南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

上机大作业

完成编译器

小组成员：孙国桐，王耀

实验报告作者：孙国桐 2113358

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2024 年 1 月 16 日

摘要

在此次实验中，在此前一次次实验的基础上，我们完成了我们的 Sysy 语言编译器，它实现了所要求的基础功能以及 break、continue、数组等进阶功能。通过这门课程的实验，对于编译器的工作流程与工作原理有了一个大致的理解，这学期的实验经历对我来说十分的难能可贵。

关键字：词法分析，语法分析，类型检查，中间代码生成，目标代码生成

目录

一、 实验平台介绍

系统名称：ubuntu-22.04.2
操作系统类型：Linux 64 位
虚拟机：VMware Workstation Pro
编译器：gcc，llvm

二、 分工介绍

本次实验从整体架构上来说主要包含了 5 大部分：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。在语法分析中，本人主要完成了 CFG 的设计与 lexer.l 的编写，王耀完成了符号表的相关函数。在词法分析中，我组二人一齐设计了 CFG，本人主要完成了 parser.y 的编写，王耀完成了构建抽象语法树的相关函数。类型检查与中间代码生成为之前同一次作业所布置，本人主要完成了类型检查中变量与函数重定义与函数重载等相关工作，王耀主要完成了变量类型隐式转换与类型控制流的翻译，表达式翻译中我组二人均各有所编写。在目标代码生成部分中，本人主要完成了寄存器分配算法与 MachineCode.h/cpp 相关代码，王耀主要完成了 Instruction.h/cpp 中的相关代码，并对先前代码进行改进，使之能够适配大作业的框架。

三、 词法分析

(一) CFG 表述 SysY 语言特性

上下文无关文法是一种用于描述程序设计语言语法的表示方式。一般来说，一个上下文无关文法（context-free grammar）由四个元素组成：一个终结符号集合 V_T，一个非终结符号集合 V_N，一个产生式集合 P，和指定的一个非终结符号为开始符号 S。

因此，上下文无关文法可以通过 (VT ,VN ,P,S) 这个四元式定义。在本次实验中将利用上下文无关文法来描述编译器支持的 SysY 特性。具体来说，在描述文法时，我们将数位、符号和黑体字符串看作终结符号，将斜体字符串看作非终结符号，以同一个非终结符号为头部的多个产生式的右部可以放在一起表示，不同的右部之间用符号 | 分隔。

1. 终结符集合 V_T

一个终结符号集合 V_T，它们有时也称为“词法单元”。终结符号是该文法所定义的语言的基本符号的集合，如数字、运算符、黑体字符串。

1、标识符 Ident 的定义

此处定义了所有的常量类型即非数值型标识符（_ 和大小写字母）、数值型标识符以及它们的结合。

```
Ident    -> Nondigit
          | Ident Nondigit
          | Ident Digit
Nondigit -> '_' | 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
Digit    -> '0' | '1' | ... | '9'
```

2、整数

此处定义了整型终结符，可支持识别的进制即 10 进制、8 进制和 16 进制。其中定义了 8 进制的前缀为 0，16 进制的前缀为 0x 或 0X。

```
integer-const      -> decimal-const | octal-const | hexadecimal-const
decimal-const      -> nonzero-digit | decimal-const digit
octal-const        -> '0' | octal-const octal-digit
hexadecimal-const  -> hexadecimal-prefix hexadecimal-digit
                      | hexadecimal-const hexadecimal-digit
hexadecimal-prefix -> '0x' | '0X'
nonzero-digit      -> '1' | '2' | ... | '9'
octal-digit        -> '0' | '1' | ... | '7'
digit              -> '0' | nonzero-digit
hexadecimal-digit  -> '0' | '1' | ... | '9'
                      | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
                      | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
```

3、浮点数

此处定义了浮点数型的终结符，在整数型的基础上添加小数点（dot），在程序中出现的的浮点数都被视为十进制的浮点数，且程序中仅允许存在十进制的浮点数。浮点数格式为：数字序列‘.’数字序列

```
float -> number '.' number
number -> digit | digit number
```

4、运算符语句中的运算符也是终结符的一部分，初步需要支持识别基本的算术运算、关系运算和逻辑运算。具体如下：算术运算：+、-、*、/、%

关系运算：==，>，<，>=，<=，!=

逻辑运算：&&、||、!

