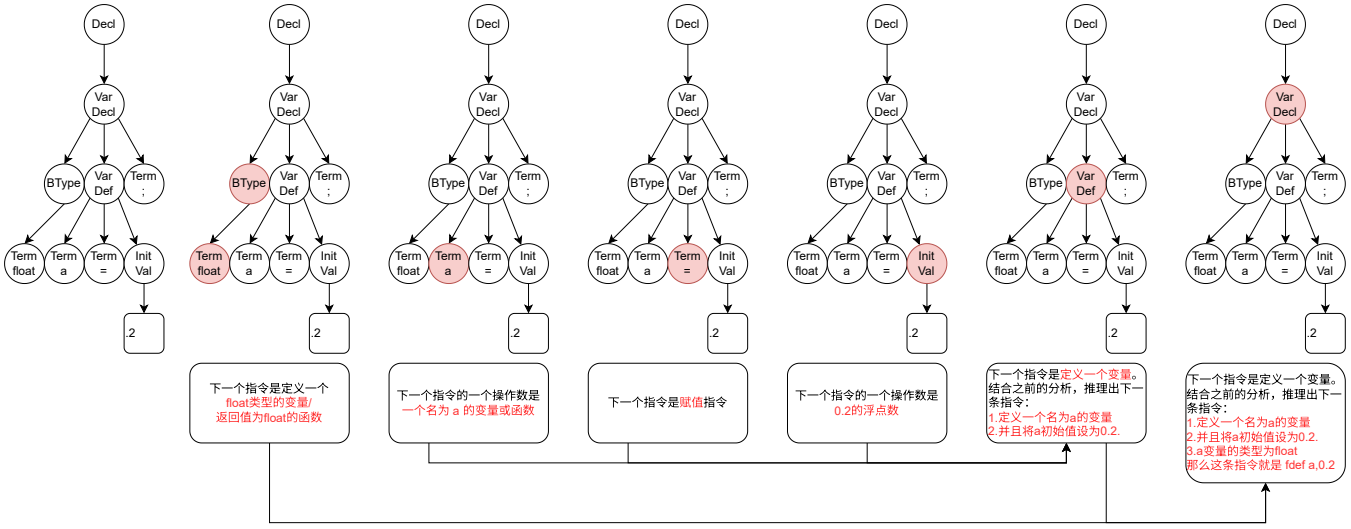
本文档为实验指导书的补充说明，请同学们优先仔细阅读实验指导书

实验二

将实验1的AST抽象语法树转换成IR（中间表示）



数据结构

IR

实验对 SysY 语言设计了统一的 IR 框架，所有 IR 采用四元的形式，即：

`opcode`, `des`, `operand1`, `operand2`

每个IR表示可以在实验指导书中查到。举例来说，你需要将形如

```
int a = 8;
```

转换为

```
def a, 8
```

这种IR表示。

入口函数与基本数据结构

在 `main.cpp` 文件中，有

```
frontend::Analyzer analyzer;
auto program = analyzer.get_ir_program(node);
```

我们需要完成这个 `get_ir_program`

函数入口在这里，`src/front/semantic.cpp`

```
ir::Program frontend::Analyzer::get_ir_program(CompUnit* root)
```

你可以发现这个函数的输入，是实验一分析得到的根节点 `CompUnit` 的指针。 让你输出的就是一个 `ir::Program`.

来看一下 `ir::Program` 的数据结构

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

}
```

总的来说，我们需要得知 `Program` 中的 `functions` 和 `globalVal` 两个变量。

当我们把一个程序的 functions 和 globalVal 都解析完成的时候，实验也就做完了！ 😊

好耶！一定很简单吧？ 🤡

Function

这个 Function 的数据结构到底是什么呢？请查看 `include/ir/ir_function.h`：

