南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理期末实验报告

编译器的实现

林语盈

年级：2020级

专业：计算机科学与技术

学号：2012174

2023 年 1 月 11 日

摘要

在本次实验中,实现了从SysY语言到ARM汇编语言的编译器。经历了词法分析、语法分析、语法树的构建、类型检查、中间代码生成、汇编语言生成以及寄存器分配的步骤。本次报告中，将依次介绍各部分的具体实现，着重介绍本人在实验中实现的部分。代码链接：代码仓库链接

关键字：编译器；SysY语言；LLVM IR； ARM汇编

目录

一、 引言

本学期，实现了编译器，输入sysY语言代码，输出asm汇编语言。具体来说，实现了以下功能：

1. 数据类型: int
2. 变量声明、常量声明，常量、变量的初始化
3. 语句: 赋值(=)、表达式语句、语句块、if、while、return
4. 表达式:算术运算(+、-、、%，其中+、-都可以是单目运算符)、关系运算 (==，¿,¡,¿=,¡=，!=) 和逻辑运算 (&&(与)、——(或)、!(非))
5. 注释
6. 输入输出(实现连接SysY运行时库，参见文档《SysY运行时库》)
7. 函数：函数声明、函数调用；
8. 变量、常量作用域：在函数中、语句块(嵌套) 中包含变量、常量声明的处理，break、continue语句
9. 数组 (一维、二维、··) 的声明和数组元素访问

关于团队协作，在实验中，很多部分的任务难以分开，二人完成的难度要远远高于单独完成，在实验的过程中也渐渐学习团队合作。实验中的很多部分都为二人商量协作完成，也有很多模块二人都重复写了同一部分，最后商讨仅取一份或进行合并，此外，每次作业之间的代码整合、对新框架的理解也是不可或缺的工作，这里就不再体现。以下介绍我涉及的部分，有些代码太过久远，实在记不清最后采用谁的版本，也一并写上。

二、 代码整体框架

（一） 词法分析

首先，由输入的 sysY 代码进行词法分析，输出一个个识别到的单词，这一部分借助lex实现，主要代码均在 lexer.l 中。同时，构建符号表，记录识别到的id和其相关信息等，这一部分借助 SymbolTable 类实现。

（二） 语法分析与语法树构建

接下来，进行语法分析，根据拆分的词识别语法结构，并构造语法树，语法分析部分借助yacc实现，主要在 parser.y 文件中，构造不同的语句产生式。实现类型系统，主要借助 Type 类进行实现。完善符号表的实现。根据不同的节点类型，构建起语法树，不同结点的定义与结点的函数方法在 Ast.h 与 Ast.cpp 文件中实现,主要夫类为 Node，其又分为声明结点、表达式结点等多种类型，每类又继续细分，对应了不同的语句类型。

（三） 类型检查

然后，实现了类型检查。为了检查重复声明等情况，希望在符号表建立之前进行检查并及时报错，所以在实现中，将类型检查的代码嵌入在了语法分析 parser.y 文件中，而不是 Ast 中 Node 的 typeCheck 方法中。

（四） 中间代码生成

实现了中间代码生成。首先，遍历语法树，调用 genCode 函数，生成中间代码的 unit 结构。然后，根据构建好的代码 Unit，遍历一遍并调用 Unit 的 output 函数（并不断向下调用至 instruction 的 output ），生成LLVM IR中间代码。主要的实现在 Ast 中 Node 的 genCode 方法中。其结构如下：

1. 程序文件描述

整个中间代码结构由 unit 类定义。实际 unit 的实例化在main.cpp 中。为了辅助构建 unit，在 IRBuilder.h 里面，包括了unit都指针，这里还定义了当前需要增加到的语句块的指针，每次调用 genCode 函数的时候，一般会先取出这个语句块的指针，并在其上继续增加后续指令。在unit中存储了所有函数的列表，在output时会遍历每一个函数，并依次调用每个函数的output，进行中间代码的输出。

每个 function 是一个函数的调用。其中存储了 BasicBlock 列表，组织形式为数组。另外，还存储了进入函数的语句块的指针和函数返回的语句块的指针。以及函数返回值所存储的符号表表项等。

每个 BasicBlock 是不含分支跳转指令的连续执行的语句块。一个 BasicBlock 由多条语句组成，其组织形式是链表。此外，还记录了用于控制流图的前续语句块和后续语句块的指针列表，以及其上的函数的指针等。在 BasicBlock 中，实现了 insertBefore 函数，即将输入的dst指令插入到当前块中的srd指令之前。

不同语句的定义在 Instruction 类里面。对于不同的语句会生成不同的中间代码，由不同的对应指令的子类进行处理，它们都是 instruction 的子类。其中比较重要的函数包括：构造函数，用于构建当前指令；output函数，用于生成中间代码； genMachineCode 函数用于通过中间代码结构生成汇编代码结构。

在 Operand 类中，定义了操作数，在不同 Instruction 类、Function 中，会按情况用到操作数，Operand 中记录了用到该操作数的指令。

2. 控制流图

Function中的每一个 BasicBlock 不是按顺序执行的，而是根据实际的分支跳转构建控制流图，在函数声明 FunctionDef 中，生成内部代码之后，即构建好流图。控制流图中每一个节点是一个 BasicBlock ，每个 BasicBlock 中记录了他的入边和出边，分别用数组表示，声明在 BasicBlock.h 的 succ 与 pred 中。当整个遍历回到函数块的构建时，遍历这个函数中每一个块，再遍历这个块中的最后一个指令（分支跳转指令），将指令后面的 true branch 和 false branch 指示的块与本块的succ与pred连接好，这就构建了控制流图。建立好块的控制流图后，每一个块中的指令按顺序执行，这样就可以遍历控制流图调用 output ，输出代码。此外在函数之间，会根据函数中记录的ret bb，构建好函数间的调用关系。

