并行编译与优化

Advanced Compiler Technology

计算机研究所编译室

Experiment One: Generate IR from AST

实验2：从抽象语法树生成中间表示

实验内容

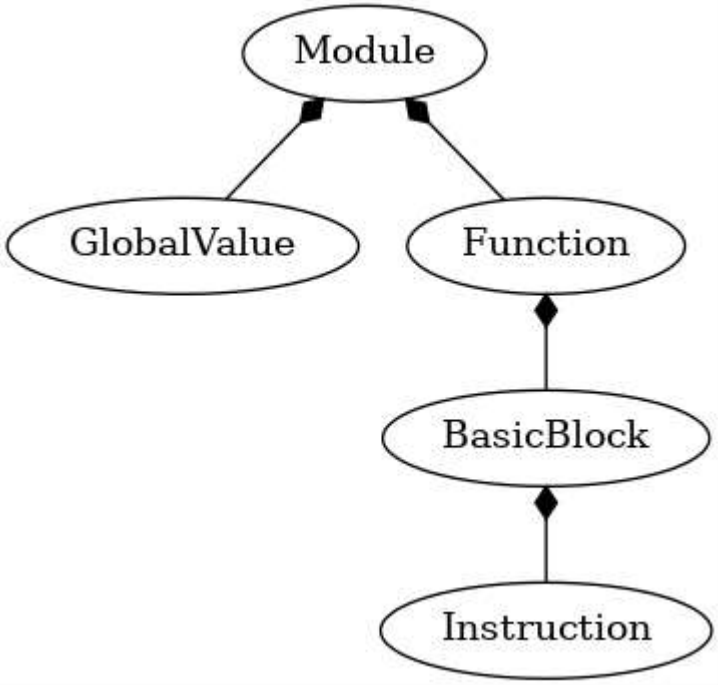
- 掌握SysY IR的定义与数据结构
- 从AST输出IR（基于visitor机制）

实验内容1

SysY IR的定义与数据结构

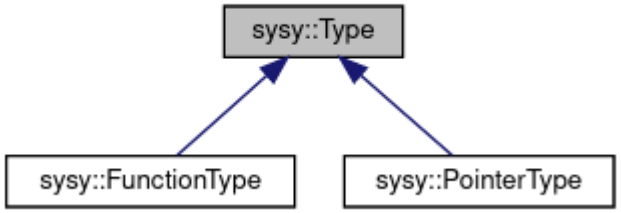
SysY IR的基本结构

- Module
  - 顶层的IR结构，对应CompUnit
- GlobalValue
  - 全局变量Decl
- Function
  - 函数定义FuncDef
- BasicBlock
  - 基本块，由If-else、while控制结构引入
- Instruction
  - 算数逻辑运算、比较、类型转换操作



类型系统

- SysIR使用一个简单的类型系统
- 基类Type用于表示基础数据类型
  - int、float、void、label
- 派生类FunctionType表示函数类型
  - 包含返回类型与参数类型信息
- 派生类Pointer表示指针类型（内存地址）
  - 包含其指向的类型信息



Type类型

- 使用一组静态方法获取Type *
  - getXXXType
- 判断是否为同一类型可直接使用指针比较
  - type1 == type2

```
34  class Type {
35  public:
36    enum Kind {
37      kInt,
38      kFloat,
39      kVoid,
40      kLabel,
41      kPointer,
42      kFunction,
43    };
44    Kind kind;
45
46  protected:
47    Type(Kind kind) : kind(kind) {}
48    virtual ~Type() {}
49
50  public:
51    static Type *getIntType();
52    static Type *getFloatType();
53    static Type *getVoidType();
54    static Type *getLabelType();
55    static Type *getPointerType(Type *baseType);
56    static Type *getFunctionType(Type *returnType,
57                                 const std::vector<Type *> &paramTypes = {});
58
59  public:
60    Kind getKind() const { return kind; }
61    bool isInt() const { return kind == kInt; }
62    bool isFloat() const { return kind == kFloat; }
63    bool isVoid() const { return kind == kVoid; }
64    bool isLabel() const { return kind == kLabel; }
65    bool isPointer() const { return kind == kPointer; }
66    bool isFunction() const { return kind == kFunction; }
67    int getSize() const;
68  }; // class Type
```