```
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();
};
```

就跟我们的程序里面的函数（main函数，add函数等等）一样，通过这些命名可以读出来一个 function 必须要有

- `name`: 函数名
- `returnType`: 返回类型，数据结构是一个Type。
- `ParameterList`: 参数列表，数据结构是一个 **Operand** 的数组。
- `InstVec`: 指令的列表，数据结构是一个 **Instruction**指针的数组。

不知道你们还记不记得，在实验一中有节点类型专门就是用来解析名字、返回类型的，是什么呢，好难猜啊？

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

Operand

顾名思义，**操作数**。

还记得刚才我们IR的具体表示为一个四元组吗？`opcode`, `des`, `operand1`, `operand2`

opcode很好理解，无非就是一个操作的枚举类型，比如有add, store, ...

这里面其余三个，op1、op2和des都是操作数。

```
namespace ir {

enum class Type {
    Int,
    Float,
    IntLiteral,
    FloatLiteral,
    IntPtr,
    FloatPtr,
    null
};

std::string toString(Type t);

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

}
```

一个 Operand 有它的类型和它的名字

类型type：

- Int：整型。比如 `int x` 这里面 x 就是一个整型。
- Float：浮点型。
- IntLiteral：整型字面量。比如`1`, `2`, `1024`这种。
- FloatLiteral：浮点型字面量。
- IntPtr：整型指针，数组需要用。
- FloatPtr：浮点型指针。

比如 `int x = 2;` 这条指令，`x` 是一个 Int 的操作数，`1` 是一个 IntLiteral 的操作数。

Instruction

怎么定义一条指令？在`ir_instruction.h`中有：

```
struct Instruction {
    Operand op1;  // 操作数1
    Operand op2;  // 操作数2
    Operand des;  // 结果
    Operator op;  // 操作
    Instruction();
    Instruction(const Operand& op1, const Operand& op2, const Operand& des, const
Operator& op);
    virtual std::string draw() const;
};
```

就跟刚才的四元组一样，我们需要照着这个来生成指令。

比如：

```
int x = a + b;  // 假设a, b 都是int类型
```

这条指令，你就需要构造类似这样的Instruction

```
auto inst = ir::Instruction{Operand{"a", Type::Int}, Operand{"b", Type::Int},
Operand{"x", Type::Int}, Operator::add}
```

回顾一下，一个Function有它的name，returnType，参数列表和一堆指令。而这些数据我们现在都可以在某些节点获得，通过这些节点来完成这个function，我们的任务就完成了。

globalVal

全局变量，定义为这样：

```
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);
    };
```

有两个数据，一个操作数，和一个maxlen，记录数组长度，当不是数组时值为0.

全局变量，顾名思义，我们可以在全局中访问到这些操作数。与之一个相关的概念是作用域。

scope（作用域）

回顾一下程序设计基础，假设有这么一段程序

```
int main() {
    int a = 1;
    if (true) {
        int a = 3;
        while (true) {
            if (a == 5) {
```

```
                break;
            }
            a++;
        }
    }
    cout << a << endl;
    return 0;
}
```

这里打印的值会是多少？运行程序跑出来是 `1`，这是因为作用域的缘故。

在进入一个block（大括号括起来的区域），变量的作用域会发生变更，也就是我们讨论变量的上下文发生了切换。

执行 `int a = 3` 时，已经是在一个新的作用域发生的事情了，直到遇到大括号的末尾退出block，才退回到了原来的作用域。

仔细想想，这跟我们学过的一个数据类型很相似，当遇到一个 `{` 进入新的作用域，访问变量时去获取这个变量存在的最新作用域。

没错，这是一个**栈**。

在程序中，有一个数据结构叫**符号表**，我们可以在这里看到相关的数据和函数：

```
// definition of symbol table
struct SymbolTable {
    vector<ScopeInfo> scope_stack;
    map<std::string, ir::Function*> functions;
    int scope_cnt;

    void add_scope();
    void exit_scope();

    string get_scoped_name(string id) const;
    ir::Operand get_operand(string id) const;
    STE get_ste(string id) const;
    void add_ste(const string& id, STE ste);
};
```

这里的scope_stack就是一个作用域的栈。

符号表是一个在实验二中重要的数据结构，请根据以上内容自行研究。

算法

该从哪里入手

我们最终要写的函数：

```
ir::Program frontend::Analyzer::get_ir_program(CompUnit* root){
    ir::Program program;
    // do something
    // ...
    reutrn program;
}
```

不妨我们先来试一下这个程序

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

它的AST应该是

