Table of Contents

重庆大学编译原理2022教改项目

实验方案

本次实验将实现一个由 SysY (精简版 C 语言，来自 https://compiler.educg.net/) 翻译至 RISC-V 汇编的编译器，生成的汇编通过 GCC 的汇编器翻译至二进制，最终运行在模拟器 qemu-riscv 上

实验至少包含四个部分: 词法和语法分析、语义分析和中间代码生成、以及目标代码生成，每个部分都依赖前一个部分的结果，逐步构建一个完整编译器

实验一：词法分析和语法分析，将读取源文件中代码并进行分析，输出一颗语法树

实验二：接受一颗语法树，进行语义分析、中间代码生成，输出中间表示 IR (Intermediate Representation)

实验三：根据 IR 翻译成为汇编

实验四(可选)：IR 和汇编层面的优化

实验开始之前

本次实验将从零开始，写一个完整的编译器，编译 SysY 语言至 risc-v 汇编，并在 qemu-riscv 上跑起来

在开始之前，请不要抱有畏惧心理，我们准备好的了详细的文档、代码框架和构建方案、自动化测试程序

还有几个小建议：

1. 在编写代码时，注意代码规范，避免出现低级但难以排查的错误甚至是奇奇怪怪的链接错误，请阅读 C++ coding style 和助教为你准备的 如果不遵守可能会出现各种奇奇怪怪问题的编码小技巧
2. 在遇到 bug 时，请相信计算机一定不会出错，一定是代码的问题
3. 在需要查找资料来解决问题时，请至少使用 Bing 而不是百度，尽量查阅 Google, wikipedia, stackoverflow 等网站的资料，因为与编译原理相关的中文资料可能会非常少，英文资料相对丰富（以助教的经验来说 CSDN 不太能解决问题
4. 上网查资料并不能解决所有的问题，他们的回答甚至有可能时错的！正确的做法应当是阅读官方文档，官方手册包含了查找对象的所有信息，关于查找对象的一切问题都可以在官方手册中找到答案。如果你需要了解如何使用 GDB，你应该阅读Debugging with GDB，或者是你需要了解 riscv 的指令集定义、汇编规则，你应该到 riscv 官网中的技术手册 中寻找答案
5. 在向助教和同学提问之前，请学习 提问的智慧，以更高效的解决问题

实验环境介绍

实验工具

实验至少需要用到以下工具，请同学们自行安装并学习，在下面也有简单的使用说明，这个程度的说明只能保证基础的使用，如果出现不符合预期的情况，请阅读文档

1. Git
2. CMake
3. GNU Makefiles / GCC / GDB

代码框架介绍

获取代码

> 从希冀实验题的附件中下载

可以使用 docker 来避免配置环境的问题，但是仍需从希冀中下载相应实验的框架

```
docker使用

拉取镜像：
docker pull frankd35/demo:v3

挂载目录至 /coursegrader 并使用：
docker run -it -v {你的代码框架目录}:/coursegrader frankd35/demo:v3

在使用 docker 里面的 cmake 时会遇到一个问题，因为安装了交叉编译器以及 cmake 不知道怎么把 riscv 的交叉编译器配置成了默认编译器，所以编出来的可执行文件是 riscv 版的。要解决这个问题需要在 CMakeLists.txt 中指定 x86-linux 的编译器：
    set(CMAKE_C_COMPILER    "/usr/bin/x86_64-linux-gnu-gcc-7")
    set(CMAKE_CXX_COMPILER  "/usr/bin/x86_64-linux-gnu-g++-7")
```

因为我们希望以库的形式提供 IR 测评相关的函数实现，所以请在 /lib 下根据你使用的 linux 或 windows 平台将对应的库文件重命名为 libxx.a 才能通过编译

测评程序

在测评时 你应该提交一个直接包含至少 /include /src 的压缩包

我们的测评程序会解压你的压缩包，并将 CMakeLists.txt 和 main.cpp 复制到同一目录下（会覆盖你的文件），并链接合适的 libxx.a，使用 /test/test.py [s1/s2/S] 来进行测评, 所以请严格按照实验指导书指定的接口来完成实验（最近简单的方式是不要修改 CMakeLists.txt 和 main.cpp

```
目录结构：
--- dir
    --- {你的压缩包解压出来的文件}
    --- CMakeLists.txt
    --- main.cpp
```

目录结构

```
/bin        可执行文件 + 库文件

/build        构建项目的文件

/include    头文件

/src        源代码

/src/...    子模块: IR, frontend, backend, opt
            third_party: 第三方库, 目前使用了 jsoncpp 用于生成和读取 json

/lib        我们提供的库文件, 你需要根据你使用的 linux 或 windows 平台将对应的库文件重命名为 libIR.a 才能通过编译


/test       测试框架, 可以用于自测

/CMakeList.txt

/readme.txt
```

编译

```
首先进入 /build 若CMakeList修改后应执行 cmake 命令
1. cd /build
2. cmake .. （如果是在 windows 环境下第一次支持 cmake 命令，需要使用 'cmake -G "MinGW Make files" ..' 以构建 makefile）
如果一切正常没有报错 执行make命令
3. make
```

执行

```
1. cd /bin
2. compiler <src_filename> [-step] -o <output_filename> [-O1]
    -step: 支持以下几种输入
        s0: 词法结果 token 串
        s1: 语法分析结果语法树, 以 json 格式输出
```

```
        s2: 语义分析结果, 以 IR 程序形式输出
        -S: RISC-v 汇编
```

测试

```
1. cd /test
2. python [files]:
    build.py:   进入到 build 目录, 执行 cmake .. & make
    run.py: 运行可执行文件 compiler 编译所有测试用例, 打印 compiler 返回值和报错, 输出编译结果至 /test/output
        执行方法: python run.py [s0/s1/s2/S]
    score.py: 将 run.py 生成的编译结果与标准结果进行对比并打分
        执行方法: python score.py [s0/s1/s2/S]

    test.py 编译生成 compiler 可执行文件，执行并生成结果，最后对结果进行判断并打分，结果将出现在标准输出的最后一行
        执行方法: python test.py [s0/s1/s2/S]
```

Windows 环境准备

在 Windows 下至少可以完成实验一和实验二的部分，实验三部分需要在 linux 运行 qemu-riscv 来完成

为了在 windows 下运行 CMake 及 Makefiles，同学们需要安装

> - CMake is an open-source, cross-platform family of tools designed to build, test and package software
> - Mingw A native Windows port of the GNU Compiler Collection (GCC), with freely distributable import libraries and header files for building native Windows applications

在测试中我们还用到了 `diff` 工具，Windows 原生是不支持的，这个工具在安装了 Git 之后，在 {ur_path_to_git}/Git/usr/bin 中可以找到 diff.exe，所以我们需要将这个目录添加到 环境变量 **PATH** 下，才能正常使用测评功能

> 完成了以上步骤以后，在 cmd 或 powershell 里敲出 cmake, make, diff 命令时，响应不应该为 **xxx** 不是内部或外部命令，也不是可运行的程序 或批处理文件。
>
> 如果安装了 Mingw 后仍然不支持 make, 请检查 gcc 命令在 command 里面是否支持，如果支持的话说明你的 Mingw 是安装好了的，那么请你到 {ur_path_to_Mingw}/bin 下面找一下有没有一个叫 mingw64-make.exe 的程序（应该是有的），把他的名字改成 make.exe，重启 command 就可以支持 make 命令了

Mac 环境准备

可以完成实验一和实验二的部分，实验三部分需要使用 qemu-riscv 来完成，建议使用我们提供的 docker 至少需要支持 cmake, git，GNU Makefiles, gcc, diff 等工具

Linux 环境准备

部分 linux 环境不是原生支持 CMake 的，安装 CMake 即可，实验三部分需要使用 qemu-riscv 来完成，建议使用我们提供的 docker

实验一

实验目标

实验一将实现编译器前端的词法分析和语法分析部分，目标是分析输入的 源文件 得到一颗 抽象语法树

实验步骤

从希冀上下载实验框架

```
4.29 日对测试用例进行了更新
```

实验一标准输出

这是一段最简单的 SysY 程序

```
int main() {
    return 3;
}
```

实验一将把他解析为一颗语法分析树，我们用 json 来输出语法树，标准如下：

```
{
   "name" : "CompUnit",
   "subtree" : [
      {
         "name" : "FuncDef",
         "subtree" : [
            {
               "name" : "FuncType",
               "subtree" : [
                  {
                     "name" : "Terminal",
                     "type" : "INTTK",
                     "value" : "int"
                  }
               ]
            },
            {
               "name" : "Terminal",
               "type" : "IDENFR",
               "value" : "main"
            },
            {
```

```
                "name" : "Terminal",
                "type" : "LPARENT",
                "value" : "("
             },
             {
                "name" : "Terminal",
                "type" : "RPARENT",
                "value" : ")"
             },
             {
                "name" : "Block",
                "subtree" : [
                   {
                      "name" : "Terminal",
                      "type" : "LBRACE",
                      "value" : "{"
                   },
                   {
                      "name" : "BlockItem",
                      "subtree" : [
                         {
                            "name" : "Stmt",
                            "subtree" : [
                               {
                                  "name" : "Terminal",
                                  "type" : "RETURNTK",
                                  "value" : "return"
                               },
                               {
                                  "name" : "Exp",
                                  "subtree" : [
                                     {
                                        "name" : "AddExp",
                                        "subtree" : [
                                           {
                                              "name" : "MulExp",
                                              "subtree" : [
                                                 {
                                                    "name" : "UnaryExp",
                                                    "subtree" : [
                                                       {
                                                          "name" : "PrimaryExp",
                                                          "subtree" : [
                                                             {
                                                                "name" : "Number",
                                                                "subtree" : [
                                                                   {
                                                                      "name" : "Term
inal",
                                                                      "type" : "INTL
TR",
```

```
                                                                "value" : "3"
                                                             }
                                                          ]
                                                       }
                                                    ]
                                                 }
                                              ]
                                           }
                                        ]
                                     }
                                  ]
                               }
                            ]
                         }
                      ]
                   },
                   {
                      "name" : "Terminal",
                      "type" : "SEMICN",
                      "value" : ";"
                   }
                ]
             }
          ]
       },
       {
          "name" : "Terminal",
          "type" : "RBRACE",
          "value" : "}"
       }
    ]
 }
 ]
}
```

词法分析

词法分析的目的是读入外部的字符流（源程序）对其进行扫描，把它们组成有意义的词素序列，对于每个词素，词法分析器都会产生词法单元(**Token**) 作为输出

1. Token

Token 的定义在 token.h 中，同时 Token 类型的枚举类 **TokenType** 也定义在其中

```
struct Token {
    TokenType type;
    string value;
};

enum class TokenType{
    IDENFR,          // identifier
    INTLTR,             // int literal
    FLOATLTR,         // float literal
    CONSTTK,         // const
    VOIDTK,             // void
    ...
}
```

其中 **string** value 是 Token 所代表的字符串， **TokenType** type 是指 Token 的类型

2. DFA

在词法分析中，我们使用确定有限状态自动机 (deterministic finite automaton, DFA) 来进行分词，对于一个给定的属于该自动机的状态和一个属于该自动机字母表 Σ 的字符，它都能根据事先给定的 转移函数 转移到下一个状态，某些转移函数会进行输出

我们需要为词法分析设计这样一个 DFA：它可以接收输入字符，进行状态改变，并在某些转移过程中输出累计接受到的字符所组成的字符串

该 DFA 中应存在五种状态，我们用枚举类 **State** 来表示

```
enum class State {
    Empty,              // space, \n, \r ...
    Ident,              // a keyword or identifier, like 'int' 'a0' 'else' ...
    IntLiteral,         // int literal, like '1' '1900', only in decimal
    FloatLiteral,       // float literal, like '0.1'
    op                  // operators and '{', '[', '(', ',' ...
};
```

我们将 DFA 及其行为的抽象为类和类方法，定义在 lexical.h 中

```
struct DFA {
    /**
     * @brief take a char as input, change state to next state, and output a Token
if necessary
     * @param[in] input: the input character
     * @param[out] buf: the output Token buffer
     * @return  return true if a Token is produced, the buf is valid then
     */
    bool next(char input, Token& buf);

    /**
     * @brief reset the DFA state to begin
     */
    void reset();

private:
    State cur_state;    // record current state of the DFA
    string cur_str;    // record input characters
};
```

其中 **State** cur_state 记录 DFA 当前的状态，**string** cur_str 记录 DFA 已经接受的字符串

每次字符输入都应调用 `bool next(char input, Token& buf);` 该函数是实现 DFA 的核心，即 转移函数 ，其根据自身当前状态和输入来决定转移后的状态，如果产生 Token 则返回 true

TODO: 实现 DFA 中的 bool next(char input, Token& buf) 函数和 void reset(); 函数

3. Scanner

Scanner 是扫描器，其职责是将字符串输入转化为 Token 串，词法分析实际上就是实现一个 Scanner

Scanner 的定义在 lexical.h 中

```
struct Scanner {
    /**
     * @brief constructor
     * @param[in] filename: the input file
     */
    Scanner(std::string filename);

    /**
     * @brief run the scanner, analysis the input file and result a token stream
     * @return std::vector<Token>: the result token stream
     */
    std::vector<Token> run();

private:
    std::ifstream fin;  // the input file
};
```

