摘 要

本文主要讨论了针对SysY2020语言的编译器实现及优化。首先设计源代码层的预优化，实现了循环展开和表达式优化等预优化。然后将代码经过词法分析，语法分析转化为AST，并自主设计了中间代码表示，从而遍历AST根据语义信息生成中间代码。经过中间代码优化器，完成了公共子表达式消除等优化手段。最后将中间代码通过所实现的映射规则转换为Arm v7-M汇编代码，在Rasp-bery 4B上运行，并进行结果校验。编译器使用C++实现，完成了所有功能测试的编译，并通过了正确性验证。

**关键词：编译器，中间代码生成，编译优化，Arm v7**

# 目 录

[目 录 II](#_Toc49994780)

[第一章 针对复杂工程问题的方案设计与实现 1](#_Toc49994781)

[1.1 针对复杂工程问题的方案设计 1](#_Toc49994782)

[1.1.1 汇编语言分析及信息提取 1](#_Toc49994783)

[1.1.2 目标文件生成 4](#_Toc49994784)

[1.1.3 链接及可执行文件生成 5](#_Toc49994785)

[1.2 针对复杂工程问题的推理分析 6](#_Toc49994786)

[1.2.1 汇编语言信息收集分析 6](#_Toc49994787)

[1.2.2 目标文件生成分析 7](#_Toc49994788)

[1.2.3 链接过程分析 7](#_Toc49994789)

[1.3 针对复杂工程问题的方案实现 8](#_Toc49994791)

[1.3.1 汇编分析器的实现 8](#_Toc49994792)

[1.3.2 目标文件生成的实现 21](#_Toc49994793)

[1.3.3 链接器的实现 23](#_Toc49994794)

[第二章 系统测试 30](#_Toc49994796)

[2.1 测试环境的搭建 30](#_Toc49994797)

[2.2 测试用例的构造与测试结果 31](#_Toc49994798)

[第三章 知识技能学习情况 41](#_Toc49994799)

[3.1 ELF文件格式 41](#_Toc49994800)

[3.2 静态链接机制 41](#_Toc49994801)

[3.3 等等等等 42](#_Toc49994802)

[第四章 分工协作与交流情况 43](#_Toc49994803)

[4.1 分工协作 43](#_Toc49994804)

[4.2 团队交流情况 43](#_Toc49994805)

[参考文献 44](#_Toc49994806)

[致谢 45](#_Toc49994807)

# 第一章 针对复杂工程问题的方案设计与实现

## 针对复杂工程问题的方案设计

### 1.1.1 语言描述及编译器架构

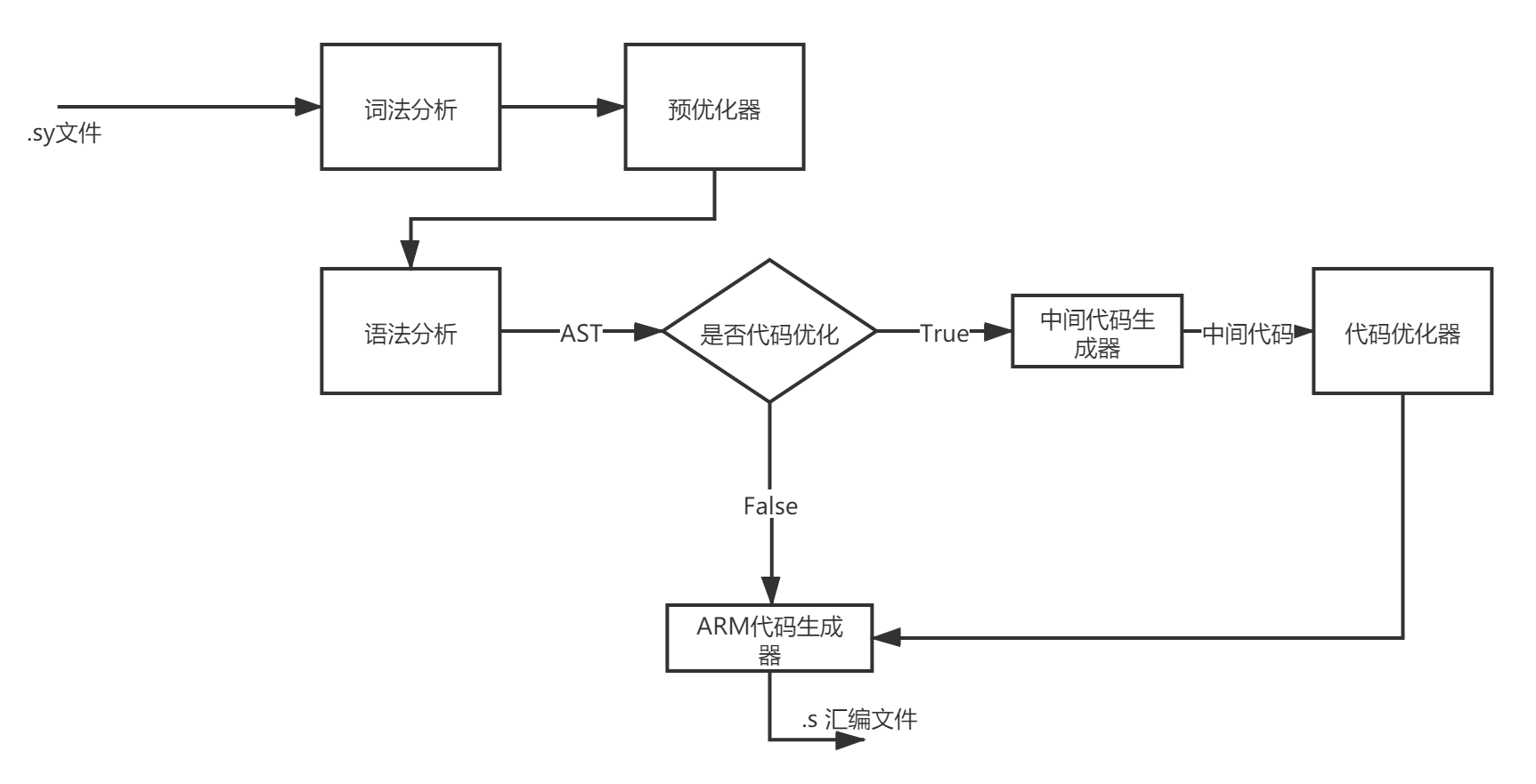
根据本次综合设计题目“编译器指令生成和优化”中的要求，我们需要实现一个编译器，能够根据Sysy2020语言程序生成对应的Arm汇编程序，同时编译器还需要能够提供在编译过程中对程序进行优化的功能。

其中Sysy2020语言为C语言子集，支持全局变量，常量声明和其他函数定义。Sysy2020语言支持int类型的32位有符号数，按行优先存储的多维数组，const可用于声明常量。在语句方面，Sysy2020支持赋值语句，表达式语句，if语句，while循环语句，break语句，continue语句，以及return语句。对于表达式，Sysy2020中包含了基本的算数运算（+，-，\*，/）以及关系运算（==，!=，>，<，>=，<=）和逻辑运算（！，&&，||）且其中的优先级和C语言保持一致。

其余特性Sysy2020语言和C语言保持一致。

对于该编译器的设计，我们采取了传统编译器的设计思路，同时加入了部分改进。首先编译器需要读取Sy源文件进行词法分析生成tokenSteam，如果编译时参数具有-O2则编译器还需要对tokenStream进行部分优化，之后通过对tokenStream进行语法分析生成对应的AST抽象语法树结构，在AST的基础上如果在编译时无 -O2参数，则直接由AST生成对应的ARM汇编代码，否则编译器需要将AST转换为中间代码，之后对中间代码进行优化，并将优化后的代码编译成ARM汇编。