```
{
   "name" : "CompUnit",
   "subtree" : [
      {
         "name" : "FuncDef",
         "subtree" : [
            {
               "name" : "FuncType",
               "subtree" : [
//...
```

一开始，传进来的时候一个CompUnit*，它的文法是这样的

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

我们要去分析这个节点，就和实验一一样，**整个过程是递归的。**

我们定义一个 `analysisCompUnit` 函数：

```
// CompUnit -> (Decl | FuncDef) [CompUnit]
void frontend::Analyzer::analysisCompUnit(CompUnit* root,
                                          ir::Program& program) {
    if (Decl* node = dynamic_cast<Decl*>(root->children[0])) { // Decl
        vector<Instruction*> decl_insts;
        analysisDecl(node, decl_insts);
        for (auto& inst : decl_insts) {
            symbol_table.functions["global"]->addInst(inst);
        }
    } else if (FuncDef* node = dynamic_cast<FuncDef*>(root->children[0])) {
        Function* new_func = new Function();
        analysisFuncDef(node, new_func);
        program.addFunction(*new_func);
    } else {
        assert(0 && "Unknown node type");
    }
    if (root->children.size() == 2) {
```

```
        GET_CHILD_PTR(child_comp, CompUnit, 1);
        analysisCompUnit(child_comp, program);
    }
}
```

`root->children[0]` 只有两种情况，一个是 `Decl` 节点，一个是 `FuncDef` 节点。我们该怎么做判断，dynamic_cast<Decl*>(root->children[0]) 这种方式，其含义是，将 `root->children[0]` 动态转换为 `Decl*` 的类型，如果不可以转换，`node` 则是一个 `nullptr`，`nullptr` 的bool值为`false。所以等价于做了一个类型判断。

回到程序中，我们的程序在这一个节点会得到其第一个子节点的类型是 `FuncDef`，于是在这里进入第二个逻辑分支。此时我们要主动new 一个新的function（因为已经要函数定义了），我们继续写 `analysisFuncDef` 函数。

```
// FuncDef -> FuncType Ident '(' [FuncFParams] ')' Block
void frontend::Analyzer::analysisFuncDef(frontend::FuncDef* root,
                                         Function* func) {
    symbol_table.add_scope();
    GET_CHILD_PTR(functype, FuncType, 0);
    GET_CHILD_PTR(ident, Term, 1);
    GET_IDENFR_NAME(id, ident);
    // ...
    symbol_table.exit_scope();
}
```

在 `semantic.cpp` 中，写好了几个宏可以调用，比如：

```
#define GET_CHILD_PTR(node, type, index)                                \
    auto node = static_cast<type*>(root->children[index])
```

GET_CHILD_PTR(functype, FuncType, 0); 其含义是，我需要把第一个子节点赋值到 `functype` 这个变量里，其类型是FuncType

此时 `functype` 就是一个节点指针了。

还有一个宏是：

```
#define GET_IDENFR_NAME(_id, _term)                                     \
    std::string _id;                                                    \
    _id = _term->token.value
```

GET_IDENFR_NAME(id, ident); 其含义是，我需要把ident（是一个`Term*`）的值放在id这里，id是一个string类型。

也就是说，通过分析GET_IDENFR_NAME，我们获得了id其值为`main`

你要写每一个节点的analyze函数，形如：

```
// analysis functions
void analysisCompUnit(CompUnit*, ir::Program&);
void analysisDecl(Decl*, vector<ir::Instruction*>&);
void analysisConstDecl(ConstDecl*, vector<ir::Instruction*>&);
void analysisVarDecl(VarDecl*, vector<ir::Instruction*>&);
void analysisConstDef(ConstDef*, ir::Type, vector<ir::Instruction*>&);
void analysisVarDef(VarDef*, ir::Type, vector<ir::Instruction*>&);
```

```
void analysisFuncDef(FuncDef*, ir::Function*);
void analysisFuncFParams(FuncFParams*, ir::Function*);
void analysisFuncFParam(FuncFParam*, ir::Function*);
void analysisBlock(Block*, vector<ir::Instruction*>&, bool);
void analysisBlockItem(BlockItem*, vector<ir::Instruction*>&);
void analysisStmt(Stmt*, vector<ir::Instruction*>&);

void analysisCond(Cond*, vector<ir::Instruction*>&);