其中 **std::ifstream** fin 是源文件打开得到的文件流

`std::vector<Token> run();` 将执行分析过程，从文件流中读取字符，实例化一个 DFA 对象来获取 Token，但是我们的 DFA 只能识别代码部分，源文件中的注释输入 DFA 后并不能被正确的处理，我们需要在使用 DFA 接受字符串前对字符串进行预处理，所以 run() 的伪代码如下：

```
vector<Token> ret;
Token tk;
DFA dfa;
string s = preproccess(fin);    // delete comments
for(auto c: s) {
        if(dfa.next(c, tk)){
            ret.push_back(tk);
        }
}
```

TODO: 实现 Scanner 的 std::vector run() 函数

我们用一段最简单的 SysY 程序来展示词法分析的结果

```
int main() {
    return 3;
}
```

结果如下

TokenType	Value
INTTK	int
IDENFR	main
LPARENT	(
RPARENT	)
LBRACE	{
RETURNTK	return
INTLTR	3
SEMICN	;
RBRACE	}

词法分析部分到此结束

语法分析

1. SysY 文法

我们对 SysY 文法进行了一定的限制以减少难度，主要改变是同学们不需要支持二维以上的数组解析、不需要支持各种形式的浮点数字面量解析(不需要支持即我们在测试中不会出现这样的用例)，并对左递归文法做了处理。新的文法请参考 文法定义。请注意，实现必须以该文法为准

2. 抽象语法树

抽象语法树(abstract syntax tree, AST) 是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构

我们知道文法中的一个 产生式 可以化为树的形式，如 `FuncDef -> FuncType Ident '(' [FuncFParams] ')' Block` 可以展开为以 **FuncDef** 为父节点， `FuncType` `Ident` `'('` `[FuncFParams]` `')'` `Block` 为从左至右子节点的一颗多叉树

为了之后语义分析的实现方便，我们为每一个非终结符定义一个类，以便不同的树节点可以拥有他们自己的成员变量，他们拥有共同的基类 **AstNode**，定义在 abstract_syntax_tree.h 中

```
struct AstNode{
    NodeType type;  // the node type
    AstNode* parent;    // the parent node
    vector<AstNode*> children;     // children of node

    /**
     * @brief Get the json output object
     * @param root: a Json::Value buffer, should be initialized before calling this
 function
     */
    void get_json_output(Json::Value& root) const;
}

struct CompUnit: AstNode {
struct Decl: AstNode{
struct FuncDef: AstNode{
struct ConstDecl: AstNode {
...
```

其中 **NodeType** 代表了树节点的类型，每个非终结符有自己的 NodeType，所有的终结符认为式 Terminal 类型，NodeType 枚举类定义在abstract_syntax_tree.h 中

```
enum class NodeType {
    TERMINAL,       // terminal lexical unit
    COMPUINT,
    DECL,
```

```
    FUNCDEF,
    ...
}
```

`void get_json_output(Json::Value& root) const;` 可以实现语法树的输出，我们已经提供了实现，请不要改动它，除非你能保证改动后输出仍与我们提供的标准输出一致

3. Parser

解析器(Parser) 一般是指指把某种格式的文本（字符串）转换成某种数据结构的程序，在我们的语法分析过程中，Parser 接收 Token 串，转化为 AST，它被定义在 syntax.h

```
struct Parser {
    uint32_t index; // current token index
    const std::vector<Token>& token_stream;

    /**
     * @brief creat the abstract syntax tree
     * @return the root of abstract syntax tree
     */
    CompUnit* get_abstract_syntax_tree();
}
```

其中 **uint32_t index** 记录了已分析 Token 串的位置，**vector& token_stream** 是输入的 Token 串，`CompUnit* get_abstract_syntax_tree();` 是提供给外部的接口，返回一颗分析完成的语法树

TODO: 实现 CompUnit* get_abstract_syntax_tree(); 函数

4. 递归下降法实现

语法分析有很多种方式，在此我们介绍用递归下降法实现 LL(k) 分析的方法，但不要求同学们必须使用该方法，实现 **get_abstract_syntax_tree** 接口即可

在递归下降法中，我们首先要解决 左递归 问题：在某些文法，特别是表达式相关的文法中存在形如 `AddExp -> MulExp | AddExp ('+' | '-') MulExp` 的产生式，我们需要进行左递归的消除才能使用递归下降法，消除后产生式形为

```
AddExp -> MulExp [ ('+' | '-') AddExp ]
```

但是这个产生式在语义分析中并不好用，同学们可以思考一下为什么不对 (提示: 运算顺序)

于是经过进一步的变换，这个表达式可以变成我们实验规定的文法的样子，并且这三个产生式所产生的语言是等价的

```
AddExp -> MulExp { ('+' | '-') MulExp }
```

实现递归下降法，我们需要为每个非终结符节点创建一个 parse 函数

```
struct Parser {
    /**
     * @brief recursive descent functions for non-terminals
     * @param root: current parsing non-terminal
     * @return true: 为了程序通用性 对 [] 包括起来的非终结符我们一样会调用这个递归下降函数, bool 返回值用来判断是否为空
     */
    bool parseCompUint(CompUnit* root);
    bool parseDecl(Decl* root);
    bool parseConstDecl(ConstDecl* root);
    bool parseBType(BType* root);
    bool parseConstDef(ConstDef* root);
    ...
}
```

在 parse 函数中，我们根据下一个需要处理的 token 类型和该节点不同产生式的 first 集来选择处理哪一个产生式；在处理产生式时，应该按顺序从左到右依次处理，对非终结符调用其相应的 parse 函数，并将得到的语法树节点加入该节点的子节点中；对终结符，我们使用一个特殊的 parse 函数 `Term* parseTerm(AstNode* parent, TokenType expected);` 来处理，其功能是

1. 判断当前 index 所指的 Token 是否为产生式所要求的 Token 类型，如果不是则发生了错误，程序运行结果则不可预计
2. 如果是符合预期的 Token 类型
    i. a. 则 new 一个 Term 节点并将 Token 内容拷贝到节点中
    ii. 将该节点加入 parent 的子节点
    iii. Parser 的 index++

```
struct Term: AstNode {
    Token token;

    /**
     * @brief constructor
     */
    Term(Token t, AstNode* p = nullptr);
};

struct Parser {
    /**
     * @brief parse a specific type terminal token
     * @param expected: the specific type
     * @return Term*
     */
    Term* parseTerm(AstNode* parent, TokenType expected);
}
```

> TIPS：这里建议大家对任何不符合预期的地方直接 assert(0)，因为我们保证提供的源程序符合文法，所以如果不符合预期一定是你自己写错了

对于 parse 函数的三种最基本的操作，我们提供了宏和使用方法供同学们参考

1. 根据下一个 Token 类型的类型选择处理的产生式(一般只看下一个 Token 就可以选择产生式，少数情况下多个产生式的 first 集有交集时，应多向后看几个 Token)

```
 #define CUR_TOKEN_IS(tk_type) (token_stream[index].type == TokenType::tk_type)

if (CUR_TOKEN_IS(tk_type1)) {
     // code
}
else if (CUR_TOKEN_IS(tk_type2)) {
     // code
}
...
```

2. 如果是非终结符，则调用其 pasrse 函数，并将其挂在 root 节点上

```
 #define PARSE(name, type) auto name = new type(root); assert(parse##type(name)
); root->children.push_back(name);

 PASER(exp_node, Exp);    // create a Exp node: exp_node, parse it, and add it
to root.children
```

3. 如果是终结符，则调用 pasrseTerm 函数，并将其挂在 root 节点上

```
 #define PARSE_TOKEN(tk_type) root->children.push_back(parseTerm(root, TokenTyp
e::tk_type))

 PARSE_TOKEN(CONSTTK);    // parse root's first child as Exp, and add it to roo
t.children
```

我们还为大家准备了用于 debug 的 log 函数，只需要在每个 parse 函数的头部加上它，就可以监视到你的解析程序的执行过程

```
void Parser::log(AstNode* node){
        std::cout << "in parse" << toString(node->type) << ", cur_token_type::" <<
toString(token_stream[index].type) << ", token_val::" << token_stream[index].value
<< '\n';
}
```

我们仍然以 `FuncDef -> FuncType Ident '(' [FuncFParams] ')' Block` 为例，展示 parseFuncDef 函数的实现

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

```
bool Parser::parseFuncDef(FuncDef* root) {
    log(root);

    PARSE(functype, FuncType);
    PARSE_TOKEN(IDENFR);
    PARSE_TOKEN(LPARENT);

    // no [FuncFParams], FuncType Ident '(' ')' Block
    if(CUR_TOKEN_IS(RPARENT)) {
        PARSE_TOKEN(RPARENT);
    }
    // FuncType Ident '(' FuncFParams ')' Block
    else {
        PARSE(node, FuncFParams);
        PARSE_TOKEN(RPARENT);
    }

    PARSE(block, Block);
    return true;
}
```

看到这里，你应该可以自己用递归下降法实现语法分析程序了，请不要忘记最终提供的接口是 **get_abstract_syntax_tree**

文法定义

对于实验一 只需要关注第一行的文法

对于实验二 在文法之外我们还提供了语法树中属性的参考定义, 但这不是强制要求的 ###FIXME 更详细的描述

Extended Backus-NaurForm

SysY 语言的文法采用扩展的 Backus 范式（EBNF，Extended Backus-NaurForm）表示，其中：

```
符号[...]表示方括号内包含的为可选项；

符号{...}表示花括号内包含的为可重复 0 次或多次的项；

终结符或者是单引号括起的串，或者是 Ident、InstConst、floatConst 这样的记号
```

文法规则

CompUnit -> (Decl | FuncDef) [CompUnit]

Decl -> ConstDecl | VarDecl

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

```
ConstDecl.t
```

BType -> 'int' | 'float'

```
BType.t
```

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

```
ConstDef.arr_name
```

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

```
ConstInitVal.v
ConstInitVal.t
```

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

```
VarDecl.t
```

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

```
VarDef.arr_name
```

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

```
InitVal.is_computable
InitVal.v
InitVal.t
```

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

```
FuncDef.t
FuncDef.n
```

FuncType -> 'void' | 'int' | 'float'

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

FuncFParams -> FuncFParam { ',' FuncFParam }

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

BlockItem -> Decl | Stmt

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

Exp -> AddExp

```
Exp.is_computable
Exp.v
Exp.t
```

Cond -> LOrExp

```
Cond.is_computable
Cond.v
Cond.t
```

LVal -> Ident {'[' Exp ']'}

```
LVal.is_computable
LVal.v
LVal.t
LVal.i
```

Number -> IntConst | floatConst

PrimaryExp -> '(' Exp ')' | LVal | Number

```
PrimaryExp.is_computable
PrimaryExp.v
PrimaryExp.t
```

UnaryExp -> PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp

```
UnaryExp.is_computable
UnaryExp.v
UnaryExp.t
```

UnaryOp -> '+' | '-' | '!'

FuncRParams -> Exp { ',' Exp }

MulExp -> UnaryExp { ('*' | '/' | '%') UnaryExp }

```
MulExp.is_computable
MulExp.v
MulExp.t
```

AddExp -> MulExp { ('+' | '-') MulExp }

```
AddExp.is_computable
AddExp.v
AddExp.t
```

RelExp -> AddExp { ('<' | '>' | '<=' | '>=') AddExp }

```
RelExp.is_computable
RelExp.v
RelExp.t
```

EqExp -> RelExp { ('==' | '!=') RelExp }

```
EqExp.is_computable
EqExp.v
EqExp.t
```

LAndExp -> EqExp ['&&' LAndExp]

```
LAndExp.is_computable
LAndExp.v
```

```
LAndExp.t
```

LOrExp -> LAndExp ['||' LOrExp]

```
LOrExp.is_computable
LOrExp.v
LOrExp.t
```

ConstExp -> AddExp

```
ConstExp.is_computable: true
ConstExp.v
ConstExp.t
```

实验二

实验目标

由实验一的 抽象语法树 经过语义分析和语法制导翻译，生成中间表示 **IR**

实验步骤

从希冀上下载实验框架

compiler [src_filename] -s2 -o [output_filename] 将输出你的 IR 程序至 [output_filename]

> 用于观察自己的 IR 是否生成正确

compiler [src_filename] -e -o [output_filename] 将执行你的 IR 程序并输出其结果到 [output_filename]

> 包括源程序调用 putint 等函数输出到标准输出的内容 以及将程序 main 函数的返回值打印到最后一行，这个才是用于测试比对的

```
5.12 日对测试用例和测评机进行了更新
```

[src_filename] 是 SysY 源程序，[output_filename] 是输出文件，根据参数不同而输出不同

测评方法：

在中间代码的测评中，考虑到相同的程序可以用不同的 **IR** 序列来表示，我们设计了 **IR 测评机** 来执行 **IR** 代码，通过测评机执行 **IR** 序列的结果判断 **IR** 序列是否正确

如果要自测，我们对自测的输入文件命名有一点要求，请务必保证以下规则（main.cpp 告诉了你为啥会有这样的限制）：

使用 **compiler [filename.sy] -e -o [output_filename]** 命令读入一个后缀为 .sy 的源文件，如果该源文件需要输入的话（如调用了 getint getch 等库函数）请将输入放到 [filename.in] 中，