Value类型——“值”

- Value类型是SysY IR的核心
  - 大多数IR数据结构都是Value的派生类
- SysY程序中的所有“值”都是Value
  - 变量、常量
  - 函数
  - 地址
- Value的主要属性
  - 类型、名称、Use

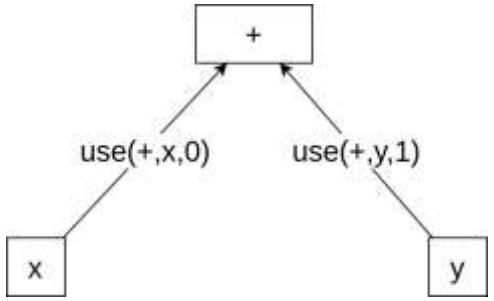
```
167  //! The base class of all value types
168  class Value {
169  protected:
170    Type *type;
171    std::string name;
172    std::list<Use *> uses;
173
174  protected:
175    Value(Type *type, const std::string &name = "")
176        : type(type), name(name), uses() {}
177    virtual ~Value() {}
178
179  public:
180    Type *getType() const { return type; }
181    bool isInt() const { return type->isInt(); }
182    bool isFloat() const { return type->isFloat(); }
183    bool isPointer() const { return type->isPointer(); }
184    const std::list<Use *> &getUses() { return uses; }
185    void addUse(Use *use) { uses.push_back(use); }
186    void replaceAllUsesWith(Value *value);
187    void removeUse(Use *use) { uses.remove(use); }
188  }; // class Value
```

Use

- 数据的价值在于被使用
  - 算数指令的操作数
  - 分支跳转的目标
  - 函数调用的目标函数

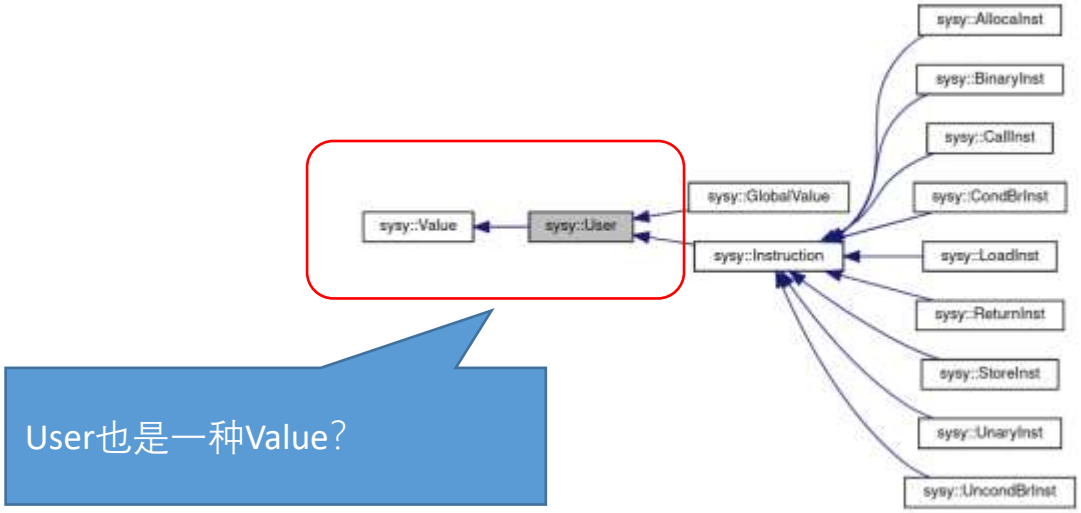
```
151 //! `Use` represents the relation between a `Value` and its `User`
152 class Use {
153 private:
154   //! the position of value in the user's operands, i.e.,
155   //! user->getOperands[index] == value
156   int index;
157   User *user;
158   Value *value;
159
160 public:
161   Use() = default;
162   Use(int index, User *user, Value *value)
163       : index(index), user(user), value(value) {}
164
165 public:
166   int getIndex() const { return index; }
167   User *getUser() const { return user; }
168   Value *getValue() const { return value; }
169   void setValue(Value *value) { value = value; }
170 }; // class Use
```