3. 回填技术

在构建块间控制关系的时候，使用了回填技术，对于树中的每一个节点，记录一个 truelist 和一个 falselist ，List 中为跳转指令的集合，跳转指令要跳转到的 block 尚未确定。 truelist 代表，这个节点所代表的语句为true的时候其下一个 block 就是跳转指令需要跳转到的 block ，flaselist 类似，即只有只有表达式节点（条件）的 truelist 和 falselist 是有意义的。每次调用会重新整合 truelist 和 falselist ，通过merge函数。在调用回到上一级可以确定跳转位置时，调用backPatch进行回填。表达式节点的处理和最终一定会在 if 或者 while 等语句中被回填。回填

即在每一个跳转 instruction 中记录这条指令的后续块（回填 node 的 truelist 与 falselist ）的 truebranch 和 falsebranch 。其中，指令可以分成条件指令和非条件指令。条件指令包括两个后续的 block 分支，非条件指令包括一个后续的block分支。所以，每次生成新的块时候，都必须要在最后一条指定建立跳转指令，但是可以先不填写跳转指定的跳转位置，而是把它加入本节点的 trulist 和 falselist ，等待回填。

（五） 汇编语言生成

目标代码生成的步骤是：首先，调用 unit 的 genMachineCode 函数，调用至每个 instruction 的 genMachineCode 函数（位于 instruction.cpp 中），生成目标代码，保存在 AsmBuilder 中。然后，使用 LinearScan 类进行寄存器分配。最后，调用 munit 的 output 的函数，输出生成的目标代码。

目标代码生成的程序结构与中间代码生成类似，分为 MachineUnit 、 MachineFunction 、MachineBlock 、各类MachineInstruction 、MachineOperand 。（在 MachineCode.h 中定义）。AsmBuilder 中存有 mUnit 指针，指向 main.cpp 的 mUnit ，该类用于辅助生成 mUnit 结构，与 IRBuilder 类似。此时，为所有临时变量分配了一个虚拟寄存器 VREG （认为是无穷的）。

寄存器的分配。在构建好 mUnit 后，借助 LinearScan 实现线性扫描寄存器分配算法，单趟遍历每个活跃区间 (Interval)，为其分配物理寄存器。首先，进行活跃区间分析，框架已经完成，在 LiveVariableAnalysis 类中实现。然后，进行线性扫描寄存器分配，主要是 LinearScan 的 linearScanRegisterAllocation 函数。最后，如果有临时变量被溢出到栈中，还要生成溢出代码，主要是 LinearScan 的 spillAtInterval 函数。迭代进行以上过程，重新活跃变量分析，进行寄存器分配，直至没有溢出情况出现。

至此，则完成了整个编译器的构造，输入的sysy代码，输出ARM汇编语言的代码。

三、 词法分析模块

编写 lexer.l , 程序主要部分为头部（定义输出函数）、正则定义、规则定义几部分。并获取行号和列号。

（一） 基本词语的正则定义

基本词语的正则定义（例）

```
ADD "+"
SUB "-"
EOL (\r\n|\n|\r)
WHITE [\t ]
```

（二） 其他终结符的规则定义

包括一些符号和关键字，总体过程类似，但实际编写非常繁琐：int、bool、void、if、then、else、while、return、const、+、-、*、%、——、&&、¡、¿、==、!=、¿=、¡=、=、!、++、–、,、;、(、)、[、]、{、}、break、continue

（三） 八进制、十六进制数的定义

八进制、十六进制数的正则定义

```
OCTAL  0([1-7][0-7]*|0)
HEX  (0x|0X)([1-9a-fA-F][0-9a-fA-F]*|0)
```

以八进制为例，使用sscanf将其转换为十进制存储。

八进制的规则定义

```
{OCTAL} {
    int val;
    sscanf(yytext, "%o", &val);
    if(dump_tokens)
        DEBUG_FOR_LAB4(yytext);
    yylval.itype = val;
    return INTEGER;
}
```

（四） 标识符的定义

ID正则表达式

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

规则部分则直接将其作为字符串输出，作用范围则由语法部分进行处理。

（五） 注释的定义

行注释、块注释分别实现，声明部分使用%x 声明一个新的起始状态，而在之后的规则使用中加入状态名，表明该规则只在当前状态下生效。

注释正则定义

```
LINECOMMENT \/\/[^\n]*
commentbegin "/*"
commentelement .|\n
commentend "*/"
%x BLOCKCOMMENT
```

注释规则定义

```
{LINECOMMENT} {
  #ifdef ONLY_FOR_LEX
      DEBUG_FOR_LAB4("LINECOMMENT","0");
  #endif
}
{commentbegin} {
    #ifdef ONLY_FOR_LEX
        DEBUG_FOR_LAB4("commentbegin","0");
        BEGIN BLOCKCOMMENT;
```

```
    #endif
}
<BLOCKCOMMENT>{commentend} {
    #ifdef ONLY_FOR_LEX
        DEBUG_FOR_LAB4("commentend","0");
        BEGIN INITIAL;
    #endif
}
<BLOCKCOMMENT>{commentelement} {}
```

（六） 库函数ID的定义

实现了库函数（putint、getint、putch、getch等）的识别。函数名就是标识符（ID），将其插入到最外层即范围最大的符号表中。要注意的是，在将其插入符号表时，要确定其参数个数，并正确确定其类型（FunctionType）和作用域。具体详见代码即可。