运算符的优先级将切入在后续产生式的表达式的设计中，如 AddExp -> MulExp | AddExp ('+' | '−') MulExp，中 MulExp 可以 AddExp 推导，确保了乘除模运算优先于加减运算。

5、关键字

编程中的关键字也叫保留字，是被编程语言赋予特殊含义的单词或标识符，这也属于终结符。

基本关键字：void、int、const、if、else、while、break、continue、return

6、其他

这部分列举了在编程语言中一些比较特殊的终结符，它与语言的格式有关。

基本符号：{ 、}、[、] 、(、) 、; 、//、/*、*/、,

2. 非终结符集合 V_N

非终结符即一些语法变量，是一种中间形式，一般使用斜体字符串表示，以同一个非终结符号为头部的多个产生式的右部可以放在一起表示，不同的右部之间用符号 | 分隔。除终结符外其他均为非终结符。

3. 开始符号 S

开始符号是一个特定的非终结符，可以说它是进行翻译的开始。在这里定义开始符号为：CompUnit。

(二) 符号表

符号表的作用主要包含一下几点：

标识符的声明和定义：符号表记录了标识符的名称、类型、作用域等信息，可以帮助编译器验证标识符的正确性和一致性，例如检查重复定义、类型错误等。

符号的引用和解析：在语法分析阶段，编译器需要识别标识符的引用，并根据其在符号表中的信息进行解析。符号表可以帮助编译器确定标识符的属性、类型和定义位置等，以便生成正确的中间代码或目标代码。

作用域管理：符号表可以记录标识符的作用域信息，帮助编译器进行作用域的管理和解析。通过符号表，编译器可以检查变量的作用域是否合法，以及在不同作用域中标识符的可见性和访问权限等。

下面展示符号表，和符号表项基类的定义，其子类如 ConstantSymbolEntry、TemporarySymbolEntry 均为在 SymbolEntry 基础上派生而得。

符号表类的定义

```
class SymbolTable
{
private:
    std::map<std::string, SymbolEntry*> symbolTable;
    SymbolTable *prev;
    int level;
    static int counter;
public:
    SymbolTable();
    SymbolTable(SymbolTable *prev);
    void install(std::string name, SymbolEntry* entry);
    SymbolEntry* lookup(std::string name);
    SymbolTable* getPrev() {return prev;};
    int getLevel() {return level;};
    static int getLabel() {return counter++;};
};

 class SymbolEntry
{
private:
    int kind;
protected:
    enum {CONSTANT, VARIABLE, TEMPORARY};
    Type *type;

public:
    SymbolEntry(Type *type, int kind);
    virtual ~SymbolEntry() {};
    bool isConstant() const {return kind == CONSTANT;};
    bool isTemporary() const {return kind == TEMPORARY;};
    bool isVariable() const {return kind == VARIABLE;};
    Type* getType() {return type;};
    void setType(Type *type) {this->type = type;};
    virtual std::string toStr() = 0;
};
```

四、 语法分析

(一) 上下文无关文法的设计

为了产生式的简洁性和可读性，做出如下约定：

终结符:单引号括起的串，或者是 Ident 这样的
记号。
[...] ：选项:最多出现一次
{...} ：重复项：任意次数，包括 0 次
(...) ：分组
| ：并列选项，只能选一个

一个 SysY 程序由单个文件组成，文件内容对应 EBNF 表示中的 CompUnit。CompUnit由零个或多个的声明与零个或多个函数定义所组成，但必须存在且仅存在一个标识为 ‘main’ 、无参数、返回类型为 int 的 FuncDef：

```
CompUnit     -> [CompUnit] (Decl | FuncDef)
```

SysY 语言支持 int/float 类型和元素为 int/float 类型且按行优先存储的多维数组类型：

```
BType -> ’ int’  |  ‘float’
```

参数的类型可以是 int/float 或者数组类型；函数可以返回 int/float 类型的值，或者不返回值(即声明为 void 类型)。类型：

```
FuncType -> 'void' |'int'
```

声明分为常量声明与变量声明：

```
Decl         -> ConstDecl | VarDecl
```

常量声明，形式如“const int a=1” 或“const int a = 100, b = 0;” ，依次包含关键字const，常量类型 (BType) ，一条或多条常量定义 (ConstDef) 语句：

```
ConstDecl    -> 'const' BType ConstDef { ',' ConstDef } ';'
```

