写在前面

前言

这个文档的目的在于对编译原理实验二的内容进行一个入门的介绍，介绍实验二的代码需要干什么。由于作者水平有限，以及对框架内容的不熟，所以如果某些内容**如果与实验文档有冲突，以实验文档为准**；同时内容的正确性请**谨慎对待。**以及，**在动手前，务必先看Tips一节。**这一节介绍了很多作者遇到的坑，也许一个小tips能节省你几个小时的debug时间。

参考资料

实验文档：Lab2 · GitBook

前置内容

请确保已**充分阅读**实验文档，并已经**充分了解**实验内容与框架**（尤其是ir::下面的那五个头文件的东西）**，并**充分明白**SysY语言文法的含义。

关于表达式-这个实验要做什么?

拿到这个实验，不同于实验一，刚开始上手很容易抓瞎，然后就开始看编译原理，看会语法制导，然后看看运行环境，中间变量表示...然而看完之后，对实验还是没头绪。。。我们先来看看这个实验的根本目的：给定实验一的AST树，生成IR表示。IR是什么呢？中间表示。其实我们可以认为，我们**本来要做的工作是用AST生成汇编**（即实验三的RISC-V），但直接生成会有各种各样的问题（最明显的，汇编语言细节很多，工作量很大），因此实验设计了一种IR，可以认为IR是介于底层汇编和高级语言之间的一种语言。因此，实验二的最终的目的是：**给定AST，生成IR。**

从一个简单的例子开始

首先我们不试图引入各种前置专业知识，我们的目的只有一个：**不择手段**地完成AST->IR的任务。而为了完成这个任务，我们会用到框架里面的工具，后面我们将会一步步引入。

首先考虑这样一个表达式：

$$1 + a * b$$

我们的语法有：

```
AddExp -> MulExp {'+' MulExp}
```

因此，我们希望解析完AddExp节点下的子树后，我们能够生成若干条IR指令来计算这个表达式。但问题来了：用IR的时候，会出现什么问题？

首先，我们需要的两个MulExp，即`1`和`a*b`，这两个表达式的值对应的变量是哪个？考虑到语法树自顶向上的过程，我们在处理`AddExp`的时候，必然生成了计算MulExp的IR，并将这个结果存放在一个变量中。**在源程序中没有生成的变量，而在IR中，因为要计算某些东西产生的变量，我们称为临时变量。**`a*b`的值，在**IR层面**上，我们必然会将结果存放在某个值中。根据IR 定义，计算`a*b`的IR表示如下：

```
temp = a * b; => mul temp, a, b
```

注意到这个`temp`并没有在源程序中出现，因此他是临时变量。因此，在AddExp这个节点的任务，实际是产生IR，计算$1 + temp$的值。

好了，我们稍微缓一下，先停一下，这似乎很简单，但牢记我们的任务：**给定AST，生成IR。**假设我现在就马上让你实现`void analysisAddExp(AddExp* root, <ir::Instruction*>& buffer)`，你现在不得不考虑很多问题。下面我们先来讨论一个最重要的问题：临时变量的获取与管理。