库函数规则定义（putint例）

```
  "putint" {
    if(dump_tokens)
        DEBUG_FOR_LAB4(yytext);
    char *lexeme;
    lexeme = new char[strlen(yytext) + 1];
    strcpy(lexeme, yytext);
    yylval.strtype = lexeme;//字符串
    std::vector<Type*> paramTypes;
    paramTypes.push_back(TypeSystem::intType);//有一个参数int
    Type* funcType = new FunctionType(TypeSystem::voidType, paramTypes);//函数type
    SymbolTable* st = identifiers;
    while(st->getPrev())//找到符号表头
        st = st->getPrev();
    if(!st->lookup(yytext)){//插入
        SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, st->getLevel(), true);
        st->install(yytext, se);
    }
    return ID;
}
```

四、 语法分析模块

主要在 parser.y 中实现，由词法分析得到的词构建语法树。代码较长，就不再展示。为了便于理解，列出每个语法的规则定义，即其产生式。

语法分析模块的程序结构主要分为三个部分，第一部分是头部引用部分以及一些全局变量的声明，第二部分是对终结符和非终结符的定义，要注意的是对于不同的非终结符具有不同的

返回类型（stmttype或exprtype），使用%union定义，第三部分是产生式，也就是规则定义部分。整个程序开始符号为program，即stmts，stmts递归分为单句stmt。stmt分为各种类型，包括AssignStmt BlockStmt IfStmt WhileStmt ReturnStmt DeclStmt ExprStmt BreakStmt ContinueStmt FuncDef等。其中涉及表达式，按照优先级进行定义和实现，依次为Exp LOrExp LAndExp EqlExp RelExp AddExp MulExp preSinExp sufSinExp PrimaryExp ， PrimaryExp，又分为 LVal FuncCall等，在下文将进一步解释。

语法树的各个节点的定义和函数方法在Ast文件中实现，Node主要分为表达式节点和语句节点，每个节点根据实际语句类型，记录不同的所需信息。

产生式

```
Program
: Stmts
Stmts
    : Stmt
    | Stmts Stmt
    Stmt
    : AssignStmt
    | BlockStmt
    | IfStmt
    | WhileStmt
    | ReturnStmt
    | DeclStmt
    | FuncDef
    | ExprStmt
    | SEMICOLON
    | BreakStmt
    | ContinueStmt
    ;
```


（一） 符号表

符号表由 symboltable 类进行实现,在 symboltable.cpp 中定义了全局符号表 identifier .整个实现中的符号表类似一个树形的结构, 每一个节点是一个 symboltable ,对应着大括号组成的语句块的范围内的符号表。每一个符号表中，包括由 ID 字符串到符号表项的对应映射。实现了 lookup 函数，即根据根据所给字符串，在当前符号表中遍历并找到与字符串匹配的项，输出；如果找不到，则返回空指针。

符号表项由 symbolentry 类实现，其中包括这个符号的字符串和它的数据类型、对应的操作数地址等。如果是const类型，则另外在此记录其值，并且添加了对应获取和设置cosnt值的get和set的函数。在在标示符类型的符号表项中，额外增加了parent和sysy成员，具体作用将在后文说到。

（二） 类型系统

类型系统由type类进行实现，目前实现了INT, VOID, FUNC, ARRAY, PTR, CONST, BOOL类型。

（三） 变量、常量的声明和初始化

生成一个DeclStmt，由非终结符Type定义声明类型，接下来紧跟一个ID（或数组）或一个IDlist，或有赋初值。首先，应当根据类型、ID、当前位置创建新的符号表项，并插入符号表，若有初值还应赋好符号表的初值。然后，根据此符号表项生成DeclStmt结点，并连接到语法树中。

产生式

```
Type
    : INT| BOOL| VOID| CONST Type
IDList
    :
    ID COMMA
    |     ID ASSIGN Exp COMMA
    |     ARRAY COMMA
    |     ARRAY ASSIGN BRACEUnit COMMA
    |     IDList ID COMMA
    |     IDList ID ASSIGN Exp COMMA
    |     IDList ARRAY COMMA
    |     IDList ARRAY ASSIGN BRACEUnit COMMA
    |     IDList ID
    |     IDList ID ASSIGN Exp
    |     IDList ARRAY
    |     IDList ARRAY ASSIGN BRACEUnit
DeclStmt
    :
    Type ID SEMICOLON
    |     Type ID ASSIGN Exp SEMICOLON
    |     Type ARRAY SEMICOLON
    |     Type ARRAY ASSIGN BRACEUnit SEMICOLON
    |     Type IDList SEMICOLON
```


（四） 语句

1. 赋值语句

AssignStmt即AssignExpr与分号。AssignExpr即由一个左值、赋值号、表达式组成。找到符号表中的SymbolEntry，并由此建立新的AssignExpr结点。如果为const类型，应当在SymbolEntry中同时记录初值。

产生式

```
AssignStmt
    : AssignExpr SEMICOLON
AssignExpr
    : LVal ASSIGN Exp
```

2. 表达式语句

即仅由一个表达式构成的语句。

产生式

```
ExprStmt
:  Exp SEMICOLON
```

3. 空语句、语句块

空语句为了处理一个分号的情况而设置。语句块是一对单独大括号括起来的一些语句。如果大括号中是空的，生成一个空语句；如果其中还有其他语句，生成一个新的符号表，并建立CompoundStmt 节点，在其中的所有语句执行结束后，再将符号表更新成之前的。

产生式

```
BlockStmt
:  LBRACE Stmts RBRACE
|  LBRACE RBRACE
```

4. if、while

if分为if-then和if-else两种，while即有一种，节点中记录条件、后续分支。条件由Cond非终结符描述，即为exp。建立WhileStmt、IfStmt、IfElseStmt。