常量定义ConstDef 用于定义符号常量。ConstDef 中的 Ident 为常量的标识符，在 Ident后、 ‘=’ 之前是可选的数组维度 ('[' ConstExp ']') 和各维长度的定义部分，在 ‘=’ 之后是初始值 (ConstInitVal) ：

```
ConstDef -> Ident { '[' ConstExp ']' } ’ =’  ConstInitVal
```

ConstDef 的数组维度和各维长度的定义部分不存在时，表示定义单个变量。此时 ‘=’ 右边必须是单个初始数值。当 ConstDef 定义的是数组时， ‘=’ 右边的 ConstInitVal 表示常量初始化器：

```
ConstInitVal -> ConstExp
            | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
```

变量声明，形如“int a;” 、“int a=2,b;” ，依次包含着变量类型 (BType) ，

一个或多个变量定义语句（VarDef）：

```
VarDecl      -> BType VarDef { ',' VarDef } ';'
```

VarDef 用于定义变量。分为两种情况，当不含有‘=’时，其运行时实际初值未定义，当变量名（Ident）后存在‘=’时，将在‘=’后接变量的初始值（InitVal）：

```
VarDef       -> Ident { '[' ConstExp ']' }
              | Ident { '[' ConstExp ']' } '=' InitVal
```

变量初始化，变量的初始值可以使用表达式（Exp）进行赋值，在对数组进行赋初值时，使用‘{’、‘}’包含对应数组元素个数的表达式：

```
InitVal      -> Exp
              | '{' [ InitVal { ',' InitVal } ] '}'
```

函数声明包含了函数的返回值类型（FuncType），函数名（Ident），一对小括号与包括在其中的也可以不存在的函数形参表（FuncFParams），以及函数的定义语句块（Block）：

```
FuncDef      -> FuncType Ident '(' [FuncFParams] ')' Block
```

函数形参表，可以包含单个参数（FuncFParam）或者多个参数：

```
FuncFParams  -> FuncFParam { ',' FuncFParam }
```

对于每个参数，需要写明它的类型（BType），如果是数组的话会包含一对或多对方括号‘[]’，在多维数组的情况下需要标明非最高维的维数：

```
FuncFParam   -> BType Ident ['[' ']' { '[' Exp ']' }]
```

语句块，是一段用‘{}’包括的任意数量语句块项（BlockItem）

```
Block        -> '{' { BlockItem } '}'
```

语句块项包括了声明（Decl）与语句（Stmt）：

```
BlockItem    -> Decl | Stmt
```

声明的文法与上文所述相同，语句的语法如下，包括对左值表达式的赋值，算数运算表达式，空语句，语句块，if条件语句，while循环语句，break与continue循环控制语句，return返回语句：

```
Stmt         -> LVal '=' Exp ';'
              | [Exp] ';'
              | Block
              | 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
              | 'while' '(' Cond ')' Stmt
              | 'break' ';'
              | 'continue' ';'
              | 'return' [Exp] ';'
```

使用上下文无关文法描述编译器中所预期的表达式：

算术运算表达式：

```
Exp          -> AddExp (+法进行连接, SysY 表达式是 int型)
条件表达式:
Cond         -> LOrExp (逻辑运算表达式, 析取范式)
左值表达式:
LVal         -> Ident {'[' Exp ']'} ([i+1])
基本表达式:
PrimaryExp   -> '(' Exp ')' | LVal | Number
数值:
Number   -> IntConst
一元表达式:
UnaryExp     -> PrimaryExp
             | Ident '(' [FuncRParams] ')'
             | UnaryOp UnaryExp
单目运算符:
UnaryOp      -> '+' | '-' | '!'  // 需要保证 '!' 仅出现在 Cond 中
函数实参表:
FuncRParams  -> Exp { ',' Exp }
乘除模表达式:
MulExp       -> UnaryExp
             | MulExp ('*' | '/' | '%') UnaryExp
加减 (算术) 表达式:
AddExp       -> MulExp
             | AddExp ('+' | '-') MulExp
关系表达式:
RelExp       -> AddExp
             | RelExp ('<' | '>' | '<=' | '>=') AddExp
相等性表达式:
EqExp        -> RelExp
             | EqExp ('==' | '!=') RelExp
逻辑与表达式:
LAndExp      -> EqExp
             | LAndExp '&&' EqExp
逻辑或表达式 (析取表达式) :
LOrExp       -> LAndExp
             | LOrExp '||' LAndExp
常量表达式:
ConstExp → AddExp // 在语义上额外约束这里的 AddExp 必须是一个可以在编译期求出值的常量
```