实验二标准输出

这是一段简单的 SysY 程序

```
int main() {
    putint(100); putch(32); putch(97);
    return 3;
}
```

执行 -e 选项， [output_filename] 中输出如下

```
100 a
3
```

其中，第一行是程序本身想打印到标准输出的内容，最后一行是 main 函数的返回值，值得注意的是 **SysY** 程序 **main** 函数返回值会以 **uint8** 的形式进行输出，main.cpp 也保证了这一点

使用参数 -s2 可以调用 draw() 对 IR 程序进行输出，你可以通过它来查看你生成的 IR：

```
int main()
        0: call t0, putint(100)
        1: call t1, putch(32)
        2: call t2, putch(97)
        3: return 3
end

GVT:
```

什么是IR

IR（Intermediate Representation）为中间表示。在编译系统中，编译器前端对高级编程语言（源程序）进行词法分析、语法分析、语义分析后会生成高层次 IR，在中端对高层次 IR 逐步降低到低层次 IR，并对其进行优化。后端根据降低优化后的 IR 生成目标架构汇编指令。

低层次抽象的 IR

我们设计了一层形如（**opcode**，**des**，**operand1**，**operand2**）的 IR，其中 **opcode** 代表 IR 的种类决定了 IR 功能，**des** 是目的操作数，**operand1** & **operand2** 是源操作数

为了后端的实现简化，我们设计的 IR 更接近于汇编的层次，如

- 变量赋值IR
- 算术运算IR
- 逻辑运算IR
- 访存运算IR
- 类型转化IR
- 跳转IR

与 risc-v 汇编中对应算术运算、逻辑运算、访存、类型转换、跳转指令的含义基本相同。

同时为了屏蔽后端实现上的细节，我们为实现变量定义、函数调用与返回、指针运算等功能设计了以下几种 IR：

- 变量定义IR
- 调用返回IR
- 指针运算IR

具体功能可以参考 IR 定义

抽象计算机模型

组成原理告诉我们计算机由五个部分组成：运算器、控制器、存储器、输入设备、输出设备。

我们的 IR 可以视作在一个特定计算机模型上的一个指令系统，IR 可以在我们定义的计算机模型上完成计算机的所有功能，暂且称之为 抽象计算机模型 吧！我们还提供了一个软件来实现该计算机模型，同学们生成的 IR 将运行在这个软件上。

执行

在抽象计算机模型中，控制器、运算器以 IR 为最小单位执行运算，操作数直接从存储中获取，每次执行一条 IR。根据 IR 的具体功能，可能会将运算器的结果写回存储器中，即写回目的操作数；也可能直接导致程序执行流的改变，即跳转 IR 或函数调用与返回 IR。

操作数

在抽象计算机模型中有两种不同的运算器，浮点运算器和整型运算器，他们只能接受对应类型的操作数进行运算，所以操作数也被分为了整型和浮点型操作数，值得一提的是我们认为指针是整型的一种。

整型操作数可以分为：整型变量、整型立即数、指向整型操作数的指针、指向浮点操作数的指针

浮点操作数可以分为：浮点变量、浮点立即数

变量和指针操作数的值实际存放在抽象计算机模型的存储中，以 `Operand.name` 为唯一标识符在存储中查找得到。指针可以指向一片连续的空间用于存放数组，这一片空间只能通过 load & store IR 与基址指针来操作

> 注：
>
> 源操作数必须是已经存在存储中的操作数，即可以通过操作数名称找到的，否则接下来的行为是未定义的
>
> 目的操作数如果不存在存储中，我们认为这是一个新的变量，并为其申请空间，如果存在则直接更新其在存储中的值
>
> 如果在存储中找到两个同名的操作数，接下来的行为是未定义的
>
> IR 是类型严格的，如果操作数类型与 **opcode** 指定的类型不符合，接下来的行为是未定义的

上下文

在抽象计算机模型中运行的程序仍具有函数、全局变量、局部变量的概念。**IR** 程序的数据结构定义如下：

```
struct Program {
    std::vector<Function> functions;
    std::vector<GlobalVal> globalVal;
    Program();
    void addFunction(const Function& proc);
    std::string draw();
};
```

可以看到一个 IR 程序可以拥有多个函数、全局变量，其入口应该为 **main** 函数。每个函数在运行时，拥有自己的局部变量，也就是存放在在内存中的操作数，同时可以访问全局变量。函数在运行时的局部变量、运行位置即为上下文，在 IR 程序的运行过程中，抽象计算机模型维护了一个函数调用栈，发生函数调用时，会发生上下文的切换，原来的上下文将压入函数调用栈中，函数返回时，函数调用栈会将当前上下文弹出

上下文切换的具体含义是指：发生函数调用时，当前 IR 执行位置和存储中的操作数将被保存下来，执行流切换到下一个函数。新的函数在执行时 IR 将从第一条开始执行、存储中没有任何操作数。当从该函数返回时，将回到当初发生函数的调用的位置继续执行，存储中的操作数恢复为发生调用前保存的操作数

输入输出

我们没有提供输入输出相关的 IR，可以通过调用库函数来与标准输入输出进行交互，例如：

```
int a = getint();
putch(a);
```

在 IR 中被直接翻译为函数调用：

```
1: call a, getint
2: call null, putch(a)
```

ir::Program 的 functions vector 中不应包含与库函数同名的函数，我们的 IR 测评机会对库函数进行特殊处理

IR 测评机

IR 测评机的源码已经发放给大家了，结合文档和源码可以对 IR 测评机有一个更深入的认识，有任何问题都可以通过阅读源码来解决；接下来对其设计进行简要介绍

操作数与值

以下代码为操作数定义了一种数据结构 ir::Value，包括了操作数的值和类型

```
union _4bytes {
    int32_t ival;
    float   fval;
    int*    iptr;
    float*  fptr;
};

// definition of value of a operand in memory
struct Value {
    Type t;
    _4bytes _val;
};
```

函数与上下文

ir::Context 为一个执行中的 ir::Function 保存了其需要的基本数据包括

```
- pc            : 当前执行的指令的位置
- retval_addr   : 可能的返回值地址，如果该指针不为空的话就在 return IR 中写该地址
- mem           : 该函数运行过程中的操作数以及其对应的值
- pfunc         : 指向被指向的 ir::Function 的指针，用于获取 IR 指令等等
```

```
// definition of function context
struct Context {
    uint32_t pc;                            // program counter of a function
    Value* retval_addr;                    // if it's not nullptr, this addr will be
 written when exit a context,
    std::map<std::string, Value> mem;
    const ir::Function* pfunc;              // executing which function

    /**
     * @brief constructor
     */
    Context(const ir::Function*);
};
```

全局变量

值得单独一提的是全局变量，在设计过程中，为了实验三翻译为汇编更加方便，全局变量必须通过 ir::Program 中的 `std::vector<GlobalVal> globalVal` 来传递

IR 测评机会根据 GlobalVal 的 maxlen 字段和本身的类型为该全局变量申请一片初始化为零的空间(类似于汇编中的 .space 伪指令)，如果是数组则分配一个 maxlen 长度的全 0 数组，如果是整形或者浮点型则初始化操作数的值为 0；并将该操作数放到 ir::Executor 中的 global_vars 中

所以如果全局变量的初始值不为 **0** 时，你需要自己在 **main** 函数的一开始（或其他你觉得合适的地方）为这些全局变量赋值，一个参考的实现可以见 例子

IR 测评机

IR 测评机实际上是去执行一个 ir::Program，其成员变量包含了执行中必须储存的一些数据结构，如

```
- cxt_stack : 函数执行栈
- cur_ctx   : 当前执行的函数上下文
- global_vars : 全局变量表
    ...
```

提供了 `int run();` 函数从 main 函数开始来执行整个 ir::Program，并返回 main 函数的返回值

```
struct Executor {
    const ir::Program* program;
    std::map<std::string, Value> global_vars;

    Context* cur_ctx;
    std::stack<Context*> cxt_stack;

    /**
     * @brief constructor
     */
```

```
    Executor(const ir::Program*, std::ostream& os = std::cout);

    /**
     * @brief execute the ir program and return its main function's return value
     * @return int: the main function's return value
     */
    int run();

    /**
     * @brief execute next n IRs
     * @return true : execute without error occurs
     * @return false: sth bad happens
     */
    bool exec_ir(size_t n = 1);
}
```

IR 库

实验中助教们会提供由 IR 相关实现打包成的静态库与 IR 相关的 .h 文件，为了避免部分同学可能会因修改 .cpp 文件导致测评不通过，请不要修改 **IR** 库或者是 **IR** 测评机相关实现，因为线上测评使用我们提供的静态库去链接，你的修改并不会生效

ir.h

该文件对 IR 库中所有头文件进行了引入，同学们使用中只需引入该头文件即可

```
#ifndef IR_ALL_H
#define IR_ALL_H

#include"ir_operand.h"
#include"ir_operator.h"
#include"ir_instruction.h"
#include"ir_function.h"
#include"ir_program.h"

#endif
```

ir_operand.h

该文件中是对 IR 操作数的封装定义。我们对IR操作数进行类封装，将其视为具有name、type属性的复杂数据类型。这样将操作数绑定类型对于后续中端优化、后端处理等非常方便（毕竟类型系统也是语义分析一个重点）。

```
struct Operand {
    std::string name;
    Type type;
    Operand(std::string = "null", Type = Type::null);
};
```

该文件对所有可能操作数类型 **Type** 进行枚举定义。这里的类型与 C 或 C++ 标准类型略有不同，如下所示：

```
enum class Type {
    Int,
    Float,
    IntLiteral,
    FloatLiteral,
    IntPtr,
    FloatPtr,
    String,
```

```
    null
};
std::string toString(Type t);
```

其中Int、Float为整型、浮点型变量，IntLiteral、FloatLiteral 为立即数整型、立即数浮点型，IntPtr、FloatPtr 为整型指针、浮点型指针（对于指针的支持将在拓展实验中供同学们选择），当函数的返回值为 void 时，我们提供了特殊的 null 类型。我们还提供了 `string toString(Type t)` 函数来打印 Type。

ir_operator.h

对IR操作符进行定义，以枚举类的形式。与 Type 的 toString 函数类似，接受一个操作符的枚举类型，返回对应字符串形式。各操作符具体含义可在 IR定义 指导书中查阅。

```
enum class Operator {
    _return,    // return   op1
    _goto,      // goto     [op1=cond_var/null],    des = offset
    call,       // call     op1 = func_name,    des = retval  /* func.name = functi
on, func.type = return type*/
    // alloc [arr_size]*4 byte space on stack for array named [arr_name], do not us
e this for global arrays
    alloc,      // alloc    op1 = arr_size,     des = arr_name
    store,      // store    des,    op1,    op2    op2为下标 -> 偏移量  op1为 store 的
数组名, des 为被存储的变量
    load,       // load     des,    op1,    op2    op2为下标 -> 偏移量  op1为 load 的
数组名, des 为被赋值变量
    getptr,     // op1: arr_name, op2: arr_off

    def,
    fdef,
    mov,
    fmov,
    cvt_i2f,    // convert [Int]op1 to [Float]des
    cvt_f2i,    // convert [Float]op1 to [Int]des
    add,
    addi,
    fadd,
    ...
}
```

ir_instruction.h

`struct Instruction` 是 IR 指令的基类定义。成员变量包括 Operand 类型的两个源操作数与结果操作数以及 Operator 类型的操作符，四者也是四元式形式IR的组成。成员函数包括无参构造函数和全参构造函数，具体使用可参考ir_example.cpp中代码示例。

```
struct Instruction {
    Operand op1;
    Operand op2;
    Operand des;
    Operator op;
    Instruction();
    Instruction(const Operand& op1, const Operand& op2, const Operand& des, const O
perator& op);
    virtual std::string draw() const;
};
```

`virtual std::string draw() const;` 定义了各类型指令输出格式，并以字符串形式返回

由于函数调用指令较为特殊，需额外传入函数调用实参，这里对其进行额外定义。`Struct CallInst` 在继承基类 Instruction 的基础上多了argumentList成员变量，用于存入函数调用实参。该类中也对Instruction基类中draw方法进行重写。

```
struct CallInst: public Instruction{
    std::vector<Operand> argumentList;
    CallInst(const Operand& op1, std::vector<Operand> paraList, const Operand& des)
;
    CallInst(const Operand& op1, const Operand& des);   //无参数情况
    std::string draw() const;
};
```

ir_function.h

对函数块的定义，实质上是用于添加存放输入源程序中某个函数生成的IR指令。对各成员变量的说明如下：

- name：函数块名称，可以直接将源程序中函数名作为name。对于全局生成的IR指令，存入的函数块名称在不冲突情况下可简单命名为“global”（这里只是助教举的一个例子~）
- returnType：函数返回类型，即对应源程序中函数的返回类型。对于全局以及void类型，`ir::Type` 中的 `null` 就派上用场了。
- ParameterList：函数形参列表。该列表可以为空（无形参情况）。列表中元素为Operand，意味着传入的形参要连带类型进行封装处理。
- InstVec：函数对应的IR指令。

```
struct Function {
    std::string name;
    ir::Type returnType;
    std::vector<Operand> ParameterList;
    std::vector<Instruction*> InstVec;
    Function();
    Function(const std::string&, const ir::Type&);
    Function(const std::string&, const std::vector<Operand>&, const ir::Type&);
    void addInst(Instruction* inst);
```

```
    std::string draw();
};
```