节点属性与临时变量管理

我们首先来想一个最重要的问题：因为要生成这个IR，我们肯定是需要知道`temp`这个变量名的，但我们该如何获取这个变量名呢？一个可行的想法是使用一个后面将被提到的被称为符号表的数据结构，但作者在实现的时候并没有使用符号表，因为符号表有更重要的事情去干。作者注意到，在`abstract_syntax_tree.h`这个头文件中，`AddExp`多了两个成员变量：`.v`和`.t`。这两个变量有什么作用？注意到`.t`是`ir::Type`类型，`.v`是`std::string`类型。而在`MulExp`当中，同样也有这个成员变量。注意这个`.v`是`string`类型，用这个来传递临时变量名是个很不错的想法。因此，我们有代码：

```
// Warning: 代码不保证正确，但尽量表示需要做的内容
// 以后不再重复标明这些宏
#define GET_CHILD_PTR(node, type, index) \
    auto node = dynamic_cast<type*>(root->children[index]);\
    assert(node);
#define ANALYSIS(node, type, index) \
    auto node = dynamic_cast<type*>(root->children[index]); \
    assert(node); \
    analysis##type(node, buffer);
#define COPY_EXP_NODE(from, to) \
    to->is_computable = from->is_computable; \
    to->v = from->v; \
    to->t = from->t;
#define CVT_ND_OPD(node) Operand(node->v, node->t)
void analysisAddExp(AddExp* root, <ir::Instruction*>& buffer){
    // 注意函数的buffer是引用，同时是个ir::instruction指针的向量，很明显可以看出，他保持一系列生成的IR指令
    ANALYSIS(mulExp1, MulExp, 0);
    ANALYSIS(mulExp2, MulExp, 2);
    // 这里的临时变量我们采取用前缀加子树的方式来标识临时变量，同时注意到临时变量前面加了一个$符号，这是有好处的。
    root->v = "$addExp" + mulExp1->v + mulExp2->v;
    // 这里的ProcessOperandType表示对类型的处理，我们并不保证送上来的一定是两个整数或者两个浮点数，毕竟ir::Type有六种主要情况
    root->t = ProcessOperandType(mulExp1->t, mulExp2->t);
    // 我们在这里假设只有两个变量，并且不会有指针来
    Instruction* addInst = new Instruction(CVT_ND_OPD(mulExp1), CVT_ND_OPD(mulExp2), CVT_ND_OPD(root), Operator::add);
    buffer.push_back(addInst);
}
```

以上代码做出了很多理想的假设：子树只有两个`mulExp`变量而不是一个或者多个，传上来的操作数一定是整数或字面量，而不会是万恶的指针。但实际情况远比这个复杂，但不要紧，我们先慢慢来。下面我们可以稍微修改一下，使得其能符合多个`mulExp`的情况。

```
// Warning: 代码不保证正确，但尽量表示需要做的内容
void analysisAddExp(AddExp* root, <ir::Instruction*>& buffer){
```

```
    ANALYSIS(mulExp, MulExp, 0);
    // 注意这个COPY_EXP_NODE相当重要，其重要性在常数优化和写FuncRParams节点的时候可以知道。这个COPY可以把指针传上去。
    COPY_EXP_NODE(mulExp1, root);
    if (root->children.size() == 1){
        return;
    }
    root->v = "$add" + mulExp1->v;
    // 这里我们准备将"$add" + mulExp1->v作为存放我们最终计算结果的临时变量
    buffer.push_back(new Instruction(CVT_ND_OPD(mulExp), Operand(), CVT_ND_OPD(root), Operator::mov));
    for (int index = 2; index < root->children.size(); i += 2){
        ANALYSIS(mulExp, mulExp, i);
        // 特别特别提醒：这里写的很明显是有问题的，在实际情况中，你还要考虑解析mulExp传上来的究竟是什么玩意，他可能是个字面量，也可能是个指针，还要考虑是个整型还是浮点数，有隐式类型转换
        // 特别特别提醒：这里写的很明显是有问题的，在实际情况中，你还要考虑解析mulExp传上来的究竟是什么玩意，他可能是个字面量，也可能是个指针，还要考虑是个整型还是浮点数，有隐式类型转换
        Instruction* addInst = new Instruction(CVT_ND_OPD(root), CVT_ND_OPD(mulExp), CVT_ND_OPD(root), Operator::add);
        buffer.push_back(addInst);
    }
}
```

类型转换

上面这一节提到，我们处理操作数时要特别特别小心，因为可能有各种各样的Type的操作数传上来。根据作者的经验，**强烈建议实现这样一个函数**，专门处理类型转换的问题。

```
ir::Operand operandTypeCast(Operand op, ir::Type desType, <ir::instruction*>& buffer){
    // the implementation is omitted.
}
```