(二) 构造语法树

构造语法树的操作主要在 parser.y 文件中完成。通过孩子节点的属性值构造父节点，并设置继承关系，具体操作通过调用对应语句的构造函数并传入对应参数得到，构造抽象语法树的相关函数位于 Ast.h/cpp 中，包含了对于节点基类（Node）的定义与派生得到各种不同节点类，相关构造示例代码如下:

构造语法树示例代码

```
//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);
        }
        $$ = new Id(se);
        delete []$1;
    }
    ;
AssignStmt
    :
    LVal ASSIGN Exp SEMICOLON {
        $$ = new AssignStmt($1, $3);
    }
    ;
//Ast.h-------------------------------------------
class Node
{
private:
    static int counter;
    int seq;
protected:
    std::vector<Instruction *> true_list;
    std::vector<Instruction *> false_list;
    static IRBuilder *builder;
    void backPatch(std::vector<Instruction *> &list, BasicBlock*target,bool
        branch);
    std::vector<Instruction *> merge(std::vector<Instruction *> &list1, std::
        vector<Instruction *> &list2);

public:
    Node();
    int getSeq() const {return seq;};
    static void setIRBuilder(IRBuilder*ib) {builder = ib;};
    virtual void output(int level) = 0;
    virtual void typeCheck() = 0;
    virtual void genCode() = 0;
    std::vector<Instruction *>& trueList() {return true_list;}
    std::vector<Instruction *>& falseList() {return false_list;}

    // new
```

```
46      Node *next;
47      // 添加next结点
48      void setNext(Node *node);
49      // 获取当前结点的next结点
50      Node *getNext();
51  };
```

五、 类型检查

基础的类型检查，首先通过在 Ast.h/cpp 中添加相关的 type_check () 函数实现。函数返回值的检查即通过此实现。

函数返回值类型检查示例代码

```
1   void FunctionDef::typeCheck()
2   {
3       // Todo
4       // 获取返回值类型
5       Type *retType = ((FunctionType *)(se->getType()))->getRetType();
6       // 调用checkRet函数，归到ReturnStmt类主要实现
7       if (!stmt->checkRet(retType))
8       {
9           fprintf(stderr, "function has no return statement function type doesn
                't match return type\n");
10          assert(stmt->checkRet(retType));
11      }
12  }
```

(一) 类型转换

以 bool 转 int 的类型转换为例，代码如下，通过判断操作数是否为 bool，若是则需要进行类型转换，具体操作为，新建临时变量将其添加进当前符号表，将指向操作数的指针重新指向新建的临时变量，将操作数高位用 0 填充后赋值给临时变量，在此之后使用临时变量进行相关运算。

类型转换示例代码

```
1   // 考虑if(op||op),第一个表达式为算术表达式
2           if (!test->isCond() && !test->isUncond())
3           {
4                   int opcode = CmpInstruction::NE;
5               Operand *src1 = expr1->getOperand();
6               Operand *src2 = src0_const0;
7               // dst比较指令目的寄存器
8               SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                    SymbolTable::getLabel());
9               Operand *dst = new Operand(tse);
10              Operand *n1 = src1;
11              // bool转int的类型转换
```

```
            if (src1->getType() == TypeSystem::boolType)
            {
                SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType
                    , SymbolTable::getLabel());
                n1 = new Operand(s);
                new ZextInstruction(n1, src1, bb); // 0拓展
            }
            // 跟0比较，判断表达式真假
            new CmpInstruction(opcode, dst, n1, src2, bb);
            Instruction *temp = new CondBrInstruction(nullptr, nullptr, dst,
                bb);
            // trueList和falseList均为目标地址未确定的跳转指令的向量
            expr1->trueList().push_back(temp);
            expr1->falseList().push_back(temp);
        }
```

