// ...
else if (id_se->isParam())
{
    if (id_se->getParamNo() < 4)
```

```
    {
        mope = new MachineOperand(MachineOperand::REG, id_se->getParamNo());
    }
    else
    {
        mope = new MachineOperand(MachineOperand::REG, 3);
    }
}
```

对于参数较多需要存在栈中的值，需要使用 r3 做中中转，将值存入 r3 并计算位置并读入。

```
if (num > 4 && (*it)->isStore())
{
    MachineOperand *operand = (*it)->getUse()[0];
    if (operand->isReg() && operand->getReg() == 3)
    {
        if (first)
        {
            first = false;
        }
        else
        {
            auto fp = new MachineOperand(MachineOperand::REG, 11);
            auto r3 = new MachineOperand(MachineOperand::REG, 3);
            auto off = new MachineOperand(MachineOperand::IMM, offset);
            offset += 4;
            auto cur_inst = new LoadMInstruction(this, r3, fp, off);
            cur_inst->output();
        }
    }
}
```

在函数退出时，需要进行以下操作:

- 如果有返回值，将其移入 r0 寄存器。
- 栈顶指针归位。
- 使用 bx 指令返回。

```
void RetInstruction::genMachineCode(AsmBuilder *builder)
{
    auto cur_block = builder->getBlock();
    if (!operands.empty())
    {
        auto dst = new MachineOperand(MachineOperand::REG, 0);
        auto src = genMachineOperand(operands[0]);
        auto cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
            dst, src);
        cur_block->InsertInst(cur_inst);
```

```
    }
    auto cur_func = builder->getFunction();
    auto sp = new MachineOperand(MachineOperand::REG, 13);
    auto size = new MachineOperand(MachineOperand::IMM, cur_func->AllocSpace
        (0));
    auto cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD
        , sp, sp, size);
    cur_block->InsertInst(cur_inst);
    auto lr = new MachineOperand(MachineOperand::REG, 14);
    auto cur_inst2 = new BranchMInstruction(cur_block, BranchMInstruction::BX
        , lr);
    cur_block->InsertInst(cur_inst2);
}
```

由于寄存器分配之后栈顶的偏移量会改变，所以需要重新设置该指令的值。由于其永远处于 bx 指令之前，所以判断 bx 前的加法指令，为其设置相应的值：

```
if ((*it)->isAdd())
{
    auto dst = (*it)->getDef()[0];
    auto src1 = (*it)->getUse()[0];
    if (dst->isReg() && dst->getReg() == 13 && src1->isReg() && src1->getReg
        () == 13 && (*(it + 1))->isBX())
    {
        int size = parent->AllocSpace(0);
        (*it)->getUse()[1]->setVal(size);
    }
}
```

在基本块的输出时，增加对 bx 指令的判断，从栈中弹出相应的寄存器：

```
//。。。
if ((*it)->isBX())
{
    auto fp = new MachineOperand(MachineOperand::REG, 11);
    auto lr = new MachineOperand(MachineOperand::REG, 14);
    auto cur_inst = new StackMInstrcuton(this, StackMInstrcuton::POP, parent
        ->getSavedRegs(), fp, lr);
    cur_inst->output();
}
```

(三) 寄存器分配

使用线性扫描算法分配寄存器的步骤如下：

- 计算变量的活跃区间。
- 遍历活跃区间，分配寄存器 4 到 11。
- 如果没有冲突，修改虚拟寄存器，分配结束。

- 如果存在冲突，分割活跃区间，重新分配寄存器。

寄存器分配的规则如下：

- 遍历活跃区间。
- 从在当前遍历的区间之前的区间中将寄存器回收。
- 如果可分配的寄存器为空，则有冲突，分割活跃区间。
- 如果可分配的寄存器不为空，则没有冲突

七、 删除不可达代码

不可达代码的删除需要分函数删除，构建的步骤如下：

- 构建函数各基本块的控制流图，为了方便，使用块在函数基本块中的下标标识块。
- 对图做广度优先遍历，获取可达块。
- 遍历可达块和所有块，找出一个可达块中没有的块，删除。
- 递归调用函数，继续上述操作直到可达块和所有块一致。

递归调用的部分代码如下：

```
void ElimUnreachCode::pass(Function *func)
{
    auto blocks = getReachBlocks(func, 0);
    auto &blockList = func->getBlockList();
    int len = blockList.size();
    bool again = false;
    int i;
    for (i = 1; i < len; i++)
    {
        if (find(blocks.begin(), blocks.end(), i) == blocks.end())
        {
            again = true;
            break;
        }
    }
    if (again)
    {
        BasicBlock *block = blockList[i];
        for (auto iter = block->pred_begin(); iter != block->pred_end(); iter
            ++)
        {
            (*iter)->removeSucc(block);
        }
        for (auto iter = block->succ_begin(); iter != block->succ_end(); iter
            ++)
        {
            (*iter)->removePred(block);
        }
        blockList.erase(blockList.begin() + i);
        pass(func);
    }
}
```