编译器的总体工作流程如下图1-1-1所示：

图1-1-1 编译器工作流程

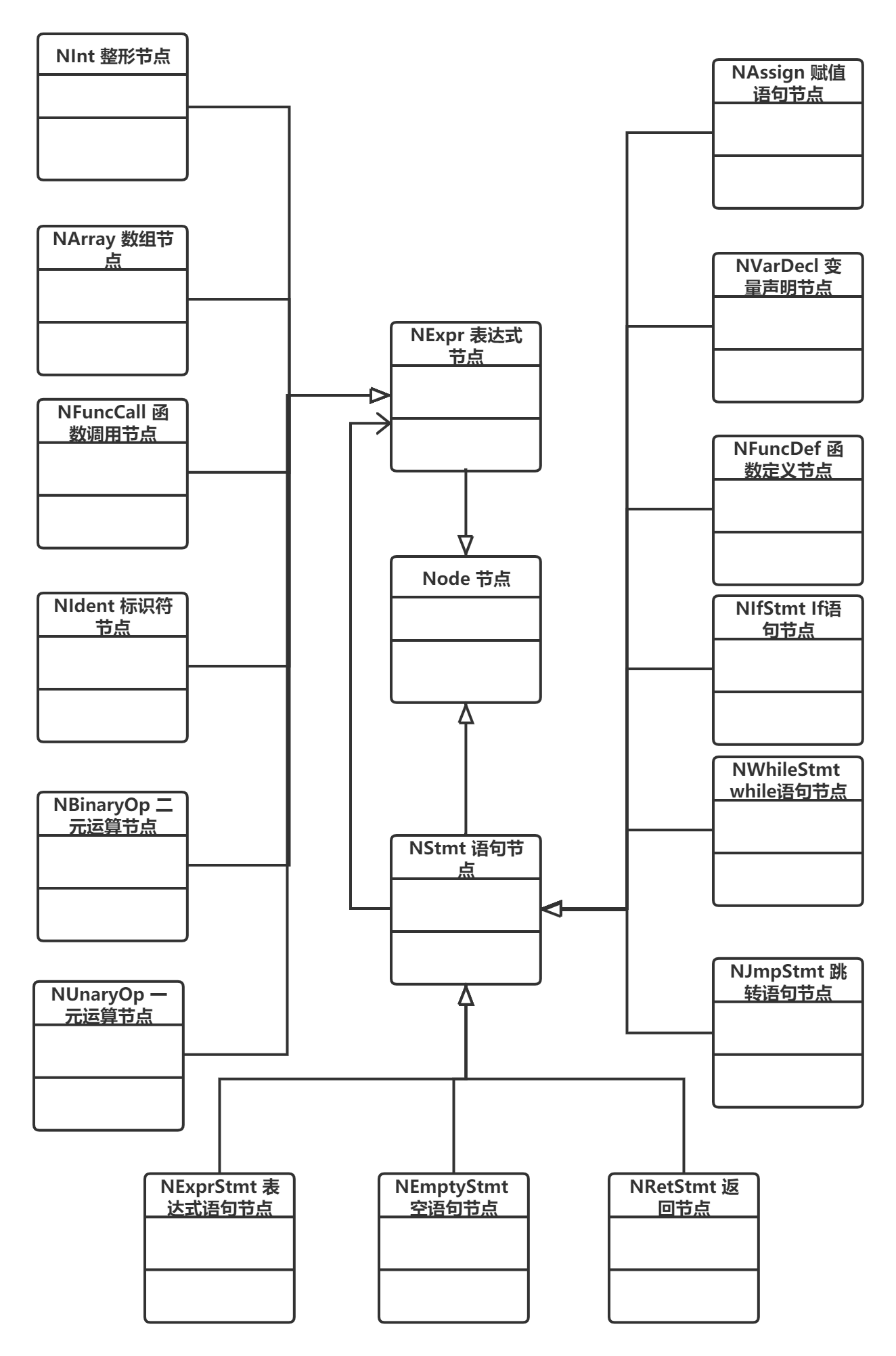
之后进行编译器各部分的详细设计。

对于词法分析，我们采用了GNU的Flex工具。根据其语法规定，在Flex文件通过手动输入正则表达式和相关定义即可自动构造自动机，生成C/C++代码文件，从而完成词法分析的工作。

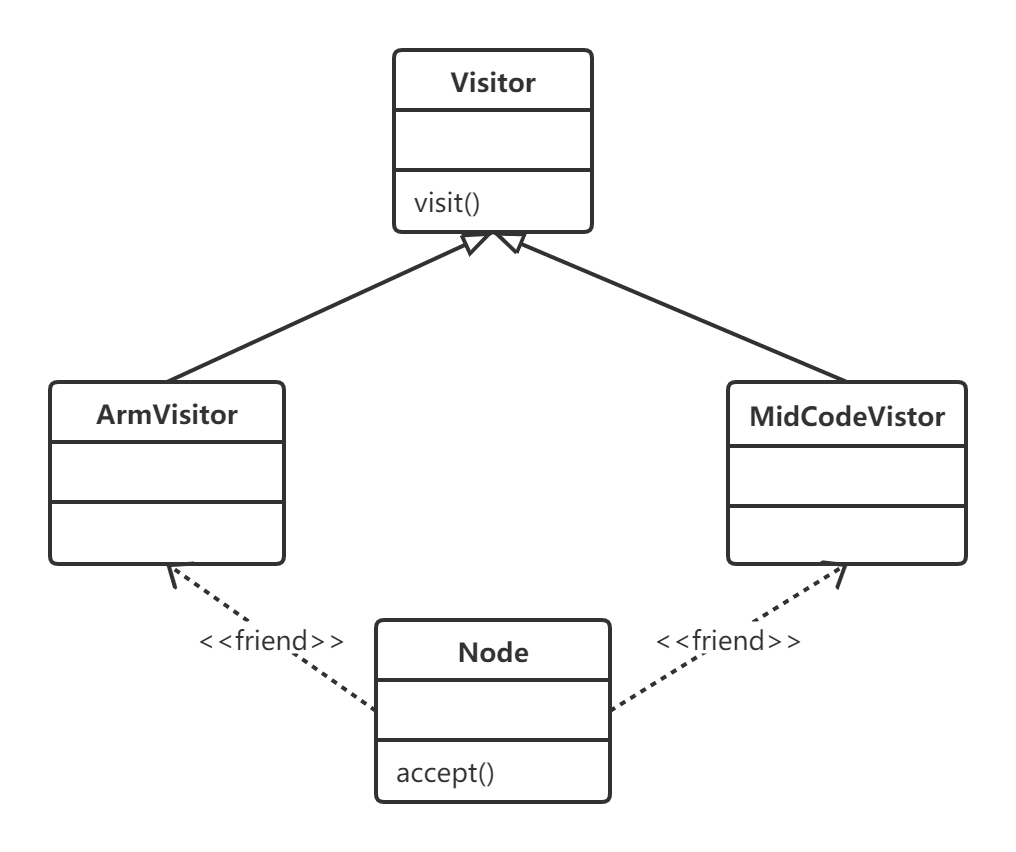
对于语法分析器的设计，我们采用Bison工具。在Bison文件中，我们根据目标语言的特点，将其语法转换成BNF输入Bison规则和对应操作即可自动生成语法分析器，并暴露语法分析接口供之后使用。

为了更清晰反应程序的层次结构，结构明确的AST必不可少。我们在设计AST的结构时采用了面向对象的设计思路，将AST节点划分为NExpr表达式节点和NStmt语句节点两大类，之后再将NExpr划分为NInt（整型节点），NArray（数组节点），NFuncCall（函数调用节点）等不同表达式节点，同理将NStmt划分为NIfStmt，NWhileStmt等不同语句节点。

AST结构如下图1-1-2所示：

图1-1-2 AST设计类图

在代码生成部分，由于我们的编译器需要根据是否进行优化来执行不同的代码生成步骤，即对同一份AST数据执行不同的操作，对此我们引入了访问者设计模式，即通过不同的Visitor对AST节点进行访问从而实现不同的操作。通过ArmGenerator和MidCodeGenerator访问AST节点可以分别生成Arm汇编代码和中间代码MidCode，结构如下图1-1-3所示：

图1-1-3 访问者类图

访问者设计模式的引入分离了数据和操作，从而降低了耦合性。其中ArmVisitor和MidCodeVisitor均被设置为AST节点类的友元类，同时Node节点使用accpet方法允许不同的Visitor进行访问操作。不同的visitor通过自身的visit方法访问AST节点，同时根据节点类型的不同执行对应操作，从而完成了不同模块对应ARM汇编或者中间代码的生成。

### 1.1.2 源代码预优化

为了保证所生成程序运行的效率，需要对源程序进行一定的优化。而优化的步骤可以放在两个地方进行优化，分别是对于中间代码进行优化，生成运行效率更高的中间代码；以及直接对源程序进行优化，生成运行效率更高的源程序。我们在本项目中同时采用了两种优化方案进行不同方面的优化。

其中，对于源程序进行的优化主要针对于循环展开，无用条件分支去除，多项式运算化简，常数替换。这个设计的核心问题就是多项式化简与求值。

通过一个单独的CodeOpt优化器，读入源码经过语法分析后的语义数据，然后根据语义，对其中所有的循环，条件判断，进行循环次数计算，条件永真永假判断。

当遇到多项式时，首先判断该多项式中的每一个值是否已知，若值已知则在优化阶段直接计算出结果并替换掉这个多项式。若该多项式存在未知的值，例如存在随机数，依赖键盘输入的值时，则根据Poly多项式化简进行化简，再判断化简后的结果是否已经可以抵消掉所有未知变量，若可以抵消掉所有未知变量则直接计算出结果并替换掉原多项式。若尚存在未知变量，则比较化简后的算式与多项式存在合并的算式哪一个存在更少次数的四则运算，并使用更少次数的算式替换在源代码中。