`void addInst(Instruction* inst);` 用于函数块初始化后向其中添加IR指令。

`std::string draw();` 函数块的输出方式定义，除调用 InstVec 中各指令的 draw 方法外，还定义了函数名、返回类型、函数形参的输出格式。

ir_program.h

对程序体的定义，实质上是用于添加存放上述函数块，一个输入源程序即对应一个程序体，该源程序中生成的所有IR指令均在程序体中存放。

```
struct Program {
    std::vector<Function> functions;
    std::vector<GlobalVal> globalVal;
    Program();
    void addFunction(const Function& proc);
    std::string draw();
};
```

除了函数体外，还需存放源程序中全局变量（这不仅仅是用于测评的需要，在后端生成汇编过程中也需对全局变量进行单独处理！）`ir::Operand val` 是全局变量的类型和名字，如果全局变量是一个数组时，`int maxlen` 是应该为数组申请的长度，否则应该为 0

```
struct GlobalVal {
    ir::Operand val;
    int maxlen = 0;     //为数组长度设计
    GlobalVal(ir::Operand va);
    GlobalVal(ir::Operand va, int len);
};
```

`void addFunction(const Function& proc);` 用于程序体初始化后向其中条件函数体。

`std::string draw();` 程序体的输出方式定义。除调用各函数体的draw方法外，还定义了全局变量的输出方式。

IR 定义

出于实验测评需要，我们针对实验Sysy语言设计并提供统一的IR框架。为简化大家工作，我们只设计一层 IR，并给出了严格的 IR 语义。所有IR均采用四元式的形式，即（**opcode**，**des**，**operand1**，**operand2**）

我们把 IR 分为几种类型，分别进行说明：

- 变量定义IR
- 变量赋值IR
- 算术运算IR
- 逻辑运算IR
- 访存与指针运算IR
- 类型转化IR
- 调用返回IR
- 跳转IR

变量定义IR

def

用于定义整形变量，第一个操作数为立即数或变量，第二个操作数不使用，结果为被赋值变量。示例如下：

`int a = 8;` => `def a, 8`

fdef

用于定义浮点数变量，操作数与结果含义与 `def` 相同。示例如下：

`float a = 8.0;` => `fdef a, 8.0`

> 注：
>
> 此处的变量是指在程序中定义的变量，在后面指令描述中变量包括程序中定义的变量与为生成 IR 而产生的临时变量等。在 IR 测评机中，认为一个出现在 **des** 位置的且没有被分配空间的变量即为一个新的变量，会自动为其分配空间，所以在 IR 中其实 def/fdef 并不是定义一个新变量的唯一方法（当然你可以用自己喜欢的方法去实现）

变量赋值IR

mov

用于整型变量间赋值情况，如临时变量给程序变量赋值或程序变量给临时变量赋值。第一个操作数为赋值变量，第二个操作数不使用，结果为被赋值变量。示例如下：

`int a = in[2];`

将生成如下IR：

`load t1, in, 2`

`mov a, t1`

优化后也可以合并为一条: `load a, in, 2`

`load` IR 在下文说明。

fmov

用于浮点型变量间赋值情况，操作数与结果含义与 `mov` 相同。示例如下：

`float a = fl[2];`

将生成如下IR：

`load t1, fl, 2`

`fmov a, t1`

算术运算IR

add

整型变量加法指令，用于两操作数均为整型变量情况。示例如下：

`a = b + c;` => `add a, b, c`

addi

立即数加法指令，用于两操作数均为整型且第二个操作数为立即数情况。示例如下：

`a = b + 2;` => `addi, a, b, 2`

fadd

浮点型变量加法指令，用于两操作数均为浮点型变量情况。示例如下：

`float b, c;`

`float a = b + c;` => `fadd a, b, c`

sub

整型变量减法指令，用于两操作数均为整型变量情况。示例如下：

`a = b - c;` => `sub a, b, c`

subi

立即数减法指令，用于两操作数均为整型且第二个操作数为立即数情况。示例如下：

`a = b - 2;` => `subi, a, b, 2`

fsub

浮点型变量减法指令，用于两操作数均为浮点型变量情况。示例如下：

```
float b, c;
```

`float a = b - c;` => `fsub a, b, c`

mul

整型变量乘法指令，用于两操作数均为整型变量情况。示例如下：

`int a = b * c;` => `mul a, b, c`

> 注：当有任一操作数为立即数时，建议额外生成一条 IR 指令来产生一个临时变量作为源操作数，保证 mul 的两个源操作数都是变量。（RISCV指令集中乘法不支持其中某个操作数为立即数TT，如果选择在后端处理源操作数是立即数的情况，寄存器的分配可能会麻烦一点）

fmul

浮点型变量乘法指令，用于两操作数均为浮点型变量情况。示例如下：

```
float b, c;
```

`float a = b * c;` => `fmul a, b, c`

div

整型变量除法指令，用于两操作数均为整型变量情况。示例如下：

`int a = b / c;` => `div a, b, c`

> 注：与 `mul` 相同，当有任一操作数为立即数时，建议额外生成一条 IR 指令来产生一个临时变量作为源操作数，保证 div 的两个源操作数都是变量

fdiv

浮点型变量除法指令，用于两操作数均为浮点型变量情况。示例如下：

```
float b, c;
```

`float a = b / c;` => `fdiv a, b, c`

mod

整型变量取余指令，示例如下：

`int a = b % c;` => `mod a, b, c`

逻辑运算IR

逻辑运算 IR 的运算结果是 1/0，同时整形与浮点型的变量之间不能直接做逻辑运算

lss

整型变量 `<` 运算，逻辑运算结果用变量表示。示例如下：

`a < b` => `lss t1, a, b`

flss

浮点型变量 `<` 运算，逻辑运算结果用变量表示。示例如下：

```
float a, b;
```

`a < b` => `flss t1, a, b`

leq

整型变量 `<=` 运算，逻辑运算结果用变量表示。示例如下：

`a <= b` => `leq t1, a, b`

fleq

浮点型变量 `<=` 运算，逻辑运算结果用变量表示。示例如下：

```
float a, b;
```

`a <= b` => `fleq t1, a, b`

gtr

整型变量 `>` 运算，逻辑运算结果用变量表示。示例如下：

`a > b` => `gtr t1, a, b`

fgtr

浮点型变量 `>` 运算，逻辑运算结果用变量表示。示例如下：

```
float a, b;
```

`a > b` => `fgtr t1, a, b`

geq

整型变量 `>=` 运算，逻辑运算结果用变量表示。示例如下：

`a >= b` => `geq t1, a, b`

fgeq

浮点型变量 `>=` 运算，逻辑运算结果用变量表示。示例如下：

```
float a, b;
```

`a >= b` => `fgeq t1, a, b`

eq

整型变量 `==` 运算，逻辑运算结果用变量表示。示例如下：

`a == b` => `eq t1, a, b`

feq

浮点型变量 `==` 运算，逻辑运算结果用变量表示。示例如下：

```
float a, b;
```

`a == b` => `feq t1, a, b`

neq

整型变量 `!=` 运算，逻辑运算结果用变量表示。示例如下：

`a != b` => `neq t1, a, b`

fneq

浮点型变量 `!=` 运算，逻辑运算结果用变量表示。示例如下：

```
float a, b;
```

`a != b` => `fneq t1, a, b`

_not

变量取非运算 `!` ，第一个操作数为取非变量，第二个操作数不使用，结果为取非结果变量。示例如下：

`a = !b;` => `not a, b`

_and

变量与运算 `&&` ，示例如下：

`a = b && c;` => `and a, b, c`

_or

变量或运算 `||` ，示例如下：

`a = b || c;` => `or a, b, c`

访存与指针运算IR

alloc

内存分配指令，用于局部数组变量声明。第一个操作数为数组长度（非栈帧移动长度），第二个操作数不使用，结果为数组名，数组名被视为一个指针。示例如下：

`int a[2];` => `alloc a, 2`

load

取数指令，这里load指从数组中取数。第一个操作数为数组名，第二个操作数为要取数所在数组下标，目的操作数为取数存放变量。示例如下：

`a = arr[2];` => `load a, arr, 2`

store

存数指令，指向数组中存数。第一个操作数为数组名，第二个操作数为要存数所在数组下标，目的操作数为存入的数。示例如下：

`arr[2] = 3;` => `store 3, arr, 2`

> 注：数组初始化时每个初始化数组元素均应生成一条 `store` IR

getptr

获取指针指令，这实际上是一个指针运算指令，第一个操作数为数组名，第二个操作数为数组下标，运算结果仍为指针，其值是数组名(基址)+数组下标(偏移量)之后的地址，目的操作数为存入的指针操作数。主要用于数组传参的情况，示例如下：

```
void f(int arr[][3]);

...

int A[3][3];
f(A[1]);
```

在传参时，**A[1]** 实际上是作为一个指针传入了函数 f，为了对这种情况进行支持，我们设计了这样一个 new_ptr = 基址(ptr) + 偏移量(int) 的IR指令，传参过程为：

```
1: getptr t1, A, 3
```

```
2: call t2, f(t1)
```

类型转换IR

cvt_i2f

整型变量转为浮点型变量，第一个操作数为待转换变量，结果为类型转换后变量，第二个操作数不使用。示例如下：

```
int a = 2;
```

```
float b = a; => cvt_i2f b, a
```

cvt_f2i

浮点型变量转为整型变量，第一个操作数为待转换变量，结果为类型转换后变量，第二个操作数不使用。示例如下：

```
float a = 2;
```

```
int b = a; => cvt_f2i b, a
```

调用返回IR

return

返回指令，第一个操作数为返回值，第二个操作数与结果不使用。示例如下：

```
return a; => return a
```

call

函数调用指令，第一个操作数的name应为函数名，结果操作数为函数返回值，固定为一临时变量（对于无返回值函数，即使在IR中看起来像是返回了一个变量，但该临时变量后续不会被使用，不影响最终测评结果）。示例如下：

```
int test(int a, int b);

res = test(arg1, arg2);
```

将生成如下IR：

```
call t1, test(arg1, arg2)
```

`mov res, t1`

> 在后端实现 **call** 和 **return** IR 时，即实现函数调用与返回功能，应该严格遵守 risc-v 的二进制接口和函数调用约定，否则将无法正确的调用库函数

跳转IR

goto

跳转指令。每条IR生成都会对应一标签，`goto` IR 跳转到某个标签的 IR 处。第一个操作数为跳转条件，其为整形变量或 `type = Type::null` 的变量，当为整形变量时表示条件跳转（值不等于0发生跳转），否则为无条件跳转。第二个操作数不使用，目的操作数应为整形，其值为跳转相对目前pc的偏移量。示例如下：

```
if (a < b) {
    ...
}
a = 1;
```

将生成如下IR：

`1: lss t1, a, b`

`2: if t1 goto [pc, 2]`

`3: goto [pc, 7]`

`...`

`10: mov a, 1`

空 IR

unuse

生成一条带有标签但无实际含义的IR，第一个操作数、第二个操作数与结果均不使用。可用于避免某些分支跳转情况假出口跳转到未知标签。示例如下：

```
if (a < b) {
    return 0;
}
```

若生成如下 IR，可以保证第三条 goto IR 跳转目标一定存在：

`1: lss t1, a, b`

`2: if t1 goto [pc, 2]`

`3: goto [pc, 2]`

`4: return 0`

`5: __unuse__`

> 该 IR 的后端实现可以实现为 nop 指令或者直接忽略

IR 使用样例

本文档是语义分析中 IR 的一个使用样例，为了便于同学们理解 IR 程序及其执行，这里直接根据源程序手动构造了 IR 程序，省去了语义分析的过程。

对于如下源程序：

```
int a;
int arr[2] = { 2, 4};
int func(int p){
    p = p - 1;
    return p;
}
int main(){
    int b;
    a = arr[1];
    b = func(a);
    if (b < a) b = b * 2;
    return b;
}
```