- Use表示Value的“使用”情况
  - value：被使用的数据
  - user：数据的使用者
  - index：表示value是user所有操作数中的第index个



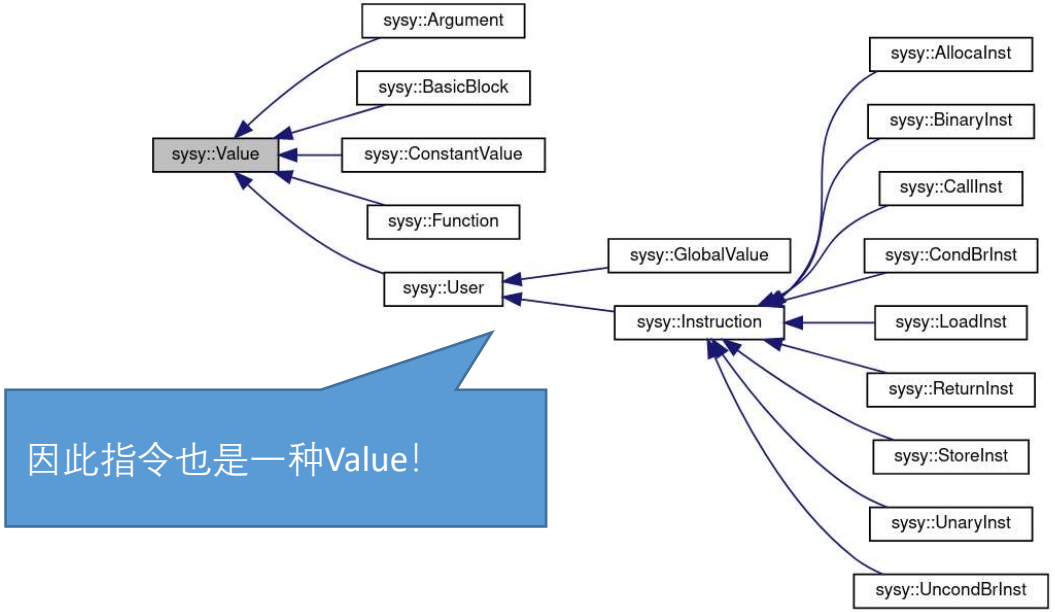
User

- User是一个抽象类型
  - IR中需要使用其他“值”作为输入的类型均为User
- 当前IR中有两类User
  - GlobalValue：全局变量/常量的定义中，可能引用其他字面量、全局常量
  - Instruction：算数逻辑运算、比较运算、分支跳转指令等



Value再审视

- （Almost） everything is Value!
- IR采用Static-Single-Assignment（SSA）设计
  - Value不同于三地址码中的寄存器的概念
  - 指令可以定义（作为输出结果）新的Value，但不可以对已有的Value再赋值
  - SysY IR指令设计中，每个Instruction最多有一个输出结果，这个输出的值与指令是一一对应的



SSA Value的合并问题

```
int a, b;
if (x)
  a = a * 2;
else
  a = a * 3;
b = a;
```

```
  %aptr = alloca : int
  %bptr = alloca : int
  %flag = cmpeq x 0
  br %flag %then %else
then:
  %a0 = load %aptr
  %double = mul %a 2
  br %exit
else
  %a1 = load %aptr
  %triple = mul %a 3
  br %exit
exit:
  store %bptr ?
```

写回哪个值?

SSA Value的合并问题

```
int a, b;
if (x)
  a = a * 2;
else
  a = a * 3;
b = a;
```

```
  %aptr = alloca : int
  %bptr = alloca : int
  %flag = cmpeq x 0
  br %flag %then %else
then:
  %a0 = load %aptr
  %double = mul %a 2
  store %aptr, %double
  br %exit
else
  %a1 = load %aptr
  %triple = mul %a 3
  store %aptr, %triple
  br %exit
exit:
  %a_new = load %aptr
  store %bptr %a_new?
```