于是，我们可以继续改写函数。

```
// Warning: 代码不保证正确，但尽量表示需要做的内容
void analysisAddExp(AddExp* root, <ir::Instruction*>& buffer){
    ANALYSIS(mulExp, MulExp, 0);
    // 注意这个COPY_EXP_NODE相当重要，其重要性在常数优化和写FuncRParams节点的时候可以知道。这个COPY可以把指针传上去。
    COPY_EXP_NODE(mulExp1, root);
    if (root->children.size() == 1){
        return;
    }
    root->v = "$add" + mulExp1->v;
    buffer.push_back(new Instruction(CVT_ND_OPD(mulExp), Operand(), CVT_ND_OPD(root), Operator::mov));
    // 直到这里，root的类型可能是浮点数，整数，指针，字面量和程序预定义的变量，情况要充分考虑到
    // ProcessTargetType(ir::Type)用于处理root的Type，这里的思路是将root处理成累积变量，目标类型是Type::Int或Type::Float.
    Type desType = ProcessTargetType(root->t);
    Operand cumOperand = operandTypeCast(CVT_ND_OPD(root), desType, buffer);
    root->v = cumOperand.name;
    root->t = desType;
```

```
    for (int index = 2; index < root->children.size(); i += 2){
        ANALYSIS(mulExp, mulExp, i);
        // 其中TYPE_FLOATSET可以看成宏定义，判断该类型是否是Float, FloatLiteral和
FloatPtr类型之一
        Type desType = root->t == Type::Float ? Type::Float :
TYPE_FLOATSET(mulExp.v) ? Type::Float : Type::Int;
        if (root->t == Type::Int && desType == Type::Float){
            std::string cvtRootName = "$cvt" + root->v;
            buffer.push_back(new Instruction(CVT_ND_OPD(root), Operand(),
Operand(cvtRootName, Type::Float), Operator::cvt_i2f));
        }
        ir::Operand succeedOperand = operandTypeCast(CVT_ND_OPD(mulExp),
desType, buffer);
        ir::Operator op = root->t == Type::Int ? Operator::add : Operator::fadd;
        ir::Instruction* addInst = new Instruction(CVT_ND_OPD(root),
succeedOperand, CVT_ND_OPD(root), op);
        buffer.push_back(addInst);
    }
}
```

当然，这里的函数实现还不是最终结果，因为我们在最后还必须处理常数优化问题（因为有测试点）。

常数优化

我们可以看到，其实很多数都可以在编译期算出来，例：

```
const int N = 9;
int a = N + N * N / 3;
```

我们很容易可以计算出a的右部表达式结果为`36`.

> **Task:** 对`analysisAddExp(AddExp* root, <ir::Instruction*>& buffer)`函数进行改写，使得其支持常数优化
>
> （思考）：对于const int/float声明的变量，在哪里存储他们的字面量？可以考虑符号表。

阅读到这里，其实你已经可以从`Exp`节点开始，自顶向下地完成绝大多数的内容了。强烈建议先上手试一下，再多加体会。

> **Tips1**:在编写的时候，一定要留意**局部变量**的管理，以及**类型转换**，这两个任务贯穿整个实验二！
>
> **Tips2**:在编写其他节点的时候，留意框架里节点的所有附属属性，他们随时都可能有用！
>
> **Tips3**:框架并不是不能改变的，你最终只需要实现`get_it_program`接口即可，如果想维护某些属性又不知道怎么弄的话，你可以修改框架内容以使符合自己的思路。
>
> **Tips4**:生成IR的时候，首先考虑这个IR是否有必要生成，即，不要生成过多无用的IR，这可能会给后续实验的处理带来麻烦。

关于定义-框架里的东西怎么用?

符号表与重命名问题

我们首先考虑以下程序`01_var_defn2.sy`：

```
//test domain of global var define and local define
int a = 3;
int b = 5;

int main(){
    int a = 5;
    return a + b;
}
```

可以看到，在全局变量和`main()`函数内有两个相同名称的变量`a`，SysY语言的文法也规定，当前作用域的变量优先，也就是说，最后return的结果是10而不是8.

如果我们考虑将以上程序翻译成IR语言，显然会有一个问题：如何管理变量a?

为了说明一个问题，我们考虑一个错误的IR生成：

```
void global():
    0:  def a, 3;
    1:  def b, 5;

int main():
    0:  def a, 5;
    1:  add $adda, a, b;
    2:  return $adda;

GVT:
    a int 0;
    b int 0;
```

这个程序乍一看没什么问题，能返回正确的值，但值得注意的是，全局变量a的值被改变成了5，而正常程序的执行来说，这是不应该发生的。执行到return，全局变量a仍应该为3.