当遇到变量时，若该变量此时为已知值，则替换为常数。

当遇到循环次数为可计算次数时，将循环根据循环次数进行展开。

将循环展开后，再针对展开后的循环体进行优化。此时针对循环展开后的代码语句，针对内部所有的变量进行常量替换以及多项式化简和求值。

除此之外，大多数循环都会使用到循环条件判断中的循环变量，此时将循环变量进行常量替换后，可以判断在循环体后的语句是否需要使用该循环变量，若不需要使用，则将其定义循环变量以及给循环变量赋值的语句给删除掉。若需要使用则删除掉循环变量有关语句后再在后面加一句赋值语句，令循环变量等于循环结束后的最终值，以免影响上下文。

对于部分条件分支代码，是存在条件判断语句为永真语句或者永假语句的情况，此时该条件分支为不可到达的死代码，理应删除掉。当遇到条件判断语句的条件为永真条件或永假条件时，则去除该部分的代码。

### 1.1.3 中间代码设计及其优化

为了实现高效的编译优化，我们将得到的AST首先生成中间代码IR，在中间代码上进行优化，最后利用优化后的中间代码生成汇编，实现编译器的前后端分隔。在中间代码的设计上，我们以著名的开源项目LLVM里的中间表示IR为基础，结合Sysy2020语言的语法特点，自主设计出了一套基于三地址指令序列的中间代码，用于连接前后端的编译工作。

在该中间代码中，我们实现了基本的算术和逻辑运算，数组赋值，标号跳转，函数定义与调用等Sysy2020语言翻译到汇编所需的指令类型，将其封装成一个结构体，以接纳前端的写入与后端的读取。

在中间代码上的优化通过独立的优化器来实现。该优化器以前端得到的中间代码表示作为输入，对其进行数据流分析，就能实现中间代码的优化，对数据流中无用与冗余的部分进行消除和替换，提高中间代码的工作效率，最后将优化过的中间代码输出给后端实现汇编的生成。

在数据流分析中，我们进行了基本块的划分，以及将基本块组织成流图，根据得到的流图，就可以进行基本块的优化与全局代码优化，如消除全局公共子表达式、消除复制传播、常量替换和死代码消除。为了避免与其他部分的耦合，我们在实现时将优化器封装为一个类，对外部实现数据与操作隔离，且方便编译构造部分代码的调用。

## 针对复杂工程问题的推理分析

### 1.2.1 编译器架构分析

首先，对于系统总体架构的设计我们采用的是基本基于传统编译器的流程，同时为了方便后续的操作和效率的提高，我们大胆尝试在在传统的流程上略做改动。这样做原因有以下几点：

1、传统的编译器设计流程已经较为成熟，且如果想要修改具有较高的风险性和困难性，因此我们考虑系统主干依然是基于传统的编译器流程。

2、在保留传统流程主干的同时稍做修改，有利于优化的执行和效率的提高。我们在读入tokenStream的同时通过加入预优化器，在保留原有high level的表示层面进行部分优化。同时在AST生成之后如果没有选择进行优化则直接通过AST进行代码的生成，这样直接省略了中间代码生成和翻译的步骤，能够提高编译器本身的工作效率。

对于词法分析和语法分析器，我们采用了Flex&Bison的传统组合，这是因为Flex&Bison作为GNU的工具，已经具有较为悠久的历史，具有高度的可靠性，而且Flex&Bison在分析效率方面处于同类软件中的领先地位。

Flex直接根据输入的正则表达式来进行有限自动机的构造，而Bison则能够根据输入的BNF文法生成语法分析程序来对Flex输入的tokenStream进行语法匹配，并且根据规则执行不同的操作，两者能够良好配合完成词法分析和语法分析的任务。因此，Flex&Bison组合能够显著提高我们编译器前端的可靠性和稳定性，有利于之后的开发进行，同时提高开发效率。

在代码生成模块我们采用访问者模式，这是因为访问者模式能够有效地进行数据和操作的分离，即当AST节点accpet不同的Visitor时，能够执行不同的操作。这样做最大的好处是降低了系统不同模块的耦合性，方便进行之后缺陷的修复和新操作的加入。

### 1.2.2 源代码预优化分析

传统的优化方案是将所有的代码都放在中间代码层面优化。此处之所以选择将部分优化放置于源码层面的原因有以下几点：

1、中间代码的代码量往往远超于源代码，而对于一些相对来说比较基础，简单的优化在中间代码中进行，往往每一次的优化的检索上下文语义时需要遍历更多的语句段，在优化时也需要生成更多的语句段，并且由于中间代码抽象层次的降低而要做出更多的代码逻辑的判断去区分语义，这对于某些优化来说是不利的。而如果遍历AST后，根据语义直接进行优化生成优化后的源码，显然在编码实现上对于部分基础性的优化来说会更简单。

2、在开发过程中，中间代码抽象层次没有源代码高，导致优化后的中间代码的逻辑难以判断与原中间代码的逻辑是否等价。若简单地以运行结果来判断中间代码是否等价也不可行，因为测试用例是无法穷尽的，永远只能证明在已测试的用例中是等价的，但是不能确定存在我们未发现的bug。如果直接针对源码进行优化，直接生成优化后的源码，则会在判断代码优化后与优化前的等价带来便捷，因为更高的抽象层次更有利于开发人员去思考代码的运行逻辑。为了保证开发效率，将一部分的优化放在源码层面是合理的。

3、增加开发效率，使多项优化可以并发进行。

在进行算式数值求解时使用的是poly多项式化简算法而不是遍历AST得到结果的理由为：遍历AST得到运算结果依赖于每一个变量都需要得知，但是对于部分情况，部分变量的值不知道，但可以相互抵消的情况也是可以求出算式的值的。

### 1.2.3 中间代码设计分析

抽象语法树AST作为树形结构，在其上直接进行优化操作是十分困难的，考虑到优化器的可行性与高效性，我们需要一个线性的，只含基础指令和跳转语句的代码来进行优化。现代编译器对中间代码的广泛使用证明了这一方法的可行性，我们决定采用同样的方式来实现编译优化的步骤。

但作为自定义的编程语言，众多开源的中间代码结构并非完全适合Sysy2020语言，如LLVM IR，其中就有过多的冗余部分。考虑到编译效率和对语言的完整支持，我们决定自主构建基于三地址指令序列的中间代码，实现对Sysy2020语言的完整支持，同时消除冗余，提高中间代码处理效率。

### 1.2.4 数据流分析

虽然得到了中间代码，但由于跳转语句的存在，指令的执行并非顺序的。想要进行中间代码的优化，必须将指令可能的执行走向表示出来。传统的中间代码分析策略给出了基本块的概念：一个只可能从第一条语句进入，最后一条语句退出，期间连续执行的指令集。通过对基本块的划分，我们可以得到多个最大的连续执行指令块，在每个这样的基本块中，指令必然是顺序、逐条执行的，所以可以通过扫描指令序列的方式进行变量活跃分析，进而完成基本块内的优化，如消除局部公共子表达式和常量传播等。

得到了基本块后，为了实现全局的数据流分析，还需要对基本块进行连接，以反映指令的跳转关系。通过将基本块末尾的跳转语句转化为到另一基本块的有向边，可以构建流图。流图反应了整个指令序列的执行路径，可以基于此实现全局的数据流优化，如消除全局公共子表达式和全局复制传播等。

## 针对复杂工程问题的方案实现

### 1.3.1 Parser的实现

对于该项目的开发，我们采用了C++语言进行开发，一是为了使用C++容器对数据进行管理，二是为了方便实现面向对象的方式。

在词法分析阶段，我们按照Flex规定语法格式完成了flexer.l文件的编写，其中对于关键字，数值，标识符等我们直接采用了正则表达式进行匹配，部分代码如下：

其中 INTEGER\_CONST, IDENT对应如下正则表达式：

IDENT [\_a-zA-Z][a-zA-Z0-9\_]\*

DECIMAL\_CONST [1-9]\*[0-9]+

OCTAL\_CONST 0[0-7]\*

HEX\_CONST 0[xX][0-9a-fA-F]\*

INTEGER\_CONST {DECIMAL\_CONST}|{OCTAL\_CONST}|{HEX\_CONST}

"int" {

yylval.type = new TInt();

return INT;

}

"break" {

return BREAK;

}

"continue" {

return CONTINUE;

}

"return" {

return RETURN;

}

"const" {

return CONST;

}

{INTEGER\_CONST} {

yylval.expr = new NInt(my\_atoi(yytext));

return NUMBER;

}

{IDENT} {

string name = string(yytext);

yylval.ident = new NIdent(name);

return IDENT;