我们手动构造了 IR 示例程序如下：

```
//IR测试样例
#include <iostream>
#include "ir/ir.h"
#include "tools/ir_executor.h"
int main() {
    ir::Program program;
    ir::Function globalFunc("global", ir::Type::null);
    ir::Instruction assignInst(ir::Operand("0",ir::Type::IntLiteral),
                                    ir::Operand(),
                                    ir::Operand("a",ir::Type::Int),ir::Operator::d
ef);
    globalFunc.addInst(&assignInst);
    ir::Instruction allocInst(ir::Operand("2",ir::Type::IntLiteral),
                              ir::Operand(),
                              ir::Operand("arr",ir::Type::IntPtr),ir::Operator::all
oc);
    globalFunc.addInst(&allocInst);
    ir::Instruction storeInst(ir::Operand("arr",ir::Type::IntPtr),
                              ir::Operand("0",ir::Type::IntLiteral),
                              ir::Operand("2",ir::Type::IntLiteral),ir::Operator::s
tore);
    ir::Instruction storeInst1(ir::Operand("arr",ir::Type::IntPtr),
                              ir::Operand("1",ir::Type::IntLiteral),
                              ir::Operand("4",ir::Type::IntLiteral),ir::Operator::s
tore);
```

```
    ir::Instruction globalreturn(ir::Operand(),
                                 ir::Operand(),
                                 ir::Operand(), ir::Operator::_return);
    globalFunc.addInst(&storeInst);
    globalFunc.addInst(&storeInst1);
    globalFunc.addInst(&globalreturn);
    program.globalVal.emplace_back(ir::Operand("a",ir::Type::Int));
    program.globalVal.emplace_back(ir::Operand("arr",ir::Type::IntPtr),2);
    program.addFunction(globalFunc);
    std::vector<ir::Operand> paraVec = {ir::Operand("p",ir::Type::Int)};
    ir::Function funcFunction("func",paraVec,ir::Type::Int);
    ir::Instruction subInst(ir::Operand("p",ir::Type::Int),
                                    ir::Operand("1",ir::Type::IntLiteral),
                                    ir::Operand("t1",ir::Type::Int),ir::Operator::
subi);
    ir::Instruction movInst(ir::Operand("t1",ir::Type::Int),
                                    ir::Operand(),
                                    ir::Operand("p",ir::Type::Int),ir::Operator::m
ov);
    ir::Instruction returnInst(ir::Operand("p",ir::Type::Int),
                                    ir::Operand(),
                                    ir::Operand(),ir::Operator::_return);
    funcFunction.addInst(&subInst);
    funcFunction.addInst(&movInst);
    funcFunction.addInst(&returnInst);
    program.addFunction(funcFunction);
    ir::Function mainFunction("main",ir::Type::Int);
    ir::CallInst callGlobal(ir::Operand("global",ir::Type::null),
                                ir::Operand("t0",ir::Type::null));
    ir::Instruction defInst(ir::Operand("0",ir::Type::IntLiteral),
                                    ir::Operand(),
                                    ir::Operand("b",ir::Type::Int),ir::Operator::d
ef);
    ir::Instruction loadInst(ir::Operand("arr",ir::Type::IntPtr),
                                   ir::Operand("1",ir::Type::IntLiteral),
                                    ir::Operand("t2",ir::Type::Int),ir::Operator::
load);
    ir::Instruction movInst1(ir::Operand("t2",ir::Type::Int),
                             ir::Operand(),
                             ir::Operand("a",ir::Type::Int),ir::Operator::mov);
    std::vector<ir::Operand> paraVec1 = {ir::Operand("a",ir::Type::Int)};
    ir::CallInst callInst(ir::Operand("func",ir::Type::Int),
                                    paraVec1,
                                    ir::Operand("t2",ir::Type::Int));
    ir::Instruction movInst2(ir::Operand("t2",ir::Type::Int),
                                    ir::Operand(),
                                    ir::Operand("b",ir::Type::Int),ir::Operator::m
ov);
    ir::Instruction lssInst(ir::Operand("b",ir::Type::Int),
                                    ir::Operand("a",ir::Type::Int),
                                    ir::Operand("t3",ir::Type::Int),ir::Operator::
```

```
lss);
    ir::Instruction gotoInst(ir::Operand("t3",ir::Type::Int),
                                     ir::Operand(),
                                     ir::Operand("2",ir::Type::IntLiteral),ir::Oper
ator::_goto);
    ir::Instruction gotoInst1(ir::Operand(),
                                     ir::Operand(),
                                     ir::Operand("4",ir::Type::IntLiteral),ir::Oper
ator::_goto);
    ir::Instruction defInst2(ir::Operand("2",ir::Type::IntLiteral),
                                     ir::Operand(),
                                     ir::Operand("t4",ir::Type::Int),ir::Operator::
def);
    ir::Instruction mulInst(ir::Operand("b",ir::Type::Int),
                                     ir::Operand("t4",ir::Type::Int),
                                     ir::Operand("t5",ir::Type::Int),ir::Operator::
mul);
    ir::Instruction movInst3(ir::Operand("t5",ir::Type::Int),
                                    ir::Operand(),
                                     ir::Operand("b",ir::Type::Int),ir::Operator::m
ov);
    ir::Instruction returnInst1(ir::Operand("b",ir::Type::Int),
                                     ir::Operand(),
                                     ir::Operand(),ir::Operator::_return);
    mainFunction.addInst(&callGlobal);
    mainFunction.addInst(&defInst);
    mainFunction.addInst(&loadInst);
    mainFunction.addInst(&movInst1);
    mainFunction.addInst(&callInst);
    mainFunction.addInst(&movInst2);
    mainFunction.addInst(&lssInst);
    mainFunction.addInst(&gotoInst);
    mainFunction.addInst(&gotoInst1);
    mainFunction.addInst(&defInst2);
    mainFunction.addInst(&mulInst);
    mainFunction.addInst(&movInst3);
    mainFunction.addInst(&returnInst1);
    program.addFunction(mainFunction);
    std::cout << program.draw();
    // 进行验证
    ir::Executor executor(&program);
    std::cout << executor.run();
}
```

打印 IR 程序

program.draw() 的结果如下：

```
void global()
```

```
    0: def a, 0
    1: alloc arr, 2
    2: store 2, arr, 0
    3: store 4, arr, 1
    4: return null
end

int func(int p)
    0: subi t1, p, 1
    1: mov p, t1
    2: return p
end

int main()
    0: call t0, global()
    1: def b, 0
    2: load t2, arr, 1
    3: mov a, t2
    4: call t2, func(a)
    5: mov b, t2
    6: lss t3, b, a
    7: if t3 goto [pc, 2]
    8: goto [pc, 4]
    9: def t4, 2
    10: mul t5, b, t4
    11: mov b, t5
    12: return b
end

GVT:
    a int 0
    arr int* 2
```

执行 IR 程序

示例代码的最后两行是调用执行器运行生成的ir。执行过程将会跟踪每一步 value 的变化并输出最后 main 函数返回结果。

如果将宏 `DEBUG_EXEC_DETAIL` `DEBUG_EXEC_BRIEF` 打开，上述示例调用执行器将会有如下输出，打印出每一条 IR 指令的执行过程和最终返回结果 `6` 。

```
0: call t0, global()
0: def a, 0
    in get_des_operand(int a), value = 0
    in find_src_operand(intLiteral 0)    eval_int: 0
    des operand(int a), value = 0
1: alloc arr, 2
    in find_src_operand(intLiteral 2)    eval_int: 2
    in get_des_operand(int* arr), value = 0x9a2760
```

```
2: store 2, arr, 0
    in find_src_operand(intLiteral 0)    eval_int: 0
    in find_src_operand(intLiteral 2)    eval_int: 2
    in find_src_operand(int* arr), value = 0xa97380
3: store 4, arr, 1
    in find_src_operand(intLiteral 1)    eval_int: 1
    in find_src_operand(intLiteral 4)    eval_int: 4
    in find_src_operand(int* arr), value = 0xa97380
4: return null
1: def b, 0
    in get_des_operand(int b), new des (int b), value = 0
    in find_src_operand(intLiteral 0)    eval_int: 0
    des operand(int b), value = 0
2: load t2, arr, 1
    in find_src_operand(intLiteral 1)    eval_int: 1
    in find_src_operand(int* arr), value = 0xa97380
    in get_des_operand(int t2), new des (int t2), value = 0
3: mov a, t2
    in get_des_operand(int a), value = 0
    in find_src_operand(int t2), value = 4
    des operand(int a), value = 4
4: call t2, func(a)
    in get_des_operand(int t2), value = 4
    in find_src_operand(int a), value = 4
0: subi t1, p, 1
    in find_src_operand(int p), value = 4
    eval_int: 1
    in get_des_operand(int t1), new des (int t1), value = 0
1: mov p, t1
    in get_des_operand(int p), value = 4
    in find_src_operand(int t1), value = 3
    des operand(int p), value = 3
2: return p
    in find_src_operand(int p), value = 3
5: mov b, t2
    in get_des_operand(int b), value = 0
    in find_src_operand(int t2), value = 3
    des operand(int b), value = 3
6: lss t3, b, a
    in find_src_operand(int b), value = 3
    in find_src_operand(int a), value = 4
    in get_des_operand(int t3), new des (int t3), value = 0
    des operand(int t3), value = 1
7: if t3 goto [pc, 2]
    in find_src_operand(intLiteral 2)    eval_int: 2
    in find_src_operand(int t3), value = 1
    in goto: pc = 9
9: def t4, 2
    in get_des_operand(int t4), new des (int t4), value = 0
    in find_src_operand(intLiteral 2)    eval_int: 2
    des operand(int t4), value = 2
```

```
10: mul t5, b, t4
    in find_src_operand(int b), value = 3
    in find_src_operand(int t4), value = 2
    in get_des_operand(int t5), new des (int t5), value = 0
    des operand(int t5), value = 6
11: mov b, t5
    in get_des_operand(int b), value = 3
    in find_src_operand(int t5), value = 6
    des operand(int b), value = 6
12: return b
    in find_src_operand(int b), value = 6
6
```

Global的处理

请注意源程序中并没有一个叫做 global 的函数，是因为需要对全局变量进行初始化，所以采用了这样一个特殊的做法

调用处理

IR生成需涉及对全局变量、全局常量的处理。一种可行的方法是将global作为一个 function 进行处理，除去其中变量、常量定义声明的 IR 外，仍需生成一条 return null 的 IR。并在 main 函数中首先生成对global的调用IR。上面的示例就是采用这种方式。

*注：IR的结果仅供参考并不唯一，比如上面func中第0条和第1条就可用一条 `subi p, p, 1` 表示。语义分析处理方式不同，生成IR就可能不同。

语义分析

1. 语义分析基本思路

我们经过词法和语法分析之后已经得到了 AST，这其实是 SysY 程序的另一种形式的表示。源程序代码中的各种顺序、结构信息都存储在树中，我们可以通过深度遍历语法树按源程序的顺序来分析源程序。

以下对语义分析中的一些重点问题做一些讨论，你们也可以修改下面的设计以符合自己的思路

2. 作用域与符号表

在 Sysy 中, 作用域是由 Block 决定的, 允许嵌套且不同作用域中可以定义同名变量。在翻译成 IR 的过程中我们需要解决不同作用域中同名变量的问题, 我们的解决方案是重命名, 为变量名加上与作用域相关的后缀使得重命名之后的变量名字在一个 IR Function 中是独一无二的

符号

```
struct STE {
    ir::Operand operand;
    vector<int> dimension;
};
```

Symbol Table Entry(STE) 是符号表中的一条记录， `ir::Operand operand` 记录了符号的名字和类型，但是对于数组来说，我们不止需要知道名字和类型，在语义分析的过程中还需要的 `vector<int> dimension`

作用域

为支持重命名, 我们需要一个数据结构来存储作用域相关的信息, 每个作用域有自己的属性来唯一标识自己, 还有一张表来存储这个作用域里所有变量的名称和类型

```
using map_str_ste = map<string, STE>;
// definition of scope infomation
struct ScopeInfo {
    int cnt;
    string name;
    map_str_ste table;
};
```

cnt 是作用域在函数中的唯一编号, 代表是函数中出现的第几个作用域

name 可以用来分辨作用域的类别, 'b' 代表是一个单独嵌套的作用域, 'i' 'e' 'w' 分别代表由 if else while 产生的新作用域（你也可以取你喜欢的名字，只是这样会表意比较清晰）

table 是一张存放符号的表, {string: STE}, string 是操作数的原始名称, 表项 STE 实际上就是一个 IR 的操作数，即 STE -> Operand, 在 STE 中存放的应该是变量重命名后的名称

符号表

作用域是支持嵌套, 我们决定使用栈式结构来存储

```
struct SymbolTable {
    vector<ScopeInfo> scope_stack;
    map<std::string,ir::Function*> functions;

    ...
}
```

并为符号表提供了一些成员函数, 用于查询变量, 并提供 ScopeInfo 相关的操作

```
void add_scope(Block*);                                    进入新作用域时，向符号表中添加 S
copeInfo，相当于压栈

void exit_scope();                                         退出时弹栈

string get_scoped_name(string id) const;                   输入一个变量名，返回其在当前作用
域下重命名后的名字（相当于加后缀）

Operand get_operand(string id) const;                      输入一个变量名，在符号表中寻找最
近的同名变量，返回对应的 Operand(注意，此 Operand 的 name 是重命名后的)

STE get_ste(string id) const;                              输入一个变量名，在符号表中寻找最
近的同名变量，返回 STE
```

3. 表达式

观察文法可以发现, 表达式的文法展开其实和计算顺序是一致的, 只需要自底向上翻译, 即可得到正确的 IR 序列

在表达式计算还需要考虑几个其他的问题: 类型转换, 临时变量的产生与管理, 以及可以在分析树中做的常数合并优化

对于表达式相关的 AstNode 我们需要增加几个属性来记录一些信息以完成以上需求

```
Exp.is_computable                                          节点以下子树是否可以化简为常数，
通过该变量，大部分常数合并可以直接在语法树中自底向上进行传递

Exp.v                                                      一个字符串，或者是重命名后的变量
名，或者是临时变量名称，也可以是常数字符串
```

```
Exp.t                                                    Type, 表示该表达式计算得到的类型
```

对于一个表达式相关的节点, 其值可以由 Operand(v, t) 来表示, 在自底向上的分析过程中认为子树的值将保存在 Operand(v, t) 所代表的操作数内, 这样的规定意味着：

- 在完成该子树的分析后，应生成一条以 Operand(v, t) 为目标操作数的 IR
- 在自底向上的过程中，可以认为该子树代表的值就存放在 Operand(v, t) 在之后的分析过程中可以使用