(二) 变量重定义检查

变量重定义检查主要借助 lookup () 函数实现，lookup () 函数的效果为，在当前符号表中按名称查找目标变量，若查找到了，返回相应的符号表项，若未找到则通过 prev 查找前一个符号表，若直至 global 都为查找到，则会返回 nullptr，表示变量尚未定义。

lookup 函数实现

```
//parser.y中变量重复声明检查-----------------------------
VarDef
    : ID {
        if(identifiers->lookup($1))
        {   //检查变量是否重复声明
            //在当前作用域能够找到这个标识符名字，表示重复定义
            fprintf(stderr, "identifier \"%s\" is defined twice\n", (char*)$1
                );
            assert(identifiers->lookup($1)==nullptr);
        }
        //新创建标识符
        SymbolEntry* se;
        se = new IdentifierSymbolEntry(TypeSystem::intType, $1, identifiers->
            getLevel());
        identifiers->install($1, se);
        $$ = new DeclStmt(new Id(se));
        delete []$1;
    }
    | ID ASSIGN Exp {
        //检查变量是否重复声明
        if(identifiers->lookup($1))
        {
            //在当前作用域能够找到这个标识符名字，表示重复定义
            fprintf(stderr, "identifier \"%s\" is defined twice\n", (char*)$1
                );
```

```
            assert(identifiers->lookup($1)==nullptr);
        }
        SymbolEntry* se;
        se = new IdentifierSymbolEntry(TypeSystem::intType, $1, identifiers->
            getLevel());
        identifiers->install($1, se);
        $$ = new DeclStmt(new Id(se), $3);
        delete []$1;
    }
    ;
//SymbolTable.cpp中lookup () 函数定义--------------------------
SymbolEntry* SymbolTable::lookup(std::string name)
{
    // Todo
    // the stack contains symbol entries in the current scope
    SymbolTable *current = identifiers;
    while (current != nullptr)
        // symbolTable为map类型的成员变量
        if (current->symbolTable.find(name) != current->symbolTable.end())
            return current->symbolTable[name];
        else
            // 向下一个SymbolTable去找
            current = current->prev;
    return nullptr;
}
```

(三) 函数重定义与重载

函数重定义与重载同样使用 lookup () 函数实现，通过先判断函数名，在判断函数参数符号表的方式，判断函数是否已被定义，若并未存在相同函数，则会将新的函数符号表链接到已有的相同函数名的符号表之后。

函数重定义与重载

```
//parser.y中函数重定义与重载检查-----------------------------------
FuncDef // 函数定义
    : // new
    Type ID {
        identifiers = new SymbolTable(identifiers);//开辟新的符号表
    }
    LPAREN PerhapsFuncDefParams RPAREN {
        Type *funcType;
        std::vector<Type*> vec;
        std::vector<SymbolEntry*> vec1;
        DeclStmt* temp = (DeclStmt*)$5;
        // 获取参数类型
        while(temp){
            vec.push_back(temp->getId()->getSymPtr()->getType());
            vec1.push_back(temp->getId()->getSymPtr());
```

```
            temp = (DeclStmt*)(temp->getNext());
        }
        funcType = new FunctionType($1, vec,vec1);
        SymbolEntry* se = new IdentifierSymbolEntry(funcType, $2, identifiers
            ->getPrev()->getLevel());
        identifiers->getPrev()->install($2, se);// 函数在前一个符号表中
    }
    ......
    FuncDefParam
    : Type ID {
        if(identifiers->lookup($2))
        {
            //在当前作用域能够找到这个标识符名字, 表示重复定义
            fprintf(stderr, "function parameter \"%s\" is defined twice\n", (
                char*)$2);
            assert(identifiers->lookup($2)==nullptr);
        }
        SymbolEntry* se;
        se = new IdentifierSymbolEntry($1, $2, identifiers->getLevel());
        identifiers->install($2, se);
        $$ = new DeclStmt(new Id(se));
        delete []$2;
    }
    | Type ID ASSIGN Exp {
        if(identifiers->lookup($2))
        {
            //在当前作用域能够找到这个标识符名字, 表示重复定义
            fprintf(stderr, "function parameter \"%s\" is defined twice\n", (
                char*)$2);
            assert(identifiers->lookup($2)==nullptr);
        }
        SymbolEntry *se;
        se = new IdentifierSymbolEntry($1, $2, identifiers->getLevel());
        identifiers->install($2, se);
        $$ = new DeclStmt(new Id(se),$4);
        delete []$2;
    }
//SymbolTable.cpp中install () 函数定义-----------------------------
void SymbolTable::install(std::string name, SymbolEntry* entry)
{
    //symbolTable[name] = entry;
// 同时检查是否有函数的重定义,如果有同名的函数, 链入符号表项
    if (symbolTable.find(name) != symbolTable.end() && symbolTable[name]->
        getType()->isFunc())
    {
        symbolTable[name]->setNext(entry);
    }
    else
```

```
60      {
61          symbolTable[name] = entry;
62      }
63  }
```