}

词法分析关键部分在于对多行注释的处理，我们处理多行注释部分代码如下所示：

"/\*" {

int c;

while( ( c = yyinput() ) != 0 ) {

if(c == '\n') {

++ yylineno;

} else if (c == '\*') {

c = yyinput();

if( c == '/')

break;

else

unput(c);

}

}

}

我们的处理方法是当遇到 ‘/\*’ 即跨行注释开始标志之后，我们每次从输入缓冲区读取一个字符，如果是换行符则行数加一，如果读到了‘\*’则说明可能注释要结束，这时尝试读取下一个字符，如果是‘/’则代表跨行注释结束，此时终止读取。否则将试探性读取的值放入输入缓冲区，对于其他字符则直接忽略丢弃。

在语法分析阶段，我们根据Sysy2020的语法将其转换为BNF之后输入到Bison的规则区并定义相关操作从而完成语法分析和AST的构造，部分代码如下：

program: comp\_unit {

prog\_block = $<block>1;

}

;

comp\_unit: comp\_unit decl {

$<block>1->append($<stmt>2);

$<block>$ = $<block>1;

$<vardecl>2->is\_global\_ = true;

}

| comp\_unit func\_def {

$<block>1->append($<stmt>2);

$<block>$ = $<block>1;

}

| %empty {

$<block>$ = new NBlock();

}

;

decl: const\_decl {$<stmt>$ = $<stmt>1;}

| var\_decl {$<stmt>$ = $<stmt>1;}

;

decl: const\_decl {$<stmt>$ = $<stmt>1;}

| var\_decl {$<stmt>$ = $<stmt>1;}

;

const\_decl: CONST INT const\_def\_list ';' {

$<type>2->is\_const\_ = true;

NVarDecl\* decl = new NVarDecl($<type>2, $<assign\_list>3);

if(loop\_depth > 0) {

loop\_decls.push\_back(decl);

decl->is\_in\_loop\_ = true;

}

$<stmt>$ = decl;

}

;

const\_def\_list: const\_def\_list ',' const\_def {

$<assign\_list>1->push\_back($<assign>3);

$<assign\_list>$ = $<assign\_list>1;

}

| const\_def {

$<assign\_list>$ = new vector<NAssign\*>();

$<assign\_list>$->push\_back($<assign>1);

}

;

const\_def: declarator\_const '=' const\_init\_val {

parse\_time\_symtbl->push($<ident>1);

$<assign>$ = new NAssign($<ident>1, $<expr>3);

// 设置ident的值,类型

$<ident>1->setValue($<expr>3);

}

我们将程序的语法结构划分成小的comp\_unit，comp\_unit则包括一般声明和函数声明，一般声明又包括了常量声明和变量声明，而函数声明则包含了函数返回值类型，函数名称，参数列表以及函数体。我们这样自顶向下逐步细分，从而完成了BNF型语法的构造。

在不同语法单元之后的花括号中，我们定义了分析的同时采取的操作。例如在const\_def单元我们直接创建一个NAssign类型的AST节点，其中存储了语法分析时各个语法单元的信息，这样即可在分析过程中并行构造树形AST反应代码的语法结构。

其中由于语法分析时存在R/S移进规约冲突，即扫描到特定的语法单元时Bison接下来有两种走向，从而导致冲突。因此，为了解决冲突，我们采取了如下的操作方式：

%precedence LOW\_PREC

%precedence ELSE

…

if\_stmt: IF '(' cond ')' stmt %prec LOW\_PREC {

$<stmt>$ = new NIfStmt($<expr>3, $<block>5, nullptr);

}

| IF '(' cond ')' stmt ELSE stmt {

$<stmt>$ = new NIfStmt($<expr>3, $<block>5, $<block>7);

}

;

通过Bison提供的%precedence我们定义了LOW\_PREC和ELSE的优先级顺序，即优先级LOW\_PREC < ELSE，这样当程序扫描完 ELSE之前的语句之后，由于ELSE优先级更高，就会采取移进的方式，从而解决了此处的R/S conflict。

在代码生成阶段，我们需要根据语法分析生成的AST来进行ARM汇编和中间代码的生成，由于我们采用了访问者设计模式，将AST的数据和代码生成操作进行了分离，其中部分代码如下所示：

class Node {

public:

virtual ~Node(){};

virtual void accept(Visitor\* visitor) = 0;

virtual void print() = 0;

};

Node类中设置了纯虚函数accpet，其中以Visitor为参数。不同的子类即可继承accpet执行不同的访问操作，部分代码如下：

void NExpr::accept(Visitor\* v) {

this->accept(v);

}

void NStmt::accept(Visitor \*v) {

this->accept(v);

}

void NIdent::accept(Visitor\* v) {

v->visitIdent(this);

}

void NInt::accept(Visitor\* v) {

v->visitIntValue(this);

}

void NBinaryOp::accept(Visitor\* v) {

v->visitBinaryOp(this);

}

void NUnaryOp::accept(Visitor\* v) {

v->visitUnaryOp(this);

}

不同类型的节点在accpet访问者之后，Visitor进行了不同的访问操作，这样就可以生成对应的ARM代码或者中间代码，同时由于操作和数据分离，降低了代码的耦合性，有利于缺陷修复和维护。

在代码生成时，针对不同的AST节点我们需要执行不同的访问操作，从而生成对应的ARM汇编代码。由于代码较长，下面就仅以visitIdent为例进行分析：

void Generator::visitIdent(NIdent \*ident) {

NIdent\* id = symtbl->findVar(ident->name\_);

// ident值的归宿

int i;

if(!id) {

fprintf(stderr, "undefined symbol: %s\n", ident->name\_);

exit(EXIT\_FAILURE);}

// 对于栈空间中的变量

if(id->loc\_->isInMemory()) {

// 对于栈空间中非数组变量

if(!id->type\_->isArray()) {

i = getReg();

Emit("ldr", regs[i].name, id->loc\_->getLocation());

cur\_str\_reg = regs[i].name;

return;

}

i = getReg();

// 取得地址

if (id->is\_addr) {

Emit("ldr", regs[i].name, id->loc\_->getLocation());

} else {

// 如果不是指针参数，则直接在栈空间中加载地址

int temp = getReg();

Emit("ldr", regs[temp].name, "=#" + to\_string(id->loc\_->getMemOff()));

Emit("add", regs[i].name, "fp", regs[temp].name);

regs[temp].is\_used = false;

}

// 加上偏移量得到元素地址

int index\_reg = genVarArrayIndex(ident);

if(index\_reg != -1) {

Emit("add", regs[i].name, regs[i].name, regs[index\_reg].name);

regs[index\_reg].is\_used = false;

}

// 提取元素

if(!isPassAddr(ident)) {

Emit("ldr", regs[i].name, regs[i].name, true);

}

// 表示当前存值寄存器

cur\_str\_reg = regs[i].name;

return;

}

// 如果在寄存器

if (id->loc\_->isInReg()) {

i = getReg();

Emit("mov", regs[i].name, id->loc\_->getLocation());

// 表示当前存值寄存器

cur\_str\_reg = regs[i].name;

return;

}

// 对于全局变量

// 如果没加，则加入函数所用的全局变量vector

if(!findGloblData(id)) {

globl\_data.push\_back(id);

}

// 对于非数组变量,直接读取值

if(!id->type\_->isArray()) {

// 读入全局变量所在地址

i = getReg();

Emit("ldr", regs[i].name, id->loc\_->getLocation());

// 从地址读取全局的值

Emit("ldr", regs[i].name, regs[i].name, true);

// 表示当前存值寄存器

cur\_str\_reg = regs[i].name;

return;

}

// 加载数组首部地址

i = getReg();

Emit("ldr", regs[i].name, id->loc\_->getLocation());

// 根据下标获得所取元素在数组的偏移

int index\_reg = genVarArrayIndex(ident);

if(index\_reg != -1) {

Emit("add", regs[i].name, regs[i].name, regs[index\_reg].name);

regs[index\_reg].is\_used = false;

}

// 取出元素

if(!isPassAddr(ident)) {

Emit("ldr", regs[i].name, regs[i].name, true);

}

// 表示当前存值寄存器

cur\_str\_reg = regs[i].name;

}

在visitIdent处理标识符节点时，首先从符号表里面查找原NIdent信息，如果没有查找到原有的id信息，则报错提示 “undefined symbol”。之后我们需要根据ident所在位置的不同来做出不同的处理，对于局部变量，如果不是数组元素，则直接从对应的栈空间中对应位置通过ldr读取其值，并加载到寄存器中。如果是数组元素，则首先读出数组首元素所在地址，并根据其下标计算出数组元素的偏移量，之后两者相加即可得到元素所在地址，这时候只需要将其从内存中ldr读取出来即可完成操作。