以表达式

```
a + b / 10;
```

为例，其应该生成的语法树为

```
{
    "name" : "Exp",
    "subtree" : [
    {
        "name" : "AddExp1",
        "subtree" : [
                ...
                {
                    "name" : "Terminal",
                    "type" : "IDENFR",
                    "value" : "a"
                }
                ...
            ]
            },
            {
                "name" : "Terminal",
                "type" : "PLUS",
                "value" : "+"
            },
            {
                "name" : "AddExp2",
                "subtree" : [
                    {
                    "name" : "MulExp",
                    "subtree" : [
                        {
                        ...
                        {
                            "name" : "Terminal",
                            "type" : "IDENFR",
                            "value" : "b"
```

```
                        }
                        ...
                        },
                        {
                            "name" : "Terminal",
                            "type" : "DIV",
                            "value" : "/"
                        },
                        ...
                        {
                            "name" : "Terminal",
                            "type" : "INTLTR",
                            "value" : "10"
                        }
                    ]
                    }
                ]
            }
        ]
    }
    ]
}
```

以 Exp 为根节点，在分析这颗语法树时，我们采用深度遍历自底向上的方式，首先遇到的是第一个左子树 AddExp

```
    "name" : "AddExp1",
    "subtree" : [
        ...
        {
            "name" : "Terminal",
            "type" : "IDENFR",
            "value" : "a"
        }
        ...
    ]
```

从 AddExp 开始向下，每一层都只有一个节点，我们之前提到一个表达式相关节点可以由 Operand(v, t) 来表示他的值，所以问题的关键是如何生成这个值对应的 IR，以下提供一种 原来的语法树结构大概如下所示

```
AddExp -> MulExp -> UnaryExp -> PrimaryExp -> LVal -> Terminal(IDENFR, a)
```

在加上节点自己的属性后应该为这样：

```
AddExp(v_add1, t) -> MulExp(v_mul, Int) -> UnaryExp(v_uan, Int) -> PrimaryExp(v_pri
, Int) -> LVal(v_lva, Int) -> Terminal(IDENFR, a)
```

在分析的过程中，最底层的 `LVal(v_lva, Int) -> Terminal(IDENFR, a)` 中变量 a 应是一个已定义的变量，可以使用 **SymbolTable::get_operand(string id);** 找到对应的 Operand，然后产生由 `(v_lva, Int) -> (v_pri, Int) -> (v_uan, Int) -> (v_mul, Int) -> (v_add1, t)` 赋值的 IR。另一种方法是，为避免产生这么多无用的赋值，值进行语法树属性的传递，即通过语法树属性 (v, t) 的赋值，使 AddExp(v_add1, t) 与 Terminal(IDENFR, a) 对应的 Operand 相同

分析完 Exp 的第一个子节点 AddExp1 后接下来第二个节点是 `Terminal(PLUS, +)`，待对第三个节点 AddExp2 分析完成后，得到代表第三个节点值的 Operand(v_add2, t)，即可生成一条加法 IR

```
add     v_exp, v_add1, v_add2
```

这里的 (v_exp, t) 是代表 Exp 节点的值的 Operand，符合 子树的值将保存在 **Operand(v, t)** 所代表的操作数中 这个假设，由此可以看到，在这样的假设下生成的 IR 可以在遍历语法树的过程中传递起来并保证顺序的正确

4. Stmt: 语句

TO BE CONTINUE

5. 程序接口

在本次实验中，你们需要实现 Analyzer 类，完成 `ir::Program get_ir_program(CompUnit*);` 接口，该接口接受一个源程序语法树的根节点 **Comp***，对其进行分析，返回分析结果 **ir::Program**

```
struct Analyzer {
    int tmp_cnt;
    vector<ir::Instruction*> g_init_inst;
    SymbolTable symbol_table;

    /**
     * @brief constructor
     */
    Analyzer();

    // analysis functions
    ir::Program get_ir_program(CompUnit*);

    // reject copy & assignment
    Analyzer(const Analyzer&) = delete;
    Analyzer& operator=(const Analyzer&) = delete;

    // analysis functions
    void analysisCompUnit(CompUnit*, ir::Program&);
    void analysisFuncDef(FuncDef*, ir::Function&);
    void analysisDecl(Decl*, vector<ir::Instruction*>&);
    void analysisConstDecl(ConstDecl*, vector<ir::Instruction*>&);
```

```
    void analysisBType(BType*, vector<ir::Instruction*>&);
    void analysisConstDef(ConstDef*, vector<ir::Instruction*>&);
    void analysisConstInitVal(ConstInitVal*, vector<ir::Instruction*>&);
    ...
};
```

运行时库

运行时库提供一系列I/O函数用于在程序中表达输入输出功能，这些库函数不在输入源程序中声明即可在源程序的函数中使用。参考实现：sylib.c sylib.h

相关文件

(你们在实验三前不需要在意提供文件的部分，因为实验二的库函数是写在测评机里的) 提供如下运行时库文件：

- **libsysy.a**运行时库的静态库（面向不同目标平台）。我们实验评测均采用静态库链接进行评测。
- **sylib.h**包含运行时库涉及的函数等的声明。

需要注意，源程序中不出现对**sylib.h**的文件包含，而是由同学们写的编译器分析处理源程序中对这些函数的调用。

I/O函数

int getint()

输入一个整数，返回对应的整数值。

示例如下：

```
int n; n = getint();
```

int getch()

输入一个字符，返回字符对应的ASCII码值。

示例如下：

```
int n; n = getch();
```

float getfloat()

输入一个浮点数，返回对应的浮点数值。

示例如下：

```
float n; n = getfloat();
```

int getarray(int[])

输入一串整数，第一个整数代表后续要输入的整数个数，该个数通过返回值返回；后续的整数通过传入的数组参数返回。

注：getarray函数获取传入的数组的起始地址，不检查调用者提供的数组是否有足够的空间容纳输入的一串整数。

示例如下：

```
int a[10][10]; int n = getarray(a[0]);
```

int getfarray(float [])

输入一个整数后跟若干浮点数，第一个整数代表后续要输入的浮点数个数，该个数通过返回值返回；后续的浮点数通过传入的数组参数返回。

注：getfarray函数获取传入的数组的起始地址，不检查调用者提供的数组是否有足够的空间容纳输入的一串浮点数。

示例如下：

```
float a[10][10]; int n = getfarray(a[0]);
```

void putint(int)

输出一个整数的值。

示例如下：

```
int n = 10; putint(n); putint(10); putint(n);
// 将输出:10 10 10
```

void putch(int)

将整数参数的值作为ASCII码，输出该ASCII码对应的字符。

注：传入的整数参数取值范围为0~255，putch()不检查参数的合法性。

示例如下：

```
int n = 10; putch(n);
// 将输出换行符
```

void putfloat(float)

输出一个浮点数的值 ()

示例如下：

```
float n = 10.0; putfloat(n);
```

```
// 将输出:0x1.400000p+3
```

void putarray(int,int[])

第一个参数表示要输出的整数个数（假设为N），后面跟上要输出的N个整数的数组。putarray在输出时会在整数之间安插空格。

注：putarray函数不检查参数的合法性。

示例如下：

```
int n = 2; int a[2]={2,3}; putarray(n,a);
// 输出:
// 2: 2 3
```

void putfarray(int,float[])

第一个参数表示要输出的浮点数个数（假设为N），后面跟上要输出的N个浮点数的数组。putfarray在输出时会在浮点数之间安插空格。

注：putfarray函数不检查参数的合法性。

示例如下：

```
int n = 2; float a[2] = {2.0, 3.0};
putfarray(n,a);
// 输出:
// 2: 0x1.000000p+1 0x1.800000p+1
```

计时函数

FIXME in lab4

实验三

实验目标

由实验二的 中间表示 **IR** 经过目标代码生成，生成可以与 `sylib.a` 链接的 **risc-v** 汇编

实验步骤

从希冀上下载实验框架

compiler [src_filename] -S -o [output_filename] 将输出你的汇编结果至 [output_filename]

测评方法：

使用 `riscv32-unknown-linux-gnu-gcc ur_assembly sylib-riscv-linux.a` 来编译你生成的 risc-v 汇编，生成 risc-v 的可执行文件

使用 qemu-riscv32 来模拟 risc-v 机器执行你的可执行文件

> 我们提供了脚本 **qemu-riscv32.sh** 来简化使用 qemu，你可以使用命令： `qemu-riscv32.sh ur_rv_executable` 来执行 risc-v 的可执行文件

实验三标准输出

这是一段简单的 SysY 程序

```
int main() {
    return 3;
}
```

本实验没有标准答案，只要汇编可以被正确编译执行即可，gcc生成的汇编可以供你参考：

```
.file    "00_main.c"
    .option nopic
    .text
    .align    1
    .globl    main
    .type    main, @function
main:
    addi    sp,sp,-16
    sw    s0,12(sp)
    addi    s0,sp,16
    li    a5,3
    mv    a0,a5
    lw    s0,12(sp)
```

```
    addi    sp,sp,16
    jr    ra
    .size    main, .-main
    .ident    "GCC: (GNU) 9.2.0"
    .section    .note.GNU-stack,"",@progbits
```

目标代码生成

目标代码生成的结果是汇编，汇编经过 gcc 的汇编器处理之后，变为可执行文件。要完成这一部分实验，需要对汇编和可执行程序有深刻的理解。这篇文档主要是在告诉你们需要生成什么样的汇编代码，你们应该解决的问题是如何完成一个生成这样的汇编代码的程序，当然文档里也有适当的提示

0. 前置知识

虽然你们可能没有专门学过汇编，但是组成原理课和硬件综合设计中你们已经和它打过交道了。一切的出发点，来自于组成原理，在这门课中你已经了解了指令如何在硬件上运行，现在你应该学习并实现如何用一条条指令来实现高级语言程序的功能了，汇编只是指令的助记符而已。以下重新介绍一下组成原理相关知识，以免你们已经忘记它了。

冯诺依曼结构

诺依曼结构（Von Neumann architecture）是一种计算机体系结构，以数学家冯·诺依曼（John von Neumann）的名字命名。它是一种将程序指令和数据存储在同一存储器中的计算机设计原则，也被称为存储程序计算机（stored-program computer）。

在冯诺依曼结构中，计算机系统包括以下主要组件：

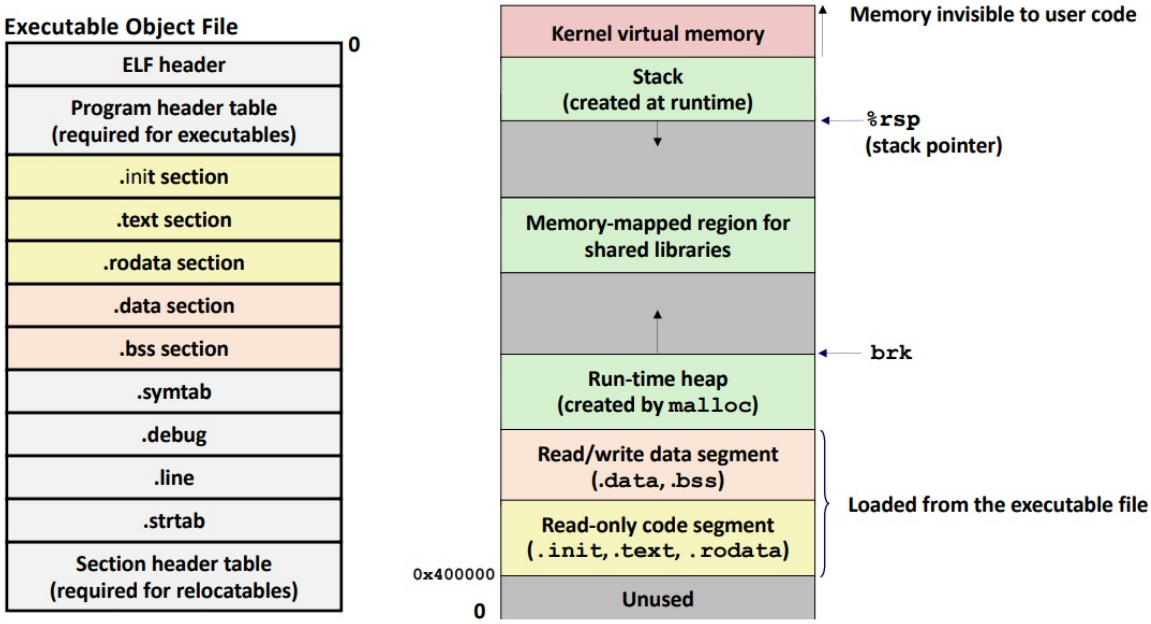
1. 中央处理器（Central Processing Unit，CPU）：负责执行指令和处理数据的核心部分，包括算术逻辑单元（Arithmetic Logic Unit，ALU）和控制单元（Control Unit）。
1. 存储器（Memory）：用于存储指令和数据的设备，包括主存储器（Main Memory）和辅助存储器（Auxiliary Memory）。
1. 输入/输出设备（Input/Output Devices）：用于与外部设备进行交互，如键盘、鼠标、显示器、硬盘等。

存储程序概念是冯诺依曼结构的关键特征之一。根据这一概念，计算机可以将指令和数据存储在同一存储器中，并按照地址访问。它将程序指令看作数据的一种形式，计算机可以像操作数据一样操作指令，使得程序可以被存储、加载和执行。