通过内存读写解决

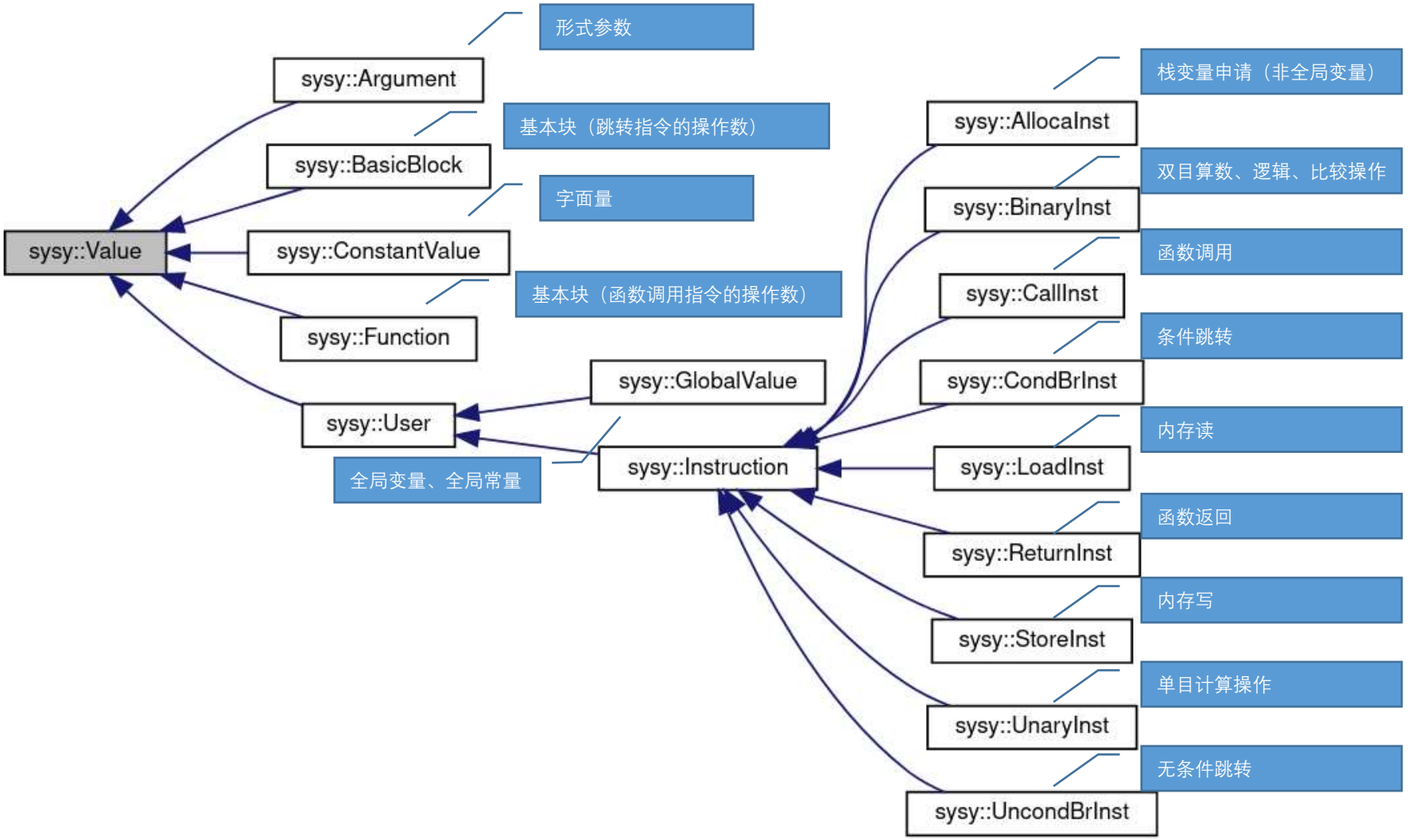
SSA Value的合并问题

```
int a, b;
if (x)
  a = a * 2;
else
  a = a * 3;
b = a;
```

```
  %aptr = alloca : int
  %bptr = alloca : int
  %flag = cmpeq x 0
  br %flag %then %else
then:
  %a0 = load %aptr
  %double = mul %a 2
  br %exit %double
else
  %a1 = load %aptr
  %triple = mul %a 3
  br %exit %triple
exit (%a_new : int):
  store %bptr %a_new?
```