产生式

```
IfStmt
: IF LPAREN Cond RPAREN Stmt %prec THEN
| IF LPAREN Cond RPAREN Stmt ELSE Stmt
WhileStmt
: WHILE LPAREN Cond RPAREN Stmt
Cond
   : Exp
```

5. return

建立ReturnStmt。若存在返回值，返回值为Exp。

产生式

```
ReturnStmt
:  RETURN SEMICOLON
|  RETURN Exp SEMICOLON
```

（五） 表达式

建立BinaryExpr、preSingleExpr等结点，其均继承ExprNode。按照优先级进行定义和实现，依次为Exp LOrExp LAndExp EqlExp RelExp AddExp MulExp preSinExp sufSinExp PrimaryExp，分别为总表达式、逻辑或、逻辑与、相等或不等、大于小于、加减、乘除模、前缀运算符、后缀运算符、基本表达式。PrimaryExp为基本表达式，包括左值（ID或数组）、数字、括号表达式、函数调用。若左值为ID，则找到其符号表中的SymbolEntry，建立ID结点。对于数字，建立新的ConstantSymbolEntry，并建立Constant型结点。

产生式

```
Exp
    : LOrExp
LOrExp
    : LAndExp | LOrExp OR LAndExp
LAndExp
    : EqlExp | LAndExp AND EqlExp
EqlExp
    : RelExp | EqlExp EQUAL RelExp | EqlExp NOT_EQUAL RelExp
RelExp
    : AddExp | RelExp LESS AddExp | RelExp MORE AddExp | RelExp LESS_E
      AddExp | RelExp MORE_E AddExp
AddExp
    : MulExp | AddExp ADD MulExp | AddExp SUB MulExp
MulExp
    : preSinExp | MulExp MUL preSinExp | MulExp DIV preSinExp | MulExp MOD
      preSinExp
preSinExp
    : sufSinExp | AADD preSinExp | SSUB preSinExp | NOT preSinExp | ADD
      preSinExp | SUB preSinExp
sufSinExp
    : PrimaryExp| sufSinExp AADD | sufSinExp SSUB
PrimaryExp
    : LVal|INTEGER|LPAREN Exp RPAREN|FuncCall
LVal
    : ID| ARRAY
```

（六） 函数定义、函数参数

ParamDef为函数参数，包括一个或多个参数类型、参数ID（或数组）。函数参数使用全局变量paramtypes、paramsymbols临时记录当前的函数参数类型与参数名。

FuncDef进行函数定义，读入函数返回值类型、函数名、函数参数、函数体。在函数体之前，首先在当前的符号表中为函数建立符号表项，然后为函数中的作用域建立新的符号表，接下来，根据全局变量 paramtypes、paramsymbols 依次为每个函数参数在新的符号表中建立符号表项。在函数体之后，根据函数参数建立新的 FunctionDef 结点，并将 paramtypes、paramsymbols 清空以便下一次使用，将符号表更新为上一层的符号表。FunctionDef结点中，记录了返回值表项、函数参数表项列表、函数参数名、函数体语句。因为要保持函数参数的符号表表项在 FunctionDef 中的记录与符号表中的一致，使用全局变量 sesymlist 记录临时的函数参数符号表项列表，并在在函数体之后据此生成 FunctionDef。如果函数没有参数，则无需处理参数列表。

产生式

```
ParamDefs
   :   Type ID
   |   Type ARRAY
   |   ParamDefs COMMA Type ID
   |   ParamDefs COMMA Type ARRAY
FuncDef
```

```
    : Type ID LPAREN ParamDefs RPAREN BlockStmt
    | Type ID LPAREN RPAREN BlockStmt
```

（七） 函数调用

使用 Params 处理函数调用的函数参数。使用全局函数参数列表 globle_tmp_paramcalls 记录临时调用的函数参数，每次在Params中向其添加参数; 每次进行函数调用的时候，更新该列表（清空）。在遇到函数调用时，首先在符号表中找到该函数返回值的符号表表项，然后根据符号表表项和 globle_tmp_paramcalls 参数列表生成新的 FunctionCall 节点。如果该函数没有参数，同理。

产生式

```
Params
    : Exp
    | Params COMMA Exp
FuncCall
    : ID LPAREN Params RPAREN
    | ID LPAREN RPAREN
```

（八） break、continue

在遇到break和continue语句后，生成 BreakStmt 和ContinueStmt，并将其记录到全局数组变量 breakList 和 continueList 中。BreakStmt 和 ContinueStmt 中需要记录其所在的循环节点。在进入 while 语句前，将breakList 和 continueList 清空；在 while 语句中所有语句识别结束后，处理 breakList 与 continueList ，将其中所有语句与while节点连接好。

产生式

```
    BreakStmt
    :
    BREAK SEMICOLON
    ContinueStmt
        :
        CONTINUE SEMICOLON
```

五、 类型检查

将类型检查的代码嵌入在了语法分析 parser.y 文件中。

（一） 变量使用前未声明

在LVal中，首先判断 identifiers 是否存在该 ID，若不存在，报错。

（二） 同一作用域下重复声明

在 DeclStmt 中，如果当前符号表中已经存在，则报错。

（三） 条件判断表达式 int 至 bool 类型隐式转换

条件判断表达式中(涉及到and、or、相等、不等)，若类型为i32，则报错。

（四） 数值运算表达式运算数类型是否正确

可能导致此错误的情况有返回值为 void 的函数调用结果参与了某表达式计算。在 FuncCall 以及其上每个涉及到的表达式中、声明语句的初值设置中，若函数返回值为 void ，则将全局变量 alarm 设置为真。在 AssignExpr 中，若 alarm 为真，则代表void参与的表达式运算，则报错。