可执行文件与其内存映像

可执行文件是一种用于在计算机系统上执行的二进制文件格式。它包含了计算机程序的机器代码、数据、符号表和其他必要的信息，包括头部、代码段、数据段、符号表和重定位表等部分。

程序执行的内存映像是可执行文件在计算机被执行过程中的内存布局和状态。它包括代码段、数据段、堆栈和其他可执行文件所需的资源。代码段存储程序的指令，数据段存储静态数据和全局变量，堆用于动态分配内存，栈用于函数调用和局部变量的存储。内存映像还包括处理器寄存器的状态和其他与程序执行相关的信息。



注：上图中 <%esp> 是 x86 中的栈指针，riscv 中应为 sp

用汇编来描述可执行文件！

汇编文件可以描述不同的节及其数据（根据存储程序概念，这里的数据有一部分就是指令），在汇编中可以通过特定的语法来描述节的开始，在节与节之间的填写指令和伪指令，由汇编器将其转换为二进制的数据，即可执行文件

你生成的汇编程序大概包括以下部分:

```
    .data
     ...

    .bss
     ...

    .text

# f 是源程序中的一个函数，通过 jr  ra 返回
f:
    ...
    jr  ra

# main 是源程序中的一个函数，执行过程中通过 jal f 调用函数 f，通过 jr  ra 返回
main:
    ...
    jal f
    ...
    jr  ra
```

.data 表示数据段开始，接下来可以使用 `.word` `.byte` `.commn` 数据相关的伪指令来记录数据，通常是指用来存放程序中已初始化的全局变量的一块内存区域，数据段属于静态内存分配

.bss 表示 bss(Block Started by Symbol) 段开始，可以使用 `.space` 等指令分配初始化为 0 的一块区域，属于静态内存分配

.text 表示代码段开始，通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读, 某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等

`f` 和 `main` 在这里被叫做符号或标签，他们本身不会被真正存放在代码段中，他们代表的是下一条指令的地址，如果在指令中使用了该符号，汇编器在汇编时会填入该地址

`jal f` 的反汇编结果如下， `10350` 是这一条指令的地址， `3fe1` 是这条 jal 指令对应的二进码， `jal 10328 <f>` 中 `10328` 是符号 `f` 所代表的地址，也就是 函数**f** 的第一条汇编的地址

```
    10350:    3fe1                    jal    10328 <f>
```

汇编中的指令和伪指令只是助记符，每一条其实就代表了一段二进制码，不同的节中的指令由汇编器放到不同的地方，所以用汇编就可以描述可执行文件！

从哪里获得示例和参考？

为了清晰的看到汇编是如何转化为可执行文件，对于一个 SysY 源程序，你可以把他转化为 C 文件，通过 gcc 编译成汇编和可执行文件，可执行文件可以通过 objdump 进行反汇编，观察汇编与 dump 结果。同时 gcc 的汇编结果可以作为你生成汇编程序的参考

> 不知道工具如何使用？ 文档不会告诉你的，STFW！

程序二进制接口

程序二进制接口 **ABI（application binary interface）** 定义了机器代码（即汇编）如何访问数据结构与运算程序，它包含了

数据类型的大小、布局和对齐

函数调用约定

系统调用约定

二进制目标文件格式

等内容。决定要不要采取既定的ABI通常由编译器，操作系统或库的开发者来决定。但如果撰写一个混和多个编程语言的应用程序，就必须直接处理ABI，采用外部函数调用来达成此目的。

我们在开发编译器的过程中，生成的汇编文件应 遵循 **riscv ABI** 规范，这样我们编译器的生成的汇编才可以使用库函数，正确的被加载，并在执行后正确的返回。

> 如果不这么做就会出现各种奇奇怪怪的段错误！你可能不了解 riscv ABI，但是我想你应该知道怎么做

1. 内存管理

我们知道在 ir 中，每个函数在执行时都需要维护自己的上下文，包括局部变量、运行位置。在程序执行的过程中，我们同样需要在内存中的栈上保存他们。

变量寻址

在 IR 测评机中我们屏蔽掉了内存上的细节，IR 测评机保证你可以通过名字找到操作数，但是汇编中只能从内存中的某一个地址读取数据。所以我们需要对内存进行管理和分配，并记录内存地址与变量的对应关系。在变量分配的过程中，需要考虑 由 **ABI** 规定的数据类型的大小、布局和对齐 等问题

因为函数的局部变量保存在栈上，可以使用 栈指针 + 偏移量 的方法来确定变量在内存中的位置。在框架代码中为你们提供了这样一个数据结构，将变量与栈指针偏移量做映射，在内存中一一对应

```
struct stackVarMap {
    std::map<ir::Operand, int> _table;

    /**
     * @brief find the addr of a ir::Operand
     * @return the offset
    */
    int find_operand(ir::Operand);

    /**
     * @brief add a ir::Operand into current map, alloc space for this variable in
memory
     * @param[in] size: the space needed(in byte)
     * @return the offset
    */
    int add_operand(ir::Operand, uint32_t size = 4);
};
```

局部变量寻址由函数 `int find_operand(ir::Operand);` 返回的偏移量 + 栈指针来实现；向 table 中添加变量并维护变量与偏移量的映射由 `int add_operand(ir::Operand, uint32_t size = 4);` 来实现，你需要实现这两个函数

变量与偏移量的映射 只是一个逻辑映射，产生汇编代码时认为变量的值存放与此，你需要产生汇编代码来完成内存的分配、使用 load/store 操作相应地址来 读取/存放 变量的值

示例

以 函数**f** 为例子说明内存管理的内容，其定义如下：

```
void f() {
    int a,b;
    int A[5];
    ...
}
```

该函数使用了两个整形变量和一个数组，共 **2x4** + **5x4** = **28** bytes

其映射可以为：

```
{a, Int}        : sp+4
{b, Int}        : sp+8
{A, IntLiteral} : sp+12
```

那么在栈中操作变量值的汇编为

```
# a
lw  t0, 4(sp)
sw  t0, 4(sp)

# b
lw  t0, 8(sp)
sw  t0, 8(sp)

# A[1]
lw  t0, 16(sp)
sw  t0, 16(sp)
```

> A 数组占用 20 bytes，从 sp+12 到 sp+28，但是可以有两种存放顺序
>
> 1. A[0] 存放在 sp+12，其余依次递增
> 2. A[0] 存放在 sp+28，其余依次递减
>
> 此处 A[1] 的地址计算为 &A[0] + 4，只是举例说明，不一定正确，你需要根据 riscv 对数组顺序的规定来生成对应的代码

2. 函数调用

栈与栈指针

内存中的栈用于存储函数调用过程中的局部变量、函数参数、返回地址等信息，通过栈指针来管理。

栈指针的变化反映了栈的状态，包括栈的扩展和收缩。当函数被调用时，栈指针会向下移动，为局部变量和其他相关数据分配空间；当函数返回时，栈指针会向上移动，释放已分配的空间，恢复到调用该函数之前的状态。

函数调用约定

函数调用过程通常分为以下六步：

1. 调用者将参数存储到被调用的函数可以访问到的位置；
2. 跳转到被调用函数起始位置；

3. 被调用函数获取所需要的局部存储资源，按需保存寄存器**(callee saved registers)** ；
4. 执行函数中的指令；
5. 将返回值存储到调用者能够访问到的位置，恢复之前保存的寄存器**(callee saved registers)**，释放局部存储资源；
6. 返回调用函数的位置。

> 查阅相关资料阅读 **riscv calling convention** 相关内容，推荐 riscv 官方技术手册

callee saved registers 是那些 caller 不希望在函数调用中被改变的寄存器，如果 callee 用到了这些寄存器，应该把他们保存下来并在函数调用返回时恢复

函数调用约定规定了函数调用者（caller）和被调用者（callee）分别需要做的事情，以及如何去做，以下分别介绍如何使用汇编实现函数调用

caller 示例

caller 需要执行步骤 1 和 2 —— 将参数存储到被调用的函数可以访问到的位置，跳转到被调用函数起始位置

其汇编应实现为

```
caller:
    # 将参数存储到被调用的函数可以访问到的位置，查阅 riscv ABI 完成此项
    ...
    # 跳转到 callee 的第一条指令所在地址
    jal callee
    # 函数调用约定保证 callee 返回后执行 jal callee 的下一条指令
    ...
```

`jal label` 其实是 `jal x1, imm` 的伪指令，其功能为将 PC+4 存放到 x1 寄存器，并使 PC += imm；callee 只需要使用指令取出 x1 的值并跳转，即可回到 caller 原来执行的位置继续执行

> callee 知道 caller 会将 PC+4 存放在 x1 中，才可以在结束时跳转回这个地址，这便是函数调用约定的意义，你知道为什么是 x1 吗？

callee 示例

每一个函数都可能是被调用者，所以每一个函数的汇编实现都应该遵守被调用函数的约定

callee 需要在开始时先将 callee saved registers 保存到栈上，即将需要保存的寄存器 push 到内存的栈中；然后进行内存分配 —— 通过将 sp 向下移动来实现；此时才可以执行实现函数功能所需要的指令；执行完成后，需要向上移动 sp 来回收分配给该函数的栈空间，再 pop 出 callee saved registers 到原来的寄存器里，将返回值存储到调用者能够访问到的位置，最后跳转回原来的位置

其汇编应实现为

```
callee:
    # 将 callee saved registers 保存到栈上
    # 查阅 riscv ABI 以明白哪些是 callee saved registers，此处以保存 x1 为例
```

```
    addi    sp, sp, -4
    sw      x1, 4(s0)
    ...

    # 内存分配，分配的栈空间用于存放局部变量，按需分配
    addi    sp, sp, -xxx

    # 实现函数功能所需指令
    ...

    # 内存回收，与分配值相同
    addi    sp, sp, xxx

    # 将保存到栈上的 callee saved registers 重新 pop 到原寄存器
    ...
    lw      x1, 4(s0)
    addi    sp, sp, 4

    # 返回
    # 将返回值存储到调用者能够访问到的位置，查阅 riscv ABI 完成此项
    ...
    # 最后，根据函数调用约定，调用者的 PC+4 就存放在 x1 中，跳转回去即可
    jr      x1
```

3. 全局变量

全局变量存放在内存的数据段中，在汇编可以使用 `.space` `.word` 等伪指令声明，通过标签对其进行寻址

示例

定义一个全局变量 `int a = 7;` ，以下展示如何声明并对他进行寻址

```
.data
a:  .word   7
```

使用 `lw rd, symbol` 可以加载全局变量的值， `sw rd, symbol, rt` 可以保存全局变量的值

```
lw  t0, a       # 加载
sw  t0, a, t1   # 保存
```

4. 生成指令与函数

riscv 寄存器与指令数据结构定义

代码框架中定义了一些 riscv 相关的数据结构，rv_def.h 定义了 riscv 寄存器、指令的枚举类，此处并没有列举出所有的指令类型，你需要时可以自行添加

```
// rv interger registers
enum class rvREG {
    /* Xn       its ABI name*/
    X0,         // zero
    X1,         // ra
    X2,         // sp
    X3,         // gp
    X4,         // tp
    X5,         // t0
    ...
};
std::string toString(rvREG r);  // implement this in ur own way

enum class rvFREG {
    F0,
    F1,
    F2,
    F3,
    F4,
    F5,
    ...
};
std::string toString(rvFREG r);  // implement this in ur own way

// rv32i instructions
// add instruction u need here!
enum class rvOPCODE {
    // RV32I Base Integer Instructions
    ADD, SUB, XOR, OR, AND, SLL, SRL, SRA, SLT, SLTU,       // arithmetic & logic
    ADDI, XORI, ORI, ANDI, SLLI, SRLI, SRAI, SLTI, SLTIU,   // immediate
    LW, SW,                                                 // load & store
    BEQ, BNE, BLT, BGE, BLTU, BGEU,                         // conditional branch
    JAL, JALR,                                              // jump

    // RV32M Multiply Extension

    // RV32F / D Floating-Point Extensions

    // Pseudo Instructions
    LA, LI, MOV, J,                                         // ...
};
std::string toString(rvOPCODE r);  // implement this in ur own way
```

rv_inst_impl.h 定义了一个通用的 riscv 指令数据结构（当然你可以不使用它，选择你喜欢的实现方式

```
struct rv_inst {
```

```
    rvREG rd, rs1, rs2;         // operands of rv inst
    rvOPCODE op;                // opcode of rv inst

    uint32_t imm;               // optional, in immediate inst
    std::string label;          // optional, in beq/jarl inst

    std::string draw() const;
};
```

通过 `std::string draw() const` 对不同 rvOPCODE 进行不同的输出，可以使该数据结构支持普通指令 (opcode, rd, rs, rt) ，也可以支持立即数指令(opcode, rd, rs, imm)，还可以支持在该指令前打上标签 (label: opcode, rd, rs, rt) 或是跳转指令 (jump, rd, label)

生成指令

接下来我们考虑如何将 **IR** 翻译为指令，我们的 IR 其实与指令十分相似，只是 IR 中没有寄存器的概念，变量值直接从内存中读取，现在我们需要使用 load & store 指令从内存中获取变量，加载到某个寄存器，在此过程中需要处理寄存器分配，考虑 **ABI** 对特定寄存器的功能的规定