六、 中间代码生成

(一) 控制流的翻译

控制流的翻译使用了回填技术，首先编译器通过跳转语句将程序命令划分为基本块（BasicBlock），之后会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。等到翻译其祖先结点能确定这些目标基本块时进行回填，示例代码如下：

构造流图

```
1   // 根据基本块的前驱、后继关系构造流图
2   for (auto block = func->begin(); block != func->end(); block++)
3   {
4       // 获取该块的第一条和最后一条指令
5       Instruction *i = (*block)->begin();
6       Instruction *last = (*block)->rbegin();
7       while (i != last)
8       {
9           if (i->isCond() || i->isUncond())
10          {
11              // 跳转指令的下一条指令不定,移除块中跳转指令
12              (*block)->remove(i);
13          }
14          i = i->getNext();
15      }
16      // 有条件跳转
17      if (last->isCond())
18      {
19          BasicBlock *truebranch, *falsebranch;
20          truebranch = dynamic_cast<CondBrInstruction *>(last)->
                getTrueBranch();
21          falsebranch = dynamic_cast<CondBrInstruction *>(last)->
                getFalseBranch();
22          if (truebranch->empty())
23          {
24              new RetInstruction(nullptr, truebranch);
25          }
26          else if (falsebranch->empty())
27          {
28              new RetInstruction(nullptr, falsebranch);
29          }
30          // 添加后继
31          (*block)->addSucc(truebranch);
```

```
            (*block)->addSucc(falsebranch);
            // 添加前驱
            truebranch->addPred(*block);
            falsebranch->addPred(*block);
        }
        // 无条件跳转
        else if (last->isUncond())
        {
            // 获取要跳转的基本块
            BasicBlock *dst = dynamic_cast<UncondBrInstruction *>(last)->
                getBranch();
            // 跳转块为后继
            (*block)->addSucc(dst);
            dst->addPred(*block);

            if (dst->empty())
            {
                if (((FunctionType *)(se->getType()))->getRetType() ==
                    TypeSystem::intType)
                    new RetInstruction(new Operand(new ConstantSymbolEntry(
                                                TypeSystem::intType, 0)),
                                       dst);
                else if (((FunctionType *)(se->getType()))->getRetType() ==
                         TypeSystem::voidType)
                    new RetInstruction(nullptr, dst);
            }
        }
        // 最后一条语句不是返回以及跳转
        else if (!last->isRet())
        {
            if (((FunctionType *)(se->getType()))->getRetType() ==
                TypeSystem::voidType)
            {
                new RetInstruction(nullptr, *block);
            }
        }
    }
```

(二) 表达式翻译

通过 Ast.h/cpp 中的相关 GenCode 代码，设置指令的参数，如所属基本块，op，目的操作数，源操作数等，再通过 Instruction.h/cpp 中的相关语句的构造函数，构造出对应类型指令类的实例，在通过调用 output () 函数，编译器输出源程序所等价的 LLVM IR 命令。示例代码如下:

表达式翻译

```
CmpInstruction::CmpInstruction(unsigned opcode, Operand *dst, Operand *src1,
    Operand *src2, BasicBlock *insert_bb): Instruction(CMP, insert_bb){
    this->opcode = opcode;
```

```
    operands.push_back(dst);
    operands.push_back(src1);
    operands.push_back(src2);
    dst->setDef(this);
    src1->addUse(this);
    src2->addUse(this);
    // new
    dst->getSymbolEntry()->setType(TypeSystem::boolType);
}

void CmpInstruction::output() const{
    std::string s1, s2, s3, op, type;
    s1 = operands[0]->toStr();
    s2 = operands[1]->toStr();
    s3 = operands[2]->toStr();
    type = operands[1]->getType()->toStr();
    switch (opcode)
    {
    case E:
        op = "eq";
        break;
    case NE:
        op = "ne";
        break;
    case L:
        op = "slt";
        break;
    case LE:
        op = "sle";
        break;
    case G:
        op = "sgt";
        break;
    case GE:
        op = "sge";
        break;
    default:
        op = "";
        break;
    }

    fprintf(yyout, "  %s = icmp %s %s %s, %s\n", s1.c_str(), op.c_str(), type
        .c_str(), s2.c_str(), s3.c_str());
}
```