（五） 函数未声明，及不符合重载要求的函数重复声明

在函数定义时，如果当前函数名在符号表中已经存在且函数参数个数相同（当前只实现了int），则报错重复声明。

在函数调用时，检查函数是否声明，即符号表中是否有该函数名，若存在则报错重复声明。

（六） 函数调用时形参及实参类型或数目的不一致

在函数调用时，检查函数参数类型和数目是否符合，不符合则报错。

（七） return 语句操作数和函数声明的返回值类型是否匹配

全局变量 isret 默认为假（在函数定义读函数体之前初始化），在ReturnStmt中，若函数存在返回值，则将全局变量 isret 设置为true。在函数定义读函数体之后，比较声明的函数返回值与 isret，若返回类型不符合或者没有return语句，则报错。

六、 中间代码生成

整个中间代码由 unit 类定义。实际 unit 的实例化在 main.cpp 里面，具体结构在上文已经说过。这里主要的任务是完成Ast的各类 node 中的 genCode 函数，遍历语法树并递柜调用，根据语法树构建好代码框架 unit 。最后，遍历unit，调用unit的output（并不断调用至指令的output ），即可输出中间代码。代码过长，就不在展示。

（一） Operand

Operand 为操作数类，用于处理指令中用到的操作数。在操作数类中，定义了定义此操作数的指令，使用此操作数的指令序列和这个操作数符号表表项。

（二） 变量、常量的声明和初始化

DeclStmt::genCode 在原来基础上进行修改。首先由于一条声明语句可能同时声明了多个变量，所以在处理声明语句的时候，首先应该遍历此条语句中记录的变量列表，在依次进行处理。

如果该变量是全局的，除了分配好操作数外，还应该在global defs中加入全局指令 GlobalInstruction 即全局声明，并根据该句是否有赋值，进行赋值或赋初值零。global defs定义在unit中，在unit进行output时，会首先单独输出全局变量声明的语句，间unit.cpp。在Unit中，保存了全

局的global_dst与global_src，用于后续汇编代码的生成，所以此时还要将全局变量的dst和src存入这两个列表。

如果是局部变量，首先按原来的方式，在当前函数的栈中分配空间，然后，如果有赋初值，则需要额外增加store指令 StoreInstruction 。

接下来是数组的处理。

（三） 赋值语句

首先，生成所赋值表达式的语句，然后，在当前的语句块bb最后，根据符号表中的操作数地址和该节点表达式中的最后所得操作数地址，生成新的 store 指令StoreInstruction。对数组进行额外处理。

（四） 表达式

对于逻辑与和逻辑或需要实现短路，以逻辑与为例。

1. 首先，在原来的bb中生成第1个表达式的代码，然后重新获取当前的语句块bb，因为第1个表达式中可能调用其他而修改了当前bb。
2. 接下来，判断如果第一个表达式是整数类型，就需要bb中额外增加比较指令cmpinst，第四个参数表示与常数0进行比较，将整数类型变为布尔类型。
3. 接下来，bb中增加条件跳转指令让当前语句块与下一个语句块truebb连接，但是此时还不能确定下一个语句块是什么，需要先将这个条件分支指令加入到当前节点的falselist当中，在等待if或者while语句处理时进行回填。
4. 然后，当前表达是1的truelist已经可以回填，进行回填。
5. 如果表达式1正确，需要跳到trueBB，继续计算表达式2，后续语句在trueBB块中生成；如果不正确，后续还在当前块中生成。在整个语句中插入trueBB表达式块，接下来在这里面生成表达式2的语句。
6. 接下来，与expr1相同，根据其返回类型进行比较指令的添加。
7. 最后，整合当前的true list即为表达2的truelist；当前的false list，即为表达式一和表达式二falselist的合并。

逻辑或与逻辑与基本类似，差别在于true list和false list的具体操作可能有所不同。

比较二元表达式的语句生成。首先，生成左左右两部分表达式的语句，并获取其结果的操作数，接着，判断如果左右返回的操作数是布尔类型，应先生成强制转换语句 TypefixInstruction 进行强制转换，然后根据操作数的不同，根据左右而得到的操作数，生成判断指令CmpInstruction。

加减乘模的生成。首先，生成左右两个表达式的语句并获取操作数，然后，根据加减乘模生成不同的二元运算指令BinaryInstruction。要注意的是，左右表达式生成语句之后，都应该重新获取当前的bb，因为子表达式可能会修改当前bb，而最终新的二元运算语句应该放到最后面的bb中。

1. 后缀单目运算sufSingleExpr

对于后缀单目运算++、–，首先应生成子表达式的代码，然后获取当前BB。因为其修改了原始操作数的值（+1），首先，生成临时操作数，并增加双目加一运算指令 BinaryInstruction ，再将这个临时操作数存到原始的操作结果中，增加 StoreInstruction。最后还应处理true list和false list，继承子表达式的这两个列表。

2. 前缀单目运算sufSingleExpr

前置++、–运算与后缀单目运算符相似。首先，应加载目的地址的操作数进行加1操作，再将其存入原地址。

对于前置-。如果此表达式的返回类型为布尔值，首先应增加强制转换指指令TypefixInstruction，然后再进行二元运算 BinaryInstruction，与零相减；否则直接进行二元运算。

对于前置逻辑否!，应当判断子表达式的返回值是否为整形，如果为整形，需要添加与0比较的比较指令CmpInstruction，将其转换为布尔型。然后，根据子操作数进行二元运算NE。