通过在基本块之间传递参数解决

具体的Value类型


形式参数
sysy::Argument
基本块（跳转指令的操作数）
sysy::BasicBlock
字面量
sysy::ConstantValue
基本块（函数调用指令的操作数）
sysy::Function
sysy::Value
sysy::GlobalValue
sysy::User
全局变量、全局常量
sysy::Instruction
栈变量申请（非全局变量）
sysy::AllocaInst
双目算数、逻辑、比较操作
sysy::BinaryInst
函数调用
sysy::CallInst
条件跳转
sysy::CondBrInst
内存读
sysy::LoadInst
函数返回
sysy::ReturnInst
内存写
sysy::StoreInst
单目计算操作
sysy::UnaryInst
无条件跳转
sysy::UncondBrInst

实验内容2

从AST生成IR

实施思路

- 自定义一个SysYBaseVisitor的派生类，自顶向下地遍历AST,依次完成所有语法结构的IR生成
  - visitCompUnit -> 建立Module
  - visitFuncDef -> 在Module中创建Function
  - visitBlockStmt -> 在Function中创建BasicBlock
  - visitBlockItem -> 在BasicBlock中创建Instruction
  - visitIfStmt -> 创建新的BasicBlock并创建跳转指令
  - ……

Module生成

- 访问AST根节点时，新建一个Module对象，
- 递归向下为每个全局变量/常量生成IR

```
9  any SysYIRGenerator::visitModule(SysYParser::ModuleContext *ctx) {
10   auto pModule = new Module();
11   assert(pModule);
12   module.reset(pModule);
13   visitChildren(ctx);
14   return pModule;
15 }
```

GlobalValue的生成

- GlobalValue对应静态数据区的一个内存单元，其类型为指针类型
  - 例如，`int a = 1`，对应的GlobalValue为`int *`
  - 这一点对于局部变量也适用
- SysY语言中对变量的读和写可以分别用Load和Store指令实现

Function生成

- 获取函数名
- 获取返回值类型以及形参类型，从而创建相应的函数类型
- 在Module中创建一个新的函数
- 按照函数的形参定义，在函数的入口基本块中创建形式参数

```
17  any SysYIRGenerator::visitFunc(SysYParser::FuncContext *ctx) {
18    auto name = ctx->ID()->getText();
19    auto params = ctx->funcFParams()->funcFParam();
20    vector<Type *> paramTypes;
21    vector<string> paramNames;
22    for (auto param : params) {
23      paramTypes.push_back(any_cast<Type *>(visitBtype(param->btype())));
24      paramNames.push_back(param->ID()->getText());
25    }
26    Type *returnType = any_cast<Type *>(visitFuncType(ctx->funcType()));
27    auto funcType = Type::getFunctionType(returnType, paramTypes);
28    auto function = module->createFunction(name, funcType);
29    auto entry = function->getEntryBlock();
30    for (auto i = 0; i < paramTypes.size(); ++i)
31      entry->createArgument(paramTypes[i], paramNames[i]);
32    builder.setPosition(entry, entry->end());
33    visitBlockStmt(ctx->blockStmt());
34    return function;
35  }
```

Decl生成

- 获取变量类型
- 一个Decl可以声明多个变量
  - 逐个处理
- 获取变量名，生成alloca指令，得到栈变量的指针
- 若有初始化值，创建store指令将初始化值写入变量内存

```
50  any SysYIRGenerator::visitDecl(SysYParser::DeclContext *ctx) {
51    std::vector<Value *> values;
52    auto type = any_cast<Type *>(visitBtype(ctx->btype()));
53    for (auto varDef : ctx->varDef()) {
54      auto name = varDef->lValue()->ID()->getText();
55      auto alloca = builder.createAllocaInst(type, {}, name);
56      if (varDef->ASSIGN()) {
57        auto value = any_cast<Value *>(varDef->initValue()->accept(this));
58        auto store = builder.createStoreInst(value, alloca);
59      }
60      values.push_back(alloca);
61    }
62    return values;
63  }
```