代码生成部分中一大难点在于复杂条件的处理，即如何正确生成诸如

if( a == 2 && b >= 3 || c <= 5) {

//do something

} else {

// do other things

}

对应的汇编代码。

我们通过新增Condition条件类来完成此类操作：

class Condition {

public:

// 是否已经打印

bool has\_flush;

// Label标签

NLabelStmt \*cond\_label;

// 条件及语句

string do\_when;

NExpr \*do\_expr;

// 满足条件后执行操作的label及其情况

Condition \*then\_cond;

// 不满足条件后执行的label和情况

Condition \*end\_cond;

void setThenDo(Condition \*then) {

Condition \*p = this;

while (p->then\_cond)

p = p->then\_cond;

p->then\_cond = then;

}

void setEndDo(Condition \*end) {

Condition \*p = this;

while (p->end\_cond) {

p = p->end\_cond;

}

p->end\_cond = end;

}

string getLabelStr() {

if (!this->cond\_label)

return "";

return this->cond\_label->getTag();

}

void flushCondition(Generator \*g, NLabelStmt &end, NLabelStmt &begin);

void flushCondition(MidGenerator \*m, NLabelStmt &end, NLabelStmt &begin);

};

我们在Condition类中设置了then\_cond和end\_cond分别指向当满足当前条件后的下一步条件和不满足时的下一步条件，do\_expr则表示判断当前条件语句，do\_when表示跳转条件。我们通过如下代码来进行Condition类生成：

Condition \*genCondition(NExpr \*op)

{

// dynamic\_cast可以将基类指针转换为继承类指针，如果非预期对象，返回null

NBinaryOp \*binary\_op = dynamic\_cast<NBinaryOp \*>(op);

// 如果不是binary那就直接改成binary，和0比较。

if (!binary\_op)

{

binary\_op = new NBinaryOp(EQ, op, new NInt(0));

return new Condition(binary\_op, "ne");

}

Condition \*cond = nullptr;

switch (binary\_op->op\_)

{

case '<':

return new Condition(op, "lt");

case '>':

return new Condition(op, "gt");

case LE:

return new Condition(op, "le");

case GE:

return new Condition(op, "ge");

case EQ:

return new Condition(op, "eq");

;

case NE:

return new Condition(op, "ne");

case OR:

// 先设条件为左值

cond = genCondition(binary\_op->lhs\_);

// 如果左值满足则不需要查看右值,否则需要继续查看右值

for (

Condition \*p = cond,

\*tmp = genCondition(binary\_op->rhs\_);

p; p = p->then\_cond)

{

p->setEndDo(tmp);

}

break;

case AND:

// 先设条件为左值

cond = genCondition(binary\_op->lhs\_);

// 如果条件满足再查看右值，否则根据短路原则直接结束判断

cond->setThenDo(genCondition(binary\_op->rhs\_));

break;

}

return cond;

}

对于一般条件，即除了 || 和 && 之外的操作我们直接通过Condition的构造函数实例化对象。对于 || 我们则需要考虑短路原则，即通过先将lhs递归生成Condition之后，通过setEndDo保证 condA || condB 如果condA不满足才需要考虑condB。同理对于 && 我们如果condA满足时需要检查condB，否则直接执行end\_cond

之后我们通过如下代码来进行Condition类的代码实例化：

void Condition::flushCondition(Generator \*g, NLabelStmt &end, NLabelStmt &begin)

{

// 如果为空，或者已经打印过则不执行

if (this->has\_flush)

return;

this->has\_flush = true;

// 打印执行语句

if (this->do\_expr)

{

this->do\_expr->accept(g);

}

// 如果满足条件，且下一个条件不为空，继续跳转判断

if (this->then\_cond)

{

g->Emit("b" + this->do\_when, this->then\_cond->getLabelStr());

}

else

{

// 如果then完了就说明条件全部符合，跳到下一段代码开始处

g->Emit("b" + this->do\_when, begin.getTag());

}

/\*\* 如果不满足条件

\* 1. 如果不满足时，仍然有操作，则跳转到下一个不满足时的判断操作

\* 2. 如果不满足时，无继续的操作，则直接跳转到end\_label;

\* \*/

if (this->end\_cond)

{

g->Emit("b", this->end\_cond->getLabelStr());

}

else

{

g->Emit("b", end.getTag());

}

if (this->then\_cond && !this->then\_cond->has\_flush)

{

g->EmitLabel(this->then\_cond->getLabelStr());

this->then\_cond->flushCondition(g, end, begin);

}

if (this->end\_cond && !this->end\_cond->has\_flush)

{

g->EmitLabel(this->end\_cond->getLabelStr());

this->end\_cond->flushCondition(g, end, begin);

}

}

首先先打印出其判断条件语句对应的ARM代码，同时标记当前Condition已经打印过，之后递归打印满足条件后的下一个Condition及其tag标签，不满足时的Condition及其标签tag。其中begin是满足条件后需要执行的外部代码标签，而end则是不满足时的外部代码标签。

这样就解决了代码生成时的复杂条件判断问题。

当用户需要优化时，我们需要从中间代码来生成对应的ARM指令，其中生成中间代码类似地使用visit操作，只是将生成ARM改为生成对应中间代码，再次便不赘述。

对于中间代码到汇编则是进行逐条的翻译，部分代码如下所示：

void codeGen(NmidCode mid) {

switch(mid.op) {

// Binary Calculate

case cal\_op::ADD:

case cal\_op::SUB:

case cal\_op::MUL:

case cal\_op::DIV:

case cal\_op::MOD:

printf("binary\n");

genBinary(mid);

break;

// Unary Calculate

case cal\_op::NEG:

case cal\_op::NOT:

printf("unary\n");

genUnary(mid);

break;

// Assign

case cal\_op::COPY:

printf("copy\n");

genAssign(mid);

break;

case cal\_op::COPY\_AtoI:

printf("atoi\n");

genArrayToInt(mid);

break;

case cal\_op::COPY\_ItoA:

printf("itoa\n");

genIntToArray(mid);

break;

…

default:

fprintf(stderr, "Error: illegal mid-code type!\n");

exit(EXIT\_FAILURE);

}

}

显然，由中间代码生成ARM汇编只需要根据中间代码种类的不同来执行不同的操作即可生成对应的汇编代码。

### 1.3.2 源代码预优化的实现

源码优化的核心问题其实实质上就是为算式求值，算式求值直接关系到条件判断的真假计算，循环次数的计算，多项式化简，多项式求值。

而此处使用的一个多项式化简算法则是Poly多项式化简算法。具体算法流程的伪码实现如下：

算法：Polynomials

输入：字符串形式的算式：String expression，键值对符号表内容：Map<String,int> symTable）

输出：元素为多项式化简后的每一带系数项的数组polynomial\_array。

1. expression = reversePoland(expression)

//将expression 转换为逆波兰表达式以去除括号

2. polynomial\_array= calculate(expression)//计算结果存储在数组中

//该数组用于存储多项式的每一项的具体内容

3. For item,index in polynomial\_array：

4. if (var := getVar(item)&& coefficient := getCoefficient(item))

5. newItem := coefficient+String(symTable[var])

6. polynomial\_array[Index] = newItem

7. return polynomial\_array;

使用该算法实现了poly多项式化简后只需要将对返回值进行简单的判断即可了，若返回值的数组元素个数为0，则说明该多项式的结果为0；若返回值的数组元素全为可转化为数字的字符串，则可以直接将每一个元素转为数字后求和得到数值运算结果。若返回值元素中仍含有变量名，则说明该多项式存在未知变量且无法消去。

然后是运用算法进行死代码消除以及循环优化。

在条件判断语句以及循环语句中都具有“条件”，条件本质上是一个算式。此时就可以根据上文提到的多项式计算出结果后进行相应的优化：

例如： while(a < b) 可转化成 loopTimes = (b-a+1>0)?b-a+1:0

if (a<b) 可转化成 b-a>0?true:false

除了极少数情况外不能进行优化，例如多项式变量未知且无法消去，条件中包含了函数调用等情况是无法计算出多项式结果的。其他的情况多可以根据多项式算法去进行相应的映射优化。

优化完之后通过重定向将优化完的代码生成一个“tmp.sy”文件，以该优化后的文件重新生成AST并将新的AST的语义映射转换为中间代码。

最后是循环展开后的进一步优化。

将循环展开后的代码单独存储在一个可以表示其语义的数据结构中(表形式，树形式皆可)，然后可以展开后的循环做出以下优化：

Case1 同一变量赋值合并：

a = i\*2;a = i\*2 变换为 a = i\*4

Case2 消除循环变量：

i += 1;a[i] = 6;i+=1; a[i]=6; 变换为 a[5]=6;a[6]=6;