七、 目标代码生成

(一) 目标代码生成

在原先的 Instruction.h/cpp 的基础上，对各个类补充 genMachineCode() 函数，它通过调用其它命令的 genMachineCode() 函数和 MachineCode.h/cpp 中定义的 xxxMInstruction 函数实现汇编代码结构的构造，最后调用对应的 output 函数，打印对应的汇编代码。以生成操作数的代码为例。

操作数目标代码生成

```
//Instruction.cpp---------------------------------------------------
 MachineOperand *Instruction::genMachineOperand(Operand *ope)
{
    auto se = ope->getEntry();
    MachineOperand *mope = nullptr;
    if (se->isConstant())
        mope = new MachineOperand(MachineOperand::IMM, dynamic_cast<
            ConstantSymbolEntry *>(se)->getValue());
    else if (se->isTemporary())
        mope = new MachineOperand(MachineOperand::VREG, dynamic_cast<
            TemporarySymbolEntry *>(se)->getLabel());
    else if (se->isVariable())
    {
        auto id_se = dynamic_cast<IdentifierSymbolEntry *>(se);
        if (id_se->isGlobal())
            mope = new MachineOperand(id_se->getName().c_str());
        else
            exit(0);
    }
    return mope;
}
//MachineCode.cpp---------------------------------------------------
MachineOperand::MachineOperand(int tp, int val)
{
    this->type = tp;
    if (tp == MachineOperand::IMM)
        this->val = val;
    else
        this->reg_no = val;
}

MachineOperand::MachineOperand(std::string label)
{
    this->type = MachineOperand::LABEL;
    this->label = label;
}

void MachineOperand::output()
```

```
{
    /* HINT: print operand
     * Example:
     * immediate num 1 -> print #1;
     * register 1 -> print r1;
     * lable addr_a -> print addr_a; */
    switch (this->type)
    {
    case IMM:
        fprintf(yyout, "#%d", this->val);
        break;
    case VREG:
        fprintf(yyout, "v%d", this->reg_no);
        break;
    case REG:
        PrintReg();
        break;
    case LABEL:
        if (this->label.substr(0, 2) == ".L" || isfunc)
            fprintf(yyout, "%s", this->label.c_str());
        else
            fprintf(yyout, "addr_%s", this->label.c_str());
    default:
        break;
    }
}
```

(二) 寄存器分配

采用线性扫描分配寄存器的方法。首先遍历 unhandled 列表中的 interval，依次为其分配寄存器。首先，一次遍历 active 列表，判断在当前 interval 的开始位置是否存在已经结束了的 Interval，如果有，则将其逐出，将分配给它的寄存器回收，用于后续的分配。然后，判断当前可用的寄存器数目是否为 0，如果没有的话，执行 splitAtInterval(i) 方法，从 active 列表的最后一个 interval 和当前 interval 选择一个 interval，将其溢出到 stack slot 中，选择的方法就是看谁的结束位置更迟，该场景下也就是谁的结束位置更大。如果是当前 interval 被逐出，只需要为其分配一个 stack slot 即可；否则把分配给 active 列表中的 Interval 的寄存器分配给 current interval，并溢出其至 stack slot。其核心代码如下：

线性扫描分配寄存器

```
bool LinearScan::linearScanRegisterAllocation()
{
    bool success = true;
    active.clear();
    regs.clear();
    for (int i = 4; i < 11; i++)
        regs.push_back(i);
    for (auto &i : intervals)
```

```
    {
        expireOldIntervals(i);
        if (regs.empty())
        {
            spillAtInterval(i);
            success = false;
        }
        else
        {
            i->rreg = regs.front();
            regs.erase(regs.begin());
            insertToActive(i);
        }
    }
    return success;
}
```

八、 总结

(一) 总结

为期一个学期的编译原理实验终于结束了。通过这个学期的理论学习和实验探究，我对编译原理的算法理论有了清晰的认识，并通过工程实验深入理解了编译器的前后端架构以及各模块之间的工作流程和实现细节。同时，在实验过程中，我还对 ARM 汇编和程序执行过程的底层原理进行了全面而深刻的学习。在这一个学期的努力下，随着上机实验的一步步的引导，我们成功地实现了 SysY 语言的编译器。通过这个学期的实验经验与教训，我们对工程开发的设计与编写思路的理解更上一层楼。

最后，我要感谢王刚老师在这个学期里尽职尽责地进行理论讲解，以及助教们的耐心解答，使我们能够顺利完成我们的编译器。

(二) 项目仓库链接

https://gitlab.eduxiji.net/nku2023-compiler3/final.git