各类语句的生成

- AssignStmt
  - 先为右手侧生成代码，得到右手侧表达式对应的Value
  - 创建Store指令，将右手侧对应Value写入内存
- BlockStmt
  - 建立新的基本块，将IRBuilder的插入点更新为新的基本块的头部位置
  - 由于BlockStmt会建立新的Scope，需要更新符号表

各类语句的生成

- IfStmt
  - 先生成条件表达式的IR，获取条件表达式对应的Value
  - 创建两个新的BasicBlock，分别对应then分支与else分支
  - 创建条件跳转指令
  - 分别为两个分支block产生代码
- WhileStmt
  - 创建一个新的基本块（Header），在新的基本块中生成条件表达式的IR，获取条件表达式对应的Value
  - 再创建两个新的基本块（Body与Exit），分别对应循环内部与循环体后的代码
  - 在Exit尾部创建向Header跳转的无条件分支语句
  - 为Body创建代码

符号表

- 符号表记录名称相关的信息，在IR生成中的应用包括
  - 通过变量名查询变量在内存上的位置（一个指针类型的Value）
  - 通过函数名查询函数对应的IR对象（一个函数类型的Value）
- 符号表与作用域密切相关，可以组织成一个堆栈

SysY语法规范

编译单元	CompUnit	→	[CompUnit] (Decl | FuncDef)
声明	Decl	→	ConstDecl | VarDecl
常量声明	ConstDecl	→	**'const'** BType ConstDef { ',' ConstDef } ';'
基本类型	BType	→	**'int'** | **'float'**
常数定义	ConstDef	→	**Ident** { '[' ConstExp ']' } '=' ConstInitVal
常量初值	ConstInitVal	→	ConstExp
			| '{' [ConstInitVal { ',' ConstInitVal }] '}'
变量声明	VarDecl	→	BType VarDef { ',' VarDef } ';'
变量定义	VarDef	→	**Ident** { '[' ConstExp ']' }
			| **Ident** { '[' ConstExp ']' } '=' InitVal
变量初值	InitVal	→	Exp | '{' [InitVal { ',' InitVal }] '}'
函数定义	FuncDef	→	FuncType **Ident** '(' [FuncFParams] ')' Block
函数类型	FuncType	→	**'void'** | **'int'** | **'float'**
函数形参表	FuncFParams	→	FuncFParam { ',' FuncFParam }
函数形参	FuncFParam	→	BType **Ident** ['[' ']' { '[' Exp ']' }]
语句块	Block	→	'{' { BlockItem } '}'
语句块项	BlockItem	→	Decl | Stmt
语句	Stmt	→	LVal '=' Exp ';' | [Exp] ';' | Block
			| **'if'** '(' Cond ')' Stmt [**'else'** Stmt]
			| **'while'** '(' Cond ')' Stmt
			| **'break'** ';' | **'continue'** ';'
			| **'return'** [Exp] ';'
表达式	Exp	→	AddExp　注：SysY 表达式是 int/float 型

			表达式
条件表达式	Cond	→	LOrExp
左值表达式	LVal	→	**Ident** {'[' Exp ']'}
基本表达式	PrimaryExp	→	'(' Exp ')' | LVal | Number
数值	Number	→	**IntConst** | **floatConst**
一元表达式	UnaryExp	→	PrimaryExp | **Ident** '(' [FuncRParams] ')'
			| UnaryOp UnaryExp
单目运算符	UnaryOp	→	'+' | '−' | '!'　注：'!'仅出现在条件表达式中
函数实参表	FuncRParams	→	Exp { ',' Exp }
乘除模表达式	MulExp	→	UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
加减表达式	AddExp	→	MulExp | AddExp ('+' | '−') MulExp
关系表达式	RelExp	→	AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
相等性表达式	EqExp	→	RelExp | EqExp ('==' | '!=') RelExp
逻辑与表达式	LAndExp	→	EqExp | LAndExp '&&' EqExp
逻辑或表达式	LOrExp	→	LAndExp | LOrExp '||' LAndExp
常量表达式	ConstExp	→	AddExp　注：使用的 Ident 必须是常量

实验内容

- 掌握SysY IR的定义与数据结构
- 从AST输出IR（基于visitor机制）

Let’s Go!