每种运算的最后还应处理true list和false list，继承子表达式的这两个列表。

3. 常数、ID

关于常数，无需额外指令。对于ID，需要增加额外load的指令，将其加载到当前节点的目标操作数中。

（五） 空语句、语句块、表达式语句、序列语句

空语句、语句块、表达式语句、序列语句不需进行额外操作，调用后续节点生成后续语句即可。

（六） if、while

If语句分为两种。一种是if-then语句的翻译，首先生成两个新的语句块，then bb代表then后面的语句，end bb个代表整个if之后连接的语句块。然后，在当前BB中生成条件判断语句的中间代码。然后，使用then bb、end bb回填当前的true list和false list。如果条件返回操作数类型为整型，需要将其转换为布尔型，增加与零比较的判断指令 CmpInstruction 。接着，增加条件分支指令 CondBrInstruction ，将当前与bb块连接到then bb、end bb。then bb无条件连接到end bb即可。在then bb中生成if后语句块的中间代码。还要将两个新的语句块添加到整体语句的结构 builder 中。

If-else与之整体结构相似，首先增加then bb, else bb, end bb，然后根据条件是否为布尔类型，增加比较指令CmpInstruction换为布尔类型。进行回填。增加条件分支指令，连接起不同的块；then bb、else bb无条件连接到end bb即可。差别在于额外增加了else bb，并将else的语句生成到该bb中。

while指令。条件计算的中间代码必须为一个独立的BB，定义thenbb, endbb, condbb。首先在当前bb，增加UncondBrInstruction，cond无条件接在当前bb后。在condbb中，生成条件的中间代码，并且回填条件语句的true list和false list；接下来判断，如果条件类型为整形，同样要与零比较加CmpInstruction；之后，增加条件分支指令 CondBrInstruction 将其与end BB和then BB连接。在then bb中，生成whlle体的中间代码。同时，将新生成的所有语句块加入builder。

（七） return

通过节点中的 getParent 的函数找到整个函数的节点。如果return存在返回值，返回值为表达式类型，首先在当前BB中生成表达式返回值的中间代码，再将其存储到临时的全局变量操作数中retdst，最后生成无条件跳转指令，将当前bb后设置为整个函数结束后要跳转的bb（在function中定义）。

（八） break、continue

根据记录的parent，增加无条件分支指令 UncondBrInstruction ，Break跳转到整个while结束后的endbb，continue跳转到表达式所在的condbb。

（九） 函数定义

函数定义的处理。

首先，获取当前unit并生成新的 function ，同时建立新的语句块bb，并将其保存到当前 function 的 ret_bb 中。

然后，处理返回值，为返回值分配空间，并设置好符号表项的类型和label。如果函数存在返回值，在函数的头bb中，首先插入分配函数返回值的语句 AllocaInstruction 。

接下来处理函数参数列表，对于函数中的每一个参数。首先建立操作数和符号表项，并为为形参分配空间，将该 AllocaInstruction 也插入函数的头bb的首部。然后在当前BB插入 StoreInstruction 存储形参。

然后生成函数体的中间代码。

然后建立控制流图，遍历函数的每一个语句块，将每一个块的最后一个指令的后续块和当前的bb连接好（Succ与Pred）。每一个块最后一个指令一定是跳转指令或者是函数末尾指令。临时变量f代表当前函数之后还有没有end bb。如果f为真，说明end BB为空，也就是接下来没有bb，则直接把这个函数ret BB的标号改为函数中最后一个语句块。并且，记录函数的后续bb.

接下来处理函数返回。生成无条件跳转指令 UncondBrInstruction ，将函数中最后一个bb和下一个要返回到的语句块连ret bb连接。如果函数存在返回值,要添加 LoadInstruction ，并加 RetInstruction 时将返回的数加载到目的操作数中。如果函数没有返回值，则 RetInstruction 指令中使用空指针即可。

同时，在每次function的构造函数中，会调用unit的insertFunc方法，这时候会把这个函数存储在unit的public变量func_list中，在unit的output方法中unit.cpp),会遍历 func_list ，逐个调用函数的output方法。并且，在function的output的函数中增加了对参数列表的处理，并在生成 define 语句时，根据实际情况输入参数列表。

（十） 函数调用

首先获取当前语句块。然后判断，如果函数存在参数，每个参数都是表达式，生成每一个参数的语句，再获取每一个参数的操作数，计入列表vo，再根据当前的返回值符号表项、操作数地址，进行函数调用 CallInstruction ，这是自己实现的函数调用指令。然后判断这个函数是不是sysy库函数，如果是，需要将其写入sysylist中，要注意的是进行判断防止重复加入。sysylist是一个定义在unit中的变量，用于在整个代码中声明库函数，在unit进行output时（见unit.cpp），会单独输出该列表中的函数声明语句。

七、 目标代码生成

目标代码生成的步骤是：首先，调用 unit 的 genMachineCode 函数，调用至每个 instruction 的 genMachineCode 函数（位于 instruction.cpp 中），生成目标代码，保存在 AsmBuilder 中。然后，使用 LinearScan 类进行寄存器分配。最后，调用 munit 的 output 的函数，输出生成的目标代码。

目标代码生成的程序结构与中间代码生成类似，分为 MachineUnit 、 MachineFunction 、MachineBlock 、各类MachineInstruction 、MachineOperand 。（在 MachineCode.h 中定义）。AsmBuilder 中存有 mUnit 指针，指向 main.cpp 的 mUnit ，该类用于辅助生成 mUnit 结构，与 IRBuilder 类似。此时，为所有临时变量分配了一个虚拟寄存器 VREG （认为是无穷的）。

寄存器的分配。在构建好 mUnit 后，借助 LinearScan 实现线性扫描寄存器分配算法，单趟遍历每个活跃区间 (Interval)，为其分配物理寄存器。首先，进行活跃区间分析，框架已经完成，在 LiveVariableAnalysis 类中实现。然后，进行线性扫描寄存器分配，主要是 LinearScan 的 linearScanRegisterAllocation 函数。最后，如果有临时变量被溢出到栈中，还要生成溢出代码，主要是 LinearScan 的 spillAtInterval 函数。迭代进行以上过程，重新活跃变量分析，进行寄存器分配，直至没有溢出情况出现。