一种简单的处理的处理方式是不进行寄存器的分配，将每条 IR 的 op1 & op2 对应地址的值 load 到临时寄存器 t0 & t1，使用相应的 riscv 指令进行运算，指令的 rd 选择固定的 t2，然后将其 store 回 des 所对应的地址

根据 **1.** 内存管理 的介绍，你已经知道怎么处理局部变量的地址了，记得处理全局变量这个例外情况。假设一条 IR 为 `add sum, a, b` ，他们都是局部变量，那么其汇编应实现为：

```
lw  t0, xx(sp)  # load Operand(a) 到 t0
lw  t1, xx(sp)  # load Operand(b) 到 t1
add t2, t0, t1
sw  t2, xx(sp)  # store t2 到 Operand(sum) 对应地址
```

> 并不是每一种 IR Operator 都有对应的指令，但是你可以通过指令的组合实现，具体请查阅 riscv 手册

代码框架种定义了接口

```
void gen_instr(const ir::Instruction&);
```

你可以通过实现该函数来完成 IR 到指令的生成过程，其伪代码大概如下，记得考虑全局变量的情况：

```
void backend::Generator::gen_instr(const ir::Instruction& inst) {
    // if local    : get_ld_inst() -> {op = lw, rd = t0, rs = sp, imm = map.find_op
erand(inst.op1)};
    // if global   : get_ld_inst() -> {op = lw, rd = t0, label = inst.op1.name};
    rv::rv_inst ld_op1 = get_ld_inst(inst.op1);
    rv::rv_inst ld_op2 = get_ld_inst(inst.op2);
```

```
    // translate IR into instruction
    rv::rv_inst ir_inst;
    switch (inst.op) {
    case ir::Operator::add:
        ir_inst = {op = add, rd = t2, rs = t0, rt = t1};
        break;
    ...
    default:
        assert(0 && "illegal inst.op");
        break;
    }

    rv::rv_inst st_des = get_st_inst(inst.des);

    output(ld_op1, ld_op2, ir_inst, st_des);
}
```

寄存器分配

在以上的介绍中，我们使用了固定的寄存器作为源操作数和目的操作数，但是这样程序就需要经常进行访存，所以需要进行寄存器分配，为程序处理的值找到存储位置的问题。这些变量可以存放到寄存器，也可以存放在内存中。寄存器更快，但数量有限。内存很多，但访问速度慢。好的寄存器分配算法尽量将使用更频繁的变量保存的寄存器中。

代码框架中提供了一套 api，你可以在根据自己的需求修改并实现他们以进行寄存器的分配

> 即使你不想做寄存器的分配，你也应该在 gen_instr 使用这一套 api 而不是写死 t0 t1 t2，固定分配只需要将 api 实现为返回固定寄存器即可，例如 **getRd** 实现为 **return rv::rvReg::X7;**。当你在后续的实验中做寄存器分配时就只需要修改 api 的内部实现，这样做程序才拥有可扩展性

```
// reg allocate api
rv::rvREG getRd(ir::Operand);
rv::rvFREG fgetRd(ir::Operand);
rv::rvREG getRs1(ir::Operand);
rv::rvREG getRs2(ir::Operand);
rv::rvFREG fgetRs1(ir::Operand);
rv::rvFREG fgetRs2(ir::Operand);
```

生成函数

在 **3.** 函数调用 的 **callee** 示例 种详细介绍了一个函数的汇编实现，代码框架种定义了接口

```
void gen_func(const ir::Function&);
```

你可以通过实现该函数来完成 IR Function 到函数汇编指令的生成过程，其伪代码大概如下：

```
void gen_func(const ir::Function& func) {
```

```
    // do sth to deal with callee saved register & subtract stack pointer
    ...
    output(...);

    // translate all IRs in InstVec into assembly
    for (auto inst: func.InstVec) {
        gen_instr(*inst);
    }

    // do sth to pop callee saved register & recovery stack pointer
    ...
    output(...);

    // gen instructio to jump back
    rv::rv_inst jump_back = {op = jr, rd = x1};
    output(jump_back);
}
```

5. 程序接口

在本次实验中，你们需要实现 Generator 类，main 函数保证调用 `void gen();` ，你需要在该函数中实现 ir::Program 到 riscv 汇编程序的转化，并将生成的汇编程序写入 `fout` 中

```
struct Generator {
    const ir::Program& program;         // the program to gen
    std::ofstream& fout;                 // output file

    Generator(ir::Program&, std::ofstream&);

    // reg allocate api
    rv::rvREG getRd(ir::Operand);
    rv::rvFREG fgetRd(ir::Operand);
    rv::rvREG getRs1(ir::Operand);
    rv::rvREG getRs2(ir::Operand);
    rv::rvFREG fgetRs1(ir::Operand);
    rv::rvFREG fgetRs2(ir::Operand);

    // generate wrapper function
    void gen();
    void gen_func(const ir::Function&);
    void gen_instr(const ir::Instruction&);
};
```

`void gen();` 应先处理全局变量 `program.globalVal` ，再遍历 `program.functions` 调用 `void gen_func(const ir::Function&)` 对函数进行翻译

实验四

实验目标

进行 IR 和汇编层面的优化，深入理解计算机体系结构、掌握性能调优技巧、培养系统级思维和优化能力

实验步骤

从希冀上下载实验框架

compiler [src_filename] -S -o [output_filename] O1 将输出你优化后的汇编结果至 [output_filename]

测评方法：

使用命令 `qemu-riscv32.sh ur_rv_executable` 来执行 risc-v 的可执行文件，可执行文件的执行时间会被打印到 stderr，通过此时间来观察优化效果

实验四标准输出

对链接有库函数的可执行文件，执行 `qemu-riscv32.sh ur_rv_executable` 后，最后一行会打印你的执行时间

```
TOTAL: 0H-0M-0S-0us
```

编译优化

编译优化是编译器的一个重要部分，旨在改善生成的目标代码的质量和性能。通过应用各种优化技术，可以减少程序的执行时间、减少资源消耗，并提高代码的质量和可维护性。以下是一些常见的编译优化相关技术：

1. 常量传播（Constant Propagation）：常量传播技术用于将变量或表达式中的常量值替换为其实际的值。通过在编译时进行常量传播，可以减少程序运行时的计算量，提高执行效率。
2. 复写传播（Copy Propagation）：复写传播技术用于将变量的赋值操作替换为对应的值。这样可以减少对内存的读写操作，提高程序的执行速度。
3. 死代码消除（Dead Code Elimination）：死代码消除技术用于删除程序中不会执行的代码。通过识别不会对程序结果产生影响的代码，可以减少不必要的计算和内存访问，提高程序的执行效率。
4. 循环优化（Loop Optimization）：循环优化技术针对循环结构进行优化，以减少循环的执行时间和资源消耗。常见的循环优化技术包括循环展开（Loop Unrolling）、循环合并（Loop Fusion）、循环划分（Loop Tiling）等。
5. 数据流分析（Data Flow Analysis）：数据流分析技术用于分析程序中数据的传递和变化情况，以便进行后续的优化。通过数据流分析，可以识别出不必要的计算、无用的变量等，并进行相应的优化处理。
6. 内联展开（Inline Expansion）：内联展开技术用于将函数调用处直接展开为函数体，以减少函数调用的开销。通过将函数体嵌入调用处，可以减少函数调用和返回的开销，提高程序的执行效率。
7. 代码重排（Code Reordering）：代码重排技术通过改变指令的执行顺序，以提高指令级并行性和缓存利用率。通过优化指令的顺序，可以减少指令之间的依赖关系，提高指令的执行效率。

编译器优化能力是现代编译器的核心，直接决定了生成机器码的质量好坏。

1. 基于 SSA 的中间代码优化

静态单变量赋值（Static Single Assignment，SSA）

静态单变量赋值（Static Single Assignment，简称SSA）是一种在编译器优化和中间代码表示中常用的技术。它的主要目的是通过引入额外的变量和约束，使得程序的数据流更加明确和可控，从而方便进行优化。

在 SSA 中，每个变量在程序中只被赋值一次，也就是说每个变量的赋值语句只会出现一次。为了实现这一点，编译器会进行一系列的转换操作，包括插入 φ 函数（Phi Function）和复制操作。

SSA 的优点包括：

1. 提供了明确的数据流信息，有助于进行各种编译优化。
2. 简化了编译器的数据流分析，减少了复杂度。
3. 支持更精确的内存别名分析，有助于提高代码生成的准确性和效率。

φ 函数（Phi Function）

φ 函数（Phi Function）是一种特殊的函数，用于选择不同路径上的变量版本。它通常用于控制流图中的分支节点，用于解决变量的定义在不同路径上存在的问题。

具体而言，φ 函数具有以下特点：

1. φ 函数有多个输入参数，每个参数对应一个可能的变量版本，这些版本来自于不同的路径。
2. φ 函数的结果是一个输出变量，代表根据控制流的选择从输入参数中选择的正确版本。
3. φ 函数只在控制流图中的分支节点处出现，用于合并不同路径上的变量版本。

通过插入φ 函数，SSA形式的中间表示确保了每个变量在使用点之前只有一个定义，解决了控制流分支带来的变量版本选择问题。φ 函数根据控制流的选择，选择正确的变量版本，使得在后续的数据流分析和优化中更容易进行。

以下是一个示例代码片段，说明了如何使用 φ 函数来选择正确的变量版本：

```
1: if (condition) {
2:     x = 1;
3: } else {
4:     x = 2;
5: }
6: y = φ (x1, x2); // φ  函数根据控制流的选择，选择正确的变量版本

// 后续的代码中使用 y 变量代替 x 变量
...
```

在上述代码中，根据控制流的选择，变量 x 在不同的路径上有不同的版本。在第6行的φ函数中，x1 和 x2 是来自不同路径的变量版本。根据条件判断的结果，φ函数会选择正确的变量版本作为输出，将其赋值给变量 y。

构造 SSA 形式的中间表示

构造静态单变量赋值（SSA）的中间表示通常包括以下步骤：

1. 确定基本块：首先，将程序的源代码分成基本块（Basic Blocks），每个基本块包含一系列的顺序执行语句，没有分支或跳转。
2. 构建控制流图：根据基本块之间的跳转关系构建控制流图（Control Flow Graph，简称CFG）。CFG用于表示程序中的控制流信息，包括基本块之间的控制流转移和分支条件。
3. 插入 φ 函数：对于每个变量在控制流图中的分支节点（例如if语句），需要在该节点插入 φ 函数（Phi Function）来选择正确的变量版本。φ函数的参数是来自不同路径的变量值，它会根据控制流的选择情况来选择合适的值。
4. 行变量重命名：从控制流图的入口开始，按照拓扑顺序遍历控制流图中的基本块。对于每个基本块中的变量，进行变量重命名操作。具体步骤如下：
   i. 遍历基本块的每个前驱块，获取变量在不同路径上的版本
   ii. 为变量创建新的版本，并更新变量的定义处，使其指向新版本的变量
   iii. 更新基本块中使用该变量的地方，将其替换为对应版本的变量
5. 更新使用处：在控制流图中，对于使用到变量的地方，需要根据控制流的选择情况选择正确的变量版本。这样可以保证每个变量在使用点只有一个定义。

通过以上步骤，就可以构造出SSA的中间表示。这个中间表示包含了明确的数据流信息，每个变量都有唯一的版本，并且每个变量的赋值语句只出现一次。这样便于编译器在后续的优化阶段对程序进行分析和优化。

2. 后端优化

寄存器是CPU中的稀有资源，如何高效的分配这一资源是一个至关重要的问题，如果不进行寄存器分配而把变量放在内存中，如我们在实验三中介绍的那样，你会发现你的编译器会生成大量的代码, 而且大部分都是 load/store

为了提高程序执行的效率并减少访存次数，你需要在你的编译器中实现寄存器分配

把寄存器当缓存用

一种简单的策略是把像 cache 一样使用寄存器，寄存器的生命周期局限在基本块内：

1. 首先, 在栈帧上为所有变量都分配空间.
2. 当需要用到某个变量的时候, 把这个变量读出来, 放在一个临时分配的寄存器里.
3. 下次需要读写变量时, 直接操作寄存器的值, 省去内存访问的开销.
4. 如果遇到某些情况, 比如出现了函数调用, 或者发生了控制流转移, 就把寄存器里保存的所有变量写回栈帧, 下次用的时候再重新读取.

   > 可以思考一下为什么生命周期在基本块内

基于数据流分析的寄存器分配算法

数据流分析指的是一组用来获取有关数据如何沿着程序执行路径流动的相关信息的技术

基于你构建 SSA 过程中产生的 CFG 图，可以继续进行数据流分析，以支持以下寄存器分配算法：

1. 线性扫描寄存器分配是一个线性的寄存器分配算法. 它需要按顺序扫描变量的活跃区间, 然后基于一些贪婪的策略, 把变量放在寄存器上或者栈上。这种算法只需要进行一次扫描, 就可以得到很不错的寄存器分配结果
2. 图着色寄存器分配：计算机中物理寄存器个数总是有限的，当需要一个寄存器但所有可用寄存器都在被使用时，就要将某个正在使用的寄存器溢出(spill)到内存中从而释放出一个寄存器。

> 你应该查资料来详细了解以上算法并实现