进行该优化时需要遍历原上下文确保下文没有使用循环变量i。

### 1.3.3 中间代码的设计与流图的构建

我们使用的中间代码是由三地址指令序列的基础上发展而来，对于非跳转语句，一般由一个可变的左值和两个右值组成，形如 x = y op z；而对于跳转语句，一般由条件和跳转标号组成。对中间代码结构的定义如下所示：

struct NmidCode {

cal\_op op;

cmp\_op cmp;

string lval;

string tag;

rVal rVal1, rVal2;

string to\_tag;

bool is\_init;

vector<rVal> paraList;

};

其中op作为类型标识号，用以标识该条指令的类型，如ADD，COPY和FUNC\_CALL等。为了提高程序的可读性，这里将op设置成cal\_op类型，其定义如下：

enum class cal\_op {

ADD = 0, SUB, MUL, DIV, MOD, NEG, NOT, COPY, COPY\_AtoI, COPY\_ItoA, GOTO\_U, GOTO\_C, FUNC\_CALL, FUNC\_DEF, RETURN, INT\_DECL, ARR\_DECL, NOP

};

枚举（enum）类型可以提供很好的名称到整型数值的转化，在保留对名称的使用的基础上，同时避免了字符串的匹配引起的效率问题，使得在对op进行判断和选择时的代码更容易组织，也更易于阅读分析。但传统的枚举可能会导致命名冲突，特别是在编译器这种庞大的体系结构下，不严格的数据隔离性可能导致与其余模块发生碰撞。在前端的语法分析部分，使用bison时就曾经创建过类似命名的枚举，为了避免冲突，在编码实现时我们选择了C++特有的更加安全强枚举类型（enum class），实现了数据隔离的需求。

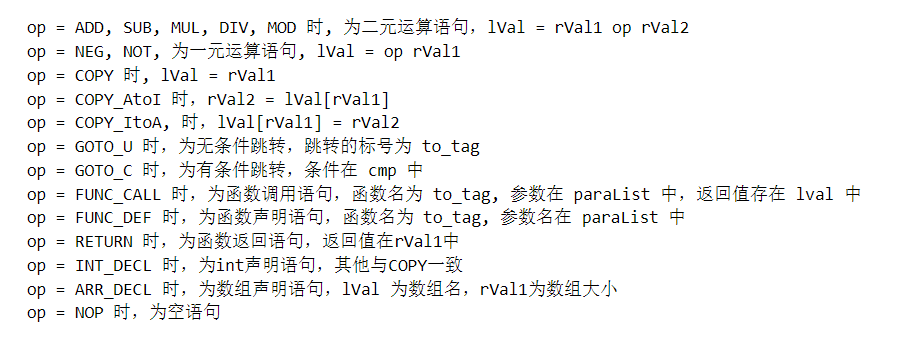
在op的取值主要分为三大类型：基本运算，标号跳转和函数调用。在基本运算中，左值必须为变量，故定义成string类型；但右值可能是常量或变量，且由于union类型不支持成员变量含有string，所以我们定义了结构体rVal，用于指示右值类型和内容。同理，跳转语句类似，将要跳转到的语句放入to\_tag变量里，函数调用将参数放入paraList中。具体的op与数据对应关系如图1-3-1所示。

图1-3-1 中间代码标识符解释

可以验证，该结构可以支持所有Sysy2020语言的指令，即可以将由源代码生成的抽象语法树AST转化成中间代码，不丢失需要的指令和控制信息。

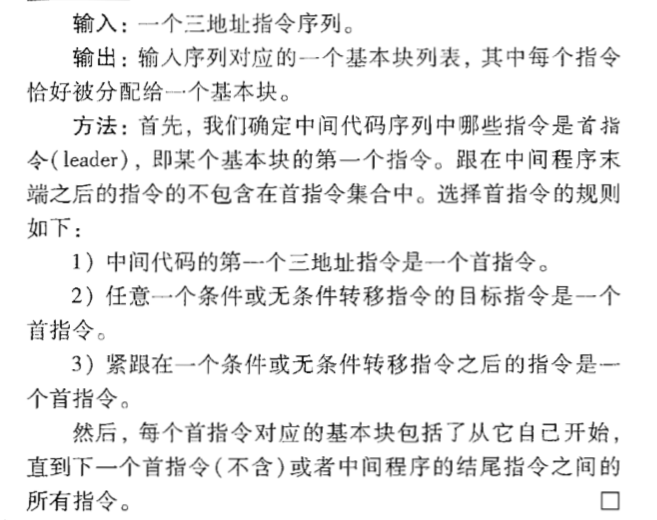
 根据设计与分析部分的结论，首先我们需要将中间代码划分为基本块。其划分策略可以由迭代的方式得到，编译原理[5]中对其的描述如下图所示：

图1-3-2 基本块划分算法

可以证明，该算法可以得到所有的首指令，进一步得到所有的基本块。具体的代码实现如下所示：

for(int i = 0; i < midCodeSize; i++)

{

    if(!midCode->at(i).tag.empty())

    {

        if(start == i)

            continue;

        temp.start = &(midCode->at(start));

        temp.end = &(midCode->at(i-1));

        fGraph.push\_back(temp);

        start = i;

    }

    if(midCode->at(i).op >= cal\_op::GOTO\_C && midCode->at(i).op <= cal\_op::FUNC\_CALL)

    {

        temp.start = &(midCode->at(start));

        temp.end = &(midCode->at(i));

        fGraph.push\_back(temp);

        start = i+1;

    }

}

if(start <= midCode->size())

{

    temp.start = &(midCode->at(start));

    temp.end = &(midCode->back());

    fGraph.push\_back(temp);

}

程序通过遍历中间代码，根据跳转语句来划分基本块，按序将基本块的首尾语句指针放入流图结构体fGraph中，将中间代码分成多个基本块组成的集合。

得到了所有的基本块之后，就可以根据跳转信息初始化流图，得到中间代码的整体结构。代码如下所示：

map<string, NBasicBlock\*> tag\_map;

for(int i = 0; i < fGraphSize; i++)

{

    if(!fGraph[i].start->tag.empty())

        tag\_map.insert(pair<string, NBasicBlock\*>(fGraph[i].start->tag, &fGraph[i]));

}

for(int i = 0; i < fGraphSize; i++)

{

    if(fGraph[i].end->op == cal\_op::GOTO\_U || fGraph[i].end->op == cal\_op::FUNC\_CALL)

    {

        fGraph[i].next = tag\_map.find(fGraph[i].end->to\_tag)->second;

        fGraph[i].jmp = NULL;

    }

    else if(fGraph[i].end->op == cal\_op::GOTO\_C)

    {

        fGraph[i].jmp = tag\_map.find(fGraph[i].end->to\_tag)->second;

        if(i < fGraphSize - 1)

            fGraph[i].next = &fGraph[i+1];

    }

    else if(i < fGraphSize - 1)

    {

        fGraph[i].next = &fGraph[i+1];

        fGraph[i].jmp = NULL;

    }

}

由此得到了具有基本块划分和跳转信息的流图结构fGraph，数据流分析就可以基于此构建。

### 1.3.4 中间代码优化的实现

这里以公共子表达式消除为例，讨论数据流分析和代码优化过程。

公共子表达式的冗余问题指，如果一个表达式E已经计算过了，并且从之前的计算到下一次使用E时，E中所有变量的值都没有发生变化，那么E的这次出现就成为了公共子表达式。对于这种表达式，无需再对它进行计算，只需要直接用前面计算过的表达式结果代替E。

要识别公共子表达式，首先要对变量的最近修改时间进行标识，这可以通过遍历流图实现。但由于具有跳转信息，所以并不能直接线性的扫描流图中的各基本块，而是必须利用搜索算法实现。我们采取的方案是利用深度优先搜索DFS来实现这一过程。调用函数如下所示：

void midCodeOpt::initChangeInfo()

{

    if(midCode->empty())

        return;

    changeInfo.resize(midCode->size());

    map<string, int> global\_change\_info;

    set<int> has\_jmp;

    initChangeInfo\_dfs(&fGraph[0], global\_change\_info, has\_jmp);

}

其中的DFS实现的搜索部分如下所示：

void midCodeOpt::initChangeInfo\_dfs(NBasicBlock\* block, map<string, int> global\_change\_info, set<int> has\_jmp)

{

int start = block->start - &(midCode->front());

int end = block->end - &(midCode->front());

for(int i = start; i <= end; i++)

{

if((midCode->at(i).op >= cal\_op::ADD && midCode->at(i).op

<= cal\_op::COPY\_AtoI) || midCode->at(i).op == cal\_op::FUNC\_CALL)