（一） 汇编语言生成genMachineCode

首先要在 main.cpp 中进行munit和unit的全局变量交接，然后调用函数，即 Instruction 类中的各个子类的 genMachineCode 函数，用于通过中间代码生成汇编代码。生成的汇编代码在 MachineCode 文件之中定义。

1. LoadInstruction

在原有基础上修改。差别在于如果临时变量的偏移量超过[500，-500]，需要增加 LoadMInstruction 和相加指令 BinaryMInstruction ,先计算出变量的addr，在进行局部变量的 LoadMInstruction 指令。

2. StoreInstruction

参照 LoadInstruction 完成。首先，要进行函数参数的处理，前三个函数的参数可以从寄存器中直接加载，如果函数参数数大于4个，需要在栈中找到对应的偏移量并根据偏移量增加load的指令，进行加载。然后，判断如果需要存储的操作数是常量类型，则需要额外增加 load指令，将其加载。接下来，与load指令相同，对全局变量，局部变量和临时变量，分别计算地址，添加 StoreMInstruction ，要注意的是dst 和 scr 应正确赋值，dst为存储的第一个操作数， src为第二个。

3. BinaryInstruction

由于汇编代码中两个操作数不可以同时为立即数，所以此情况应当首先增加 load 的指令和 mov 指令，将立即数操作数，加载到寄存器中，再进行后续操作。关于减法、乘法、除法和模，仿照加法进行操作。对于取模操作，依次进行除法、乘法、减法。对于NOT操作，首先进行比较语句 CmpMInstruction 将其与0比较，然后根据依据执行结果，将立即数零或者一，使用 MovMInstruction 加载到结果中。

4. CmpInstruction

与二元运算指令类似。首先，加载操作数，并进行立即数的 load 处理。然后，根据具体的运算符 opcode ，增加 CmpMInstruction 指令。最后，保存必要的运算结果，根据将常数0或1，生成条件 MovMInstruction ，加载到结果操作数中。

5. UncondBrInstruction

生成一条无条件(类型为B)跳转指令 BranchMInstruction。关于跳转目的操作数的生成，为自己额外记录的目的基本块号。在指令中额外记录了调用此指令的基本块，而在基本块中记录了基本块号，调用 getNo 函数进行获取。

6. CondBrInstruction

生成一条无条件跳转指令 BranchMInstruction。对于正确分支和错误分支的目的基本块号，分别根据CondBrInstruction 中记录的 true_branch 和 faslse_branch进行填写，与无条件跳转指令的处理相同。

7. RetInstruction

根据作业提示实现。

首先，当函数有返回值时，生成 MOV 指令 MovMInstruction ，将返回值保存在 R0 寄存器中。

其次，我们需要生成 ADD 指令，BinaryMInstruction，来恢复栈帧，sp为13号寄存器，首先获取sp，然后通过cur_func AllocSpace(0) 得到当前函数分配的栈结构大小，然后通过加法指令,获得执行函数之前的sp位置并更新sp。但是，这时可能无法确定add的个数，需要等待上级结构进行回填，所以这里将temp对bp置为true。

如果该函数有 Callee saved 寄存器，我们还需要生成 POP 指令恢复这些寄存器，生成 POP 指令 StackMInstructon ，恢复 src_list 中的寄存器，将在block的output中进行处理。

最后，再生成跳转指令 BranchMInstruction 来返回到 Caller，即bx lr。lr为14号寄存器，存储启动函数的下一条指令的内存地址，使程序返回到在“子”函数完成后，启动“子”函数调用的“父”函数。bx即分支交换，进行从ARM状态到Thumb状态的切换。

8. CallInstruction

实现函数调用。首先，处理函数的参数，前4个函数参数使用0-3号寄存器进行存储，生成MovMInstruction，将参数加载到寄存器中；对于第4个之后的参数，生成StackMInstructon，push到函数栈中。然后，生成bl跳转指令BranchMInstruction，根据函数名进行跳转，即在 LR 中保存（PC+4），并跳转到函数。要注意的是，函数名的获取要忽略之前的@符号。（@符号在arm汇编代码中为注释的起始符号）。接下来，由于进行了压栈，需要对SP（13号寄存器）进行恢复，生成add的指令，BinaryMInstruction。计算大于4的参数个数，也就是需要恢复的偏移量大小，由于每一个参数而均为4位，即(vo.size()-4)*4，将其恢复成之前的值。最后，通过 MovMInstruction ，和结构体中记录的操作数地址，将函数的返回值保存到 R0 寄存器中。

9. TypefixInstruction

强制转换指令，即生成带条件判断的mov指令，将其插入到当前代码语句块中。

10. GlobalInstruction

对于全局变量语句，无需在这里处理，而是在 munit 的输出函数output中直接处理。

（二） 目标代码输出

对于目标代码的构造函数和输出实现，在 MachineCode.cpp 文件中。

对于构造函数，即将传入的参数存入对应结构。 StoreMInstruction 、 MovMInstruction 、 BranchMInstruction 、 CmpMInstruction 、 StackMInstructon 的输出即按照汇编指令格式进行输出即可。在 load 和二元运算语句中，记录了是否需要回填，用于 return 和参数加载。

对于 MachineBlock 的 output 函数，遍历当前语句块中的每一个指令，并进行output输出。但是其中包括几种特殊情况，首先是return语句的add sp，且需要重填个数，则在这里进行赋值；若load指令有pop，则记录到src_list，用于return语句的add sp；若为加载参数的ldr语句，需要根据saved_regs的个数重写。

