llvm IR的生成例子解析

本例子来源于例子

```
/* Compile the AST into a module */
void CodeGenContext::generateCode(NBlock &root)
```

这个函数是主要的IR生成函数，IR的生成从这里开始。

```
vector<const Type *> argTypes;
// static FunctionType *get(Type *Result, ArrayRef<Type*> Params, bool isVarArg);
FunctionType *ftype = FunctionType::get(Type::getVoidTy(getGlobalContext()),
argTypes, false);
```

上面一段在做关于函数类型的生成，argTypes是函数参数的类型由于main函数的参数为空所以这里argTypes的vector为空。然后创建一个函数类型,也就是main函数的函数类型。void ();

```
mainFunction = Function::Create(ftype, GlobalValue::InternalLinkage, "main",
module);
```

```
static Function *Create(FunctionType *Ty, LinkageTypes Linkage, const Twine &N = "",
Module *M = nullptr)
```

该函数调用了：

```
Function::Function(FunctionType *Ty, LinkageTypesLinkage, unsigned AddrSpace,const Twine
&name, Module*ParentModule)
```

大概意思就是创建这条IR语句“define internal void @main()”

```
BasicBlock *bblock = BasicBlock::Create(getGlobalContext(), "entry", mainFunction,
0);
```

这段代码的意思是创建一个新的块，名字为entry，并且将该块插入到mainFunctuin的IR中。(也就是mainFunctionIR的末端)。现在生成的IR就为：

define internal void @main() { entry: }

```
pushBlock(bblock);
```

这段代码的意思是将当前块插入到上下文的`std::stack<CodeGenBlock *> blocks;`栈中，因为每次都是访问blocks的栈顶元素，因此现在每次生成的IR都是属于栈顶块中的IR。

```
root.codeGen(*this); /* emit bytecode for the toplevel block */
```

这一段代码是IR生成的核心。这里的codegen会调用

```
Value* NBlock::codeGen(CodeGenContext& context)
{
    StatementList::const_iterator it;
    Value *last = NULL;
    for (it = statements.begin(); it != statements.end(); it++) {
        std::cout << "Generating code for " << typeid(**it).name() << std::endl;
        last = (**it).codeGen(context);
    }
    std::cout << "Creating block" << std::endl;
    return last;
}
```

相当于为所有的statement语句调用codegen函数，首先调FunctionDeclaration，也就是一下代码：

```
Value* NFunctionDeclaration::codeGen(CodeGenContext& context)
{
    vector<const Type*> argTypes;
    VariableList::const_iterator it;
    for (it = arguments.begin(); it != arguments.end(); it++) {
        argTypes.push_back(typeOf((**it).type));
    }
    FunctionType *ftype = FunctionType::get(typeOf(type), argTypes, false);
    Function *function = Function::Create(ftype, GlobalValue::InternalLinkage,
id.name.c_str(), context.module);
    BasicBlock *bblock = BasicBlock::Create(getGlobalContext(), "entry", function,
0);

    context.pushBlock(bblock);

    for (it = arguments.begin(); it != arguments.end(); it++) {
        (**it).codeGen(context);
    }

    block.codeGen(context);
    ReturnInst::Create(getGlobalContext(), bblock);

    context.popBlock();
    std::cout << "Creating function: " << id.name << std::endl;
    return function;
}
```

上面的代码就是一整个do_math函数IR生成的代码。仔细分析下上面代码：

```
    vector<const Type*> argTypes;
    VariableList::const_iterator it;
    for (it = arguments.begin(); it != arguments.end(); it++) {
        argTypes.push_back(typeOf((**it).type));
    }
    FunctionType *ftype = FunctionType::get(typeOf(type), argTypes, false);
```

上面的代码就是生成do_match函数类型`int (int)`

```
    Function *function = Function::Create(ftype, GlobalValue::InternalLinkage,
id.name.c_str(), context.module);
    BasicBlock *bblock = BasicBlock::Create(getGlobalContext(), "entry", function,
0);
```

创建`define internal i64 @do_math(i64)`IR，并且将创建的entry块插入到do_math IR的末尾。

```
context.pushBlock(bblock);
```

将当前栈顶的块替换为do_match函数的IR块，也就是现在生成的IR都会生成在do_math的entry中。

```
for (it = arguments.begin(); it != arguments.end(); it++) {
        (**it).codeGen(context);
    }
```

这一段是生成do_math函数的参数，也就是对应这句IR`%a = alloca i64 ; [#uses=1]`

```
block.codeGen(context);
```