{

map<string, int>::iterator it = global\_change\_info.find

(midCode->at(i).lVal);

if(it == global\_change\_info.end())

global\_change\_info.insert( map<string, int>::value\_type

( midCode->at(i).lVal, i) );

else

it->second = i;

if(midCode->at(i).op >= cal\_op::ADD && midCode->at(i).op <= cal\_op::MOD)

{

if(midCode->at(i).rVal1.type == VIDENT)

{

if(changeInfo[i].pos\_r1 == 0)

{

changeInfo[i].rVal1\_lc = global\_change\_info.find

(midCode->at(i).rVal1.ident)->second;

changeInfo[i].pos\_r1 = 1;

}

else if(changeInfo[i].pos\_r1 == 1 && changeInfo[i].

rVal1\_lc == global\_change\_info.find(midCode->at(i).rVal1.ident)->second)

{}

else

++changeInfo[i].pos\_r1;

}

if(midCode->at(i).rVal2.type == VIDENT)

{

if(changeInfo[i].pos\_r2 == 0)

{

changeInfo[i].rVal2\_lc = global\_change\_info.find

(midCode->at(i).rVal2.ident)->second;

changeInfo[i].pos\_r2 = 1;

}

else if(changeInfo[i].pos\_r2 == 1 && changeInfo[i].

rVal2\_lc == global\_change\_info.find(midCode->at(i).rVal2.ident)->second)

{}

else

++changeInfo[i].pos\_r2;

}

}

}

}

if(block->next != NULL)

initChangeInfo\_dfs(block->next, global\_change\_info, has\_jmp);

if(block->jmp != NULL)

{

int index = block - &(fGraph.front());

if(has\_jmp.find(index) != has\_jmp.end())

{

initChangeInfo\_dfs(block->jmp, global\_change\_info, has\_jmp);

has\_jmp.insert(index);

}

}

}

该函数通过深度优先搜索的方式，遍历出所有可行路径，每一次搜索维护一个vector has\_jmp，避免出现无限递归搜索。在搜索中，更新所有表达式的右值的最近修改时间，实现对数据流的分析，并以此判断出可能的公共子表达式。

得到数据流信息后，只需要通过遍历将所有的公共子表达式着色，将每一组公共子表达式进行标记，最后通过一次相似的DFS即可实现所有的公共子表达式替换。

通过相似的策略可以对中间代码实现其他各项优化。

# 第二章 系统测试

## 2.1 测试环境的搭建

为了对该编译器进行测试，我们在docker上构造了测试环境，测试环境基于ubuntu20.04系统，在编译器方面则使用了clang/clang++ 10来对我们的项目进行编译。

其中dockerfile如下所示：

FROM ubuntu:20.04

# Install clang and gcc

RUN apt-get update && apt-get install -y build-essential clang-10 git libantlr4-runtime-dev

# Create link

RUN ln -s /usr/bin/clang-10 /usr/bin/clang && ln -s /usr/bin/clang++-10 /usr/bin/clang++

# Start from a Bash prompt

CMD [ "/bin/bash" ]

通过命令:

即可构建docker镜像。之后通过

共享当前目录，同时创造测试容器。

运行

或者使用MAKEFILE

即可编译生成当前平台可执行文件。

## 2.2 测试用例的构造与测试结果

由于编译器功能具有一定的复杂性，测试具有一定的难度。于是我们需要结合Sysy2020语言的特点来对我们的项目进行测试，在测试用例方面我们采用由浅入深，逐渐扩散功能的思路来对编译器的测试用例进行设计。

首先我们需要测试最基本的程序能否运行成功，测试用例0\_main.sy如下所示：

int main(){

return 0;

}

通过运行项目中实现的编译器

首先我们需要检测compiler能否正确生成对应ARM汇编代码，且不发生运行时错误，经过测试该用例生成的ARM汇编代码main.s如下所示：

.arch armv7-a

.eabi\_attribute 28, 1

.eabi\_attribute 20, 1

.eabi\_attribute 21, 1

.eabi\_attribute 23, 3

.eabi\_attribute 24, 1

.eabi\_attribute 25, 1

.eabi\_attribute 26, 2

.eabi\_attribute 30, 6

.eabi\_attribute 34, 1

.eabi\_attribute 18, 4

.text

.globl main

.arm

main:

push {r4, r5, fp, lr}

mov fp, sp

ldr r8, =#4

sub sp, sp, r8

ldr r4, =#0

mov r0, r4

mov sp, fp

pop {r4, r5, fp, pc}

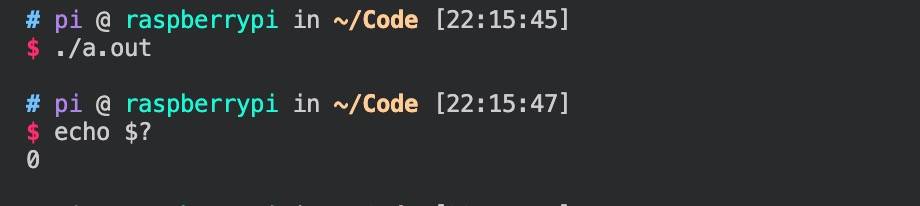
 之后我们通过将main.s 拷贝到树莓派4B上进行运行，对于该测试用例我们需要检测其返回值是否为0，即通过linux下命令 “echo $?” 来查看程序返回值，经检测结果如图2-1所示。

图2-1 运行结果1

之后为了检测变量的声明，变量的运算，我们采用了如下的测试用例add.sy：

int a;

int b;

int main(){

a=10;

b=20;

int c;

c = a + b;

return c;

}

测试过程同上。经测试compiler可以完成该用例的代码生成，生成的汇编代码如下所示：

.arch armv7-a

.eabi\_attribute 28, 1

.eabi\_attribute 20, 1

.eabi\_attribute 21, 1

.eabi\_attribute 23, 3

.eabi\_attribute 24, 1

.eabi\_attribute 25, 1

.eabi\_attribute 26, 2

.eabi\_attribute 30, 6

.eabi\_attribute 34, 1

.eabi\_attribute 18, 4

.comm a, 4, 4

.comm b, 4, 4

.text

.globl main

.arm

main:

push {r4, r5, fp, lr}

mov fp, sp

ldr r8, =#4

sub sp, sp, r8

ldr r4, =#10

ldr r5, .L0

str r4, [r5]

ldr r4, =#20

ldr r5, .L1

str r4, [r5]

ldr r8, =#4

sub sp, sp, r8

ldr r4, .L0

ldr r4, [r4]

ldr r5, .L1

ldr r5, [r5]

add r4, r4, r5

str r4, [fp, #-8]

ldr r4, [fp, #-8]

mov r0, r4

mov sp, fp

pop {r4, r5, fp, pc}

.L0:

.word a

.L1:

.word b

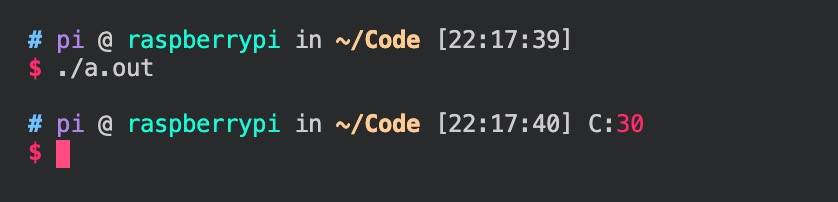
 之后将汇编文件拷贝到树莓派4B运行结果如图2-2所示。

图2-2 运行结果2

为了更深一步检测程序正确性，我们加入数组和循环while进行测试，测试用例array.sy如下所示：

int a[5];

int main(){

a[0] = getint();

a[1] = getint();

a[2] = getint();

a[3] = getint();

a[4] = getint();

int cnt;

cnt = 4;

int sum;

sum = 0;

while( cnt > 1 ){

sum = sum + a[cnt];

cnt = cnt - 1;

}

return sum;