对于 MachineFunction 的 output 函数，首先获取fp、sp和lr寄存器（11、13、14）。将fp、lr其记录到函数类结构体的 src_list 中，（在RetInstruction中会对其进行恢复），并push到栈中。将fp赋为sp。为局部变量分配空间，即，将sp减其size，并将size load到4号寄存器。

对于 MachineUnit 的 output 函数，首先，调用 PrintGlobalDecl 进行全局变量的声明语句。然后，遍历函数列表生成每个函数的目标代码。最后，调用 PrintGlobalEnd 生成全局结束的语局。

（三） 寄存器分配

在构建好汇编代码的整体架构之后，汇编代码中所有的操作数的寄存器都使用的是虚拟寄存器，即认为有无穷的寄存器。但是实际上寄存器的数量是有限的。如果物理寄存器分配不够，得需要向栈中溢出（即在栈中临时store该变量的值，等到需要使用时，再把它从内存中load进来）而寄存器分配算法就是：输入已经构建好的汇编指令结构，并将所有的语句中的虚拟寄存器都转化为物理寄存器，即在每一个指令中标记好该指令中需要分配的物理寄存器的编号。

实现了线性扫描算法 [1]的计算器分配。借助 LinearScan 类进行实现。其中，regs数组记录了从activelist中释放的可用物理寄存器列表；intervals 表示还未分配寄存器的活跃区间，其中所有的 interval 都按照开始位置进行递增排序；增加了 activelist 表示当前正在占用物理寄存器的活跃区间集合，其中所有的 interval 都按照结束位置进行递增排序。其中的 Interval 结构，代表一个寄存器的区间，和关于其的一些其他信息，如对应的reg、是否溢出、defs、uses等。

进行寄存器分配.首先,线性扫描 LinearScan 的构造函数将第4$\tilde{1}$1个寄存器添加到类结构体中的可用存储器列表中.然后,执行寄存器分配函数 allocateRegisters 。遍历当前代码中的每一个函数：1. 进行活跃变量分析，活跃变量区间结果存储在 LinearScan 类中的成员变量 intervals 中。2. 调用 linearScanRegisterAllocation 进行线性扫描，给每个虚拟寄存器分配物理寄存器。寄存器如果不存在溢出则，执行 modifyCode 进行实际的分配操作。否则，进行栈溢出处理 genSpillCode，再重新重复上述过程，直至栈没有溢出。这样就完成了寄存器的分配。

1. linearScanRegisterAllocation

该函数用于实际分配reg，若需要溢出，则进行溢出标记。如实验指导书所述。遍历 intervals 列表进行寄存器分配，对任意一个 unhandled interval 都进行如下的处理：

1. 调用 expireOldIntervals 函数，根据（unhandled）interval 的开始时间和 active 的结束时间，踢掉一些 active 占用的寄存器，即寻找可用空间。

2. 类中 regs 数组记录了尚未使用的物理寄存器列表。需要首先判断这个列表是否为空：

(a) 若为空，则说明当前所有物理寄存器都被占用，需要调用 spillAtInterval 进行寄存器溢出操作，并返回 false，即将 success 置为 false。

(b) 若不为空，则说明当前有可用于分配的物理寄存器，为 unhandled interval 分配物理寄存器。首先在当前区间的 rreg 变量中记录可以分配的计算器。然后，在activelist 中加入这个区间，这时要注意activelist中所有的区间均按区间的结束时间的递增顺序存储，所以插入新值后应当将其重新排序。接下来，在可用reg列表中删除刚刚分配的寄存器。返回true，即无需溢出，success 置为 true。

2. expireOldIntervals

expireOldIntervals 函数用于寻找可用的寄存器（即activelist中不再需要的），并将其加入regs数组.根据当前要处理的区间的开始时间和活跃期间的结束时间，踢掉区间不重叠的活跃期间占用的寄存器。从前向后,即按结束时间递增的顺序，遍历 activelist，看该列表中是否存在结束时间早于 unhandled interval 的 interval（即与当前 unhandled interval 的活跃区间不冲突），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，将其在 activelist 中删除，并且将其使用的寄存器加入到可用物理寄存器列表 regs 当中。

3. spillAtInterval

spillAtInterval 函数用于判断哪个寄存器需要溢出，并将其进行置位。将结束时间最晚的一项进行溢出。

如果 activelist 中最后一个 interval 比当前 unhandled interval 的活跃区间结束时间更晚。则需要将 unhandled interval 换入 activelist。这里同样需要注意，其按区间结束时间递增排序，将其重新排序。将其占用的寄存器分配给 unhandled interval。最后，需要置activelist 中最后一个 interval 的 spill 标志位。

如果是 unhandled interval 的结束时间更晚，只需要置位其 spill 标志位即可。

4. genSpillCode

genSpillCode 函数用于处理每个需要溢出的寄存器溢出到栈的操作。遍历每一个区间，如果该区间的 spill 为真，则需要进行溢出操作，先在每一个use之前增加load的指令，再在每一个def之后增加store指令。

遍历uselist中每一个 use 指令，在其之前插入一条load指令。首先，记录栈溢出的偏移量，并将其加载到临时的立即数操作数off中。fp（寄存器11）为栈底指针。还要判断栈偏移量如果大于255或小于-255，首先应当生成一条load的指令，将off加载到临时操作数中。接下来，按照偏移量生成 LoadMInstruction ，并将其插入 use 之前。

遍历 deflist 中每一条 def ，在其后加入一条store指令。首先，仍要计算偏移量，并且判断，如果其大小超出范围，增加load指令，再根据此情况生成一条store指令。最后，将其插入到当前指令之后或新增加的load指令之后。

八、 结论

完成了编译器的基本操作，但是还有许多优化有待完成，项目代码：代码仓库链接

参考文献

[1] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.