因此，我们考虑要对所有变量重命名，使得不同作用域的变量，**在源程序中同名的，在IR程序中不同名。**

接下来，引入框架的符号表。符号表内有两个结构：`vector<scopeInfo>`与`map<std::string, ir::Function*>`：

```
struct SymbolTable{
    vector<ScopeInfo> scope_stack;      // idents record, sort by scope
    map<std::string,ir::Function*> functions;   // record functions
    // 一些成员变量被省略了。
}
```

其中`ScopeInfo`结构如下：

```
struct ScopeInfo {
     int cnt;    // 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
     string name;    // 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
     map_str_ste table;      //string 是操作数的【原始】名称，在 STE 中存放的应该是【变量重命名】后的名称
};
```

而`map_str_ste`实际是`map<string, STE>`，而`STE`又是如下结构：

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

这一层套娃套完，基本都头晕了。其实很简单：对于ScopeInfo来说，其维护了**一个作用域**下面的所有变量名。为了简化程序编写，对于table，我们在key值记录源程序的变量名，在STE的Operand的name里面记录重命名后的变量或字面量。对于符号表的单个表项（即STE），实际上就维护了一些信息：重命名后的变量名和类型，以及对于数组的维度信息。

对于单个作用域，我们维护一个原始变量名到某个符号表表项STE的映射，而对于多个作用域是如何管理的呢？为什么要进行栈式管理呢？

考虑这样一种情况：

```
{
    int a = 3;
    putint(a);
    {
        int a = 5;
        putint(a);
    }
    {
        a = 9;
        putint(a);
    }
    {
        putint(a);
    }
}
```

期望的结果是3， 5，9，9。但落实到IR实现上，我们如何精准地找到正确的引用呢？

注意这个程序实际是1个大的Scope里面套3个Scope，每个Scope里面有相应的操作，因此很容易想到利用栈式结构来管理（这也就解释了为什么ScopeInfo的key值要存原始值，STE存重命名后的值了）。想象一个栈scope_stack，元素为ScopeInfo，首先压入外界的ScopeInfo，a被存储进来

- 第一个Scope，我们从当前作用域倒着搜变量名a，那么就可以搜到当前的作用域a，并进行修改，执行完后，当前的Scope被弹出。
- 第二个Scope对a进行引用，查看当前作用域，没有找到a，那么从栈顶向栈底的顺序搜索a，结果在最大的Scope里面找到a，修改这个a，并推出。
- 第三个继续引用a，在最大的Scope里面找到这个a，然后这个a被修改过。

这个过程，如果翻译成IR，我们期望有：

```
0:  def a_1, 5;
1:  $ret, call putint(a_1);
2:  def a_2, 5;
3:  $ret, call putint(a_2);
4:  def a_1, 9;
5:  $ret, call putint(a_1);
6:  $ret, call putint(a_1);
```

需要特别强调的是，重命名的后缀**依据出现过多少个Scope来定，而不是当前栈里有多少个Scope来定。**（为什么？）

关于ir::Program和ir::Function

看指导书即可。

关于控制流-难点在哪?

关于Stmt...Goto如何处理?

基本翻译策略

（待补）

重定位

（待补）

短路运算

Tips

关于Cpp

这个实验涉及到大量的指针操作。没有经验的同学很容易写出下列代码：

```
void func(AstNode* root, <ir::instruction*>& buffer){
    ir::Instruction Inst;
    buffer.push_back(&Inst);
}
```

为了说明这个代码的问题在哪，请阅读并运行下面的cpp程序，并考虑程序运行时栈的变化，思考输出的产生原因。

```
#include <iostream>
using namespace std;
int *a = nullptr;
void func(){
  int b = 922;
  a = &b;
}
void func2(){
  int c = 918;
  // 这里需要注意的是，func2并没有对指针的操作
}
int main(){
  a = new int(9);
  cout << (*a) << endl;
  func();
  cout << (*a) << endl;
  func2();
```

```
  cout << (*a) << endl;
}
```

因此，正确的写法如下：

```
void func(AstNode* root, <ir::instruction*>& buffer){
    ir::Instruction* Inst = new Instruction();
    buffer.push_back(Inst);
}
```

关于测评机Bug

群里有一堆大佬发现了一些测评机的Bug：

```
if (cxt_stack.size() == 1) {        // in main function return
    cxt_stack.pop();
}
else {
    cxt_stack.pop();
    cur_ctx = cxt_stack.top();
}
```

在`src/tools/ir_executor.cpp`下找到这处程序，其中else分支的两行代码应该交换，否则可能会导致某些测试点不能通过（主要是后面比较难的测试点）。

关于易错点

（待补）

附录

（待补）