}

该部分生成的汇编代码如下所示：

.arch armv7-a

.eabi\_attribute 28, 1

.eabi\_attribute 20, 1

.eabi\_attribute 21, 1

.eabi\_attribute 23, 3

.eabi\_attribute 24, 1

.eabi\_attribute 25, 1

.eabi\_attribute 26, 2

.eabi\_attribute 30, 6

.eabi\_attribute 34, 1

.eabi\_attribute 18, 4

.comm a, 20, 4

.text

.globl main

.arm

main:

push {r4, r5, fp, lr}

mov fp, sp

ldr r8, =#4

sub sp, sp, r8

bl getint

mov r4, r0

ldr r5, =#0

mov r8, r5

lsl r8, r8, #2

ldr r5, .L0

add r5, r8, r5

str r4, [r5]

bl getint

mov r4, r0

ldr r5, =#1

mov r8, r5

lsl r8, r8, #2

ldr r5, .L0

add r5, r8, r5

str r4, [r5]

bl getint

mov r4, r0

ldr r5, =#2

mov r8, r5

lsl r8, r8, #2

ldr r5, .L0

add r5, r8, r5

str r4, [r5]

bl getint

mov r4, r0

ldr r5, =#3

mov r8, r5

lsl r8, r8, #2

ldr r5, .L0

add r5, r8, r5

str r4, [r5]

bl getint

mov r4, r0

ldr r5, =#4

mov r8, r5

lsl r8, r8, #2

ldr r5, .L0

add r5, r8, r5

str r4, [r5]

ldr r8, =#4

sub sp, sp, r8

ldr r4, =#4

str r4, [fp, #-8]

ldr r8, =#4

sub sp, sp, r8

ldr r4, =#0

str r4, [fp, #-12]

b .L2

.L1:

ldr r4, [fp, #-12]

ldr r5, .L0

ldr r6, [fp, #-8]

mov r8, r6

lsl r8, r8, #2

add r5, r5, r8

ldr r5, [r5]

add r4, r4, r5

str r4, [fp, #-12]

ldr r4, [fp, #-8]

ldr r5, =#1

sub r4, r4, r5

str r4, [fp, #-8]

b .L2

.L2:

ldr r4, [fp, #-8]

ldr r5, =#1

cmp r4, r5

bgt .L1

b .L3

.L3:

ldr r4, [fp, #-12]

mov r0, r4

mov sp, fp

pop {r4, r5, fp, pc}

.L0:

.word a

 同理，将代码放在树莓派4B上运行输入1，2，3，4，5，得到结果如图2-3所示。

图2-3 运行结果3

最后我们通过一个较为复杂的堆排序来对编译器进行功能性测试，测试用例heap\_sort.sy如下所示：

int n;

int swap (int array[], int i, int j){

int temp;

temp = array[i];

array[i] = array[j];

array[j] = temp;

return 0;

}

int heap\_ajust(int arr[], int start, int end) {

int dad;

dad = start;

int son;

son = dad \* 2 + 1;

while (son < end + 1) { //

if (son < end && arr[son] < arr[son + 1])

son = son + 1;

if (arr[dad] > arr[son])

return 0;

else {

dad = swap(arr,dad,son);

dad = son;

son = dad \* 2 + 1;

}

}

return 0;

}

int heap\_sort(int arr[], int len) {

int i;

int tmp;

i = len / 2 - 1;

while ( i > -1) {

tmp = len - 1;

tmp = heap\_ajust(arr, i, tmp);

i = i - 1;

}

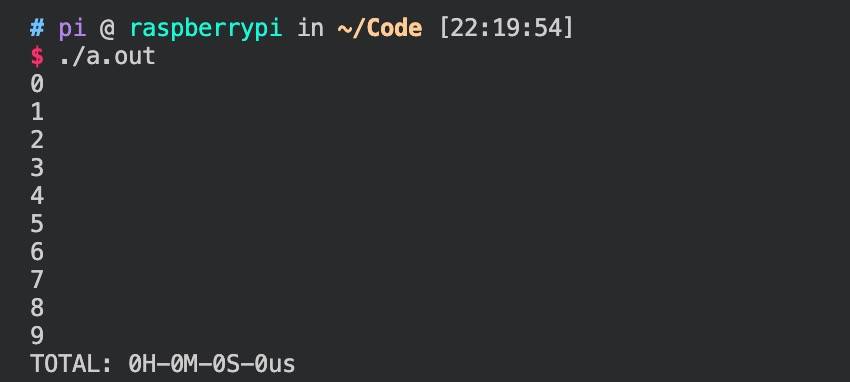
i = len - 1;

while ( i > 0) {

int tmp0;

tmp0 = 0;

tmp = swap(arr,tmp0,i);

 由于生成的汇编代码文件过长，在此不宜展示。生成汇编代码之后我们将其拷贝到树莓派进行运行，得到结果如图2-4所示

tmp = i - 1;

tmp = heap\_ajust(arr, tmp0, tmp);

i = i-1;

}

return 0;

}

int main(){

n = 10;

int a[10];

a[0]=4;a[1]=3;a[2]=9;a[3]=2;a[4]=0;

a[5]=1;a[6]=6;a[7]=5;a[8]=7;a[9]=8;

int i;

i = 0;

i = heap\_sort(a, n);

while (i < n) {

int tmp;

tmp = a[i];

putint(tmp);

tmp = 10;

putch(tmp);

i = i + 1;

}

return 0;

}

图2-4 运行结果4

显然，经过测试项目编译器生成的程序已经能够基本正确完成Sysy2020语言的编译功能了。

# 第三章 知识技能学习情况

## 3.1 编译器前端相关知识

首先使用Flex和Bison进行抽象语法树AST的生成，然后基于访问者设计模式进行从AST到ARM汇编或从AST到中间代码的转换。这个过程使我们初步掌握C++语法，学习了STL的使用，加深了我们对C++工程整体结构的理解，并且可以利用C++编写小型项目并进行编译、链接、调试，生成项目可执行文件。在编码过程中进行了设计模式、代码规范、注释规范等更加高效的开发方法和开发流程规范。学习了在无CG机制的语言中进行内存管理的编码规范，这让我们在今后的编程中减少内存安全问题出现的概率。

另外，在整个编译器项目过程中，我们使用了Git工作流管理项目开发，安排项目进展。并且使用版本管理的特点进行不同时间的版本备份，以供开发中有误时随时回滚。

我们还学习了用LLDB调试C++代码，体会了LLDB与GDB的差异以及LLDB在代码调试上的优越性。

## 3.2 ARM汇编指令的学习

要完成从AST到汇编指令的转换，需要学习arm汇编的相关知识。我们组以ARM v7-M的参考手册为基础，辅以大量博客和MOOC进行了对ARM汇编基本语法的学习。

为了能实际运行ARM汇编程序，我们还学习了使用Raspberry 4B。这样我们可以将自己编写或生成的ARM汇编程序通过汇编链接后，在Raspbian GNU/Linux 10操作系统的Raspberry 4B上运行。比起使用仿真软件，这样能更准确地得到汇编程序的运行时间，调试也更方便。

## 3.3 优化相关知识

为了在源码层面进行预优化，我们学习了常量替换、循环代码优化、条件分支代码优化等内容，为了实现这些优化，又进一步学习了Poly多项式化简算法。

为了对生成的ARM汇编指令进行优化，我们学习了编译原理中的优化内容：公共子表达式、复制传播、死代码消除、代码移动、归纳变量和强度消减等。为了实现这些中间代码上的优化，我们又参考三地址码设计了中间代码。这样可以利用中间代码完成划分基本块、建立流图、进行数据流分析等工作。

# 第四章 分工协作与交流情况

## 4.1 分工协作

从项目功能的完成情况来看，马玏主要负责编译器前端的工作，设计了项目的整体架构，并完成了生成AST和ARM汇编的工作，黎婷完成了生成中间代码的工作。在优化方面，招永骏主要进行了源码层面的预优化，唐龙翔主要进行了中间代码层面的优化和寄存器分配优化工作。

从整个项目的进行过程来看，我们先划分了编写前端和进行优化两大部分工作。在马玏进行前端开发的同时，另外三人进行优化方面的学习，并对各种优化做了大量的实验和讨论。前端开发结束后，由于优化的需要又需要加入中间代码。于是黎婷和马玏继续进行中间代码的生成。唐龙翔和招永骏则各自在前端基础上做相应优化。

## 4.2 团队交流情况

项目进行过程中，我们每周组内都会开会沟通进度和安排下一步工作，并在组间开会和另外一个小组交流部分问题的解决思路，以及听取周尔强老师和廖勇老师的意见。虽然网上交流有一定限制，但我们仍然能保持活跃的沟通，以保证项目的顺利推进。

# 参考文献

1. GNU.Bison Table of Cotent[OL].

https://www.gnu.org/software/bison/manual/bison.html, December 2019

1. GNU. Flex Table of Content[OL]. http://www.gnu.org/software/flex/
2. 周尔强,周帆,韩蒙,陈文宇. 编译技术[M].北京:机械工业出版社, 2019,121-122
3. ARM v7-M Architecture Reference Manual[OL]. https://www.arm.com/
4. Alfred V.Aho, Monica S.Lam, Ravi Sethi, Jeffrey D.Ullnab. 编译原理第2版[M]. 北京:机械工业出版社, 2009.1

# 致谢

本报告所展示的工作是在我们的指导教师周尔强老师、廖勇老师悉心指导下完成的，感谢两位老师每周组间交流时为我们提供技术和方向上的指导；同时感谢全国大学生计算机系统能力大赛-编译系统设计赛（华为毕昇杯）组委会给我们提供了一个展示编译技术和编译优化能力的竞赛平台。