Type analysisBType(BType*);
Type analysisFuncType(FuncType*);

void
    analysisConstInitVal(ConstInitVal*, ir::Type, vector<ir::Instruction*>&);
void analysisInitVal(InitVal*, ir::Type, vector<ir::Instruction*>&);
void analysisConstExp(ConstExp*, vector<ir::Instruction*>&);
void analysisAddExp(AddExp*, vector<ir::Instruction*>&);
void analysisMulExp(MulExp*, vector<ir::Instruction*>&);
void analysisUnaryExp(UnaryExp*, vector<ir::Instruction*>&);
void
    analysisFuncRParams(FuncRParams*, vector<ir::Instruction*>&, ir::Function*);
void analysisPrimaryExp(PrimaryExp*, vector<ir::Instruction*>&);
void analysisExp(Exp*, vector<ir::Instruction*>&);
void analysisLVal(LVal*, vector<ir::Instruction*>&, bool);
void analysisNumber(Number*);

void analysisLOrExp(LOrExp*, vector<ir::Instruction*>&);
void analysisLAndExp(LAndExp*, vector<ir::Instruction*>&);
void analysisEqExp(EqExp*, vector<ir::Instruction*>&);
void analysisRelExp(RelExp*, vector<ir::Instruction*>&);
```

其中有些接口参数是我的程序需要用的，可以根据实际情况调整。从 `analysisCompUnit` 函数**自顶向下**调用，过程中会依次分析到function，inst等等，把这些加入到program中，最后返回program即可。

一些处理

这些处理非必要，也可以有自己的实现方式。

函数作用域

正如刚才所说，当进入分析FuncDef节点时，函数就进入了一个新的作用域，这是因为函数参数也在这个作用域当中

```
int f(int a, int b){
    int c = 1;
    return 100;
}
```

当我进入一个函数定义的大括号时，便不再进入一个新的作用域，此时 a, b, c 在一个作用域里。

```
{
    {"a", <传过来的值>},
    {"b", <传过来的值>},
    {"c", 100},
}
```

所以这里的

```
void analysisBlock(Block*, vector<ir::Instruction*>&, bool);
```

多了一个bool类型，表明是否是函数定义的block，如果是就不增加新的scope。除此以外，遇到一个 block 便加一个 scope

外部库函数

因为测评需要根据输入来输出结果，所以要支持输入输出的外部库函数，你需要将这个map里面的function添加到的你的符号表中。

```
map<std::string, ir::Function*>* frontend::get_lib_funcs() {
    static map<std::string, ir::Function*> lib_funcs = {
        {"getint", new Function("getint", Type::Int)},
        {"getch", new Function("getch", Type::Int)},
        {"getfloat", new Function("getfloat", Type::Float)},
        {"getarray",
         new Function("getarray", {Operand("arr", Type::IntPtr)}, Type::Int)},
        {"getfarray",
         new Function(
             "getfarray", {Operand("arr", Type::FloatPtr)}, Type::Int)},
        {"putint",
         new Function("putint", {Operand("i", Type::Int)}, Type::null)},
        {"putch", new Function("putch", {Operand("i", Type::Int)}, Type::null)},
        {"putfloat",
         new Function("putfloat", {Operand("f", Type::Float)}, Type::null)},
        {"putarray",
         new Function("putarray",
                      {Operand("n", Type::Int), Operand("arr", Type::IntPtr)},
                      Type::null)},
        {"putfarray",
         new Function("putfarray",
                      {Operand("n", Type::Int), Operand("arr", Type::FloatPtr)},
                      Type::null)},
    };
    return &lib_funcs;
}

frontend::Analyzer::Analyzer() : tmp_cnt{0}, symbol_table() {
    symbol_table.add_scope();
    ir::Function* global_func = new ir::Function("global", ir::Type::null);
    symbol_table.functions.insert({"global", global_func});
    auto lib_funcs = *get_lib_funcs();
    for (const auto& pair : lib_funcs) {
```

```
34          symbol_table.functions.insert(pair);
35      }
36  }
```

Global函数

参考指导书。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。