这一句使生成do_math整个函数体的IR。要理解这一部分，可以看看NFunctionDeclaration类：

```
class NFunctionDeclaration : public NStatement {
public:
    const NIdentifier& type;
    const NIdentifier& id;
    VariableList arguments;
    NBlock& block;
    NFunctionDeclaration(const NIdentifier& type, const NIdentifier& id,
            const VariableList& arguments, NBlock& block) :
        type(type), id(id), arguments(arguments), block(block) { }
```

```
    virtual llvm::Value* codeGen(CodeGenContext& context);
};
```

NFunctionDeclaration类中将函数体存在一个NBlock& block;,现在就又可开心和递归的解析statement和expression，然后生成相应的IR了。

```
ReturnInst::Create(getGlobalContext(), bblock);
context.popBlock();
```

生成ret语句和当前栈顶块出栈，切换当前IR生成到哪一个块中。所有的IR指令会存储在Instruction.def文件中。

现在让我们回到CodeGenContext::generateCode()函数当中，继续往下分析。

```
//创建ret指令
ReturnInst::Create(getGlobalContext(), bblock);
//从栈中出块
popBlock();
```

```
/* Print the bytecode in a human-readable format
       to see if our program compiled properly
     */
PassManager pm;
pm.add(createPrintModulePass(&outs()));
pm.run(*module);
```

打印出人为可读的IR语句。

本例子的来源于LLVM官网教程

上面例子在生成AST的文法总的分析如下：
top ::= definition | extern | expression | ;
definition ::= 'def' prototype expression
prototype ::= id '(' id* ')'
expression ::= primary binopRHS
binopRHS ::= ('+' primary)*
primary ::= identifierexpr | numberexpr | parenexpr
numberexpr ::= number
parenexpr ::= '(' expression ')'
identifierexpr ::= identifier | identifier '(' expression* ')'
extern ::= 'extern' prototype

toplevelexpr ::= expression //如果单纯的只是一个expression，那么就会生成一个匿名函数

definition example: def do_math(a, b)
prototype example: do_math(a, b)
identifierexpr example: do_math(10, 20)
toplevelexpr example: 5+10 //单单就只从顶层开始解析‘5+10’expression，这里会解析成一个匿名函数。

现在开始分析下上面教程3的AST树生成和codegenIR的生成

输入的函数声明为: `def foo(a b c d e f g) a+b+(c+d)*e*f+g` 注意关于运算符的优先级采用Operator-Precedence Parsing的方法来递归解析表达式。对于上面的算数表达式首先解析主表达式a，然后他将解析为`[+,b], [+, (c+d)], [*, e] [*, f], [+, g]`。注意因为(c+d)是主表达式，因此在解析过程中不用担心子表达式嵌套的问题。

在开始分析代码之前我们先解释一些要用到的全局变量：

```
//在IR代码生成期间，TheContext是一个不透明的对象，拥有LLVM许多核心的数据结构，例如类型和常量表。
static LLVMContext TheContext;
// Builder对象是一个能够轻松生成LLVM IR指令的辅助对象， IRBuilder对象保持记录当前插入IR语句的地方和有一个新的途径去创造新的指令。
static IRBuilder<> Builder(TheContext);
// TheModule是一个包含函数和全局变量的LLVM对象，在很多方面它是包含IR指令的最上层的结构，它将拥有产生的IR语句的内存，这就是为什么我们返回原生的Value*，而不是unique_ptr<Value>。后者会导致TheModule内存的所有权转移。
static std::unique_ptr<Module> TheModule;
//这个表是由我们自定义的符号表，我们在这个设计的语言中用来保存函数参数的名字和对应IR代码的值。
static std::map<std::string, Value *> NamedValues;
```

然后开始分析代码：

```
static std::map<char, int> BinopPrecedence;

// Install standard binary operators.
// 1 is lowest precedence.
BinopPrecedence['<'] = 10;
BinopPrecedence['+'] = 20;
BinopPrecedence['-'] = 20;
BinopPrecedence['*'] = 40; // highest.
```

首先加载运算符优先级表

```
getNextToken(); //获取def
Main() -> MainLoop(); -> tok_def ->HandleDefinition() -> ParseDefinition() ->
ParsePrototype() -> ParseExpression() -> ParsePrimary() -> ParseNumberExpr() ->
ParseBinOpRHS() ->递归调用ParseBinOpRHS()直到expr被解析完
```

ParsePrototype() 解析foo(a b c d e f g) 形成PrototypeAST节点 ParseExpression() 解析a + b +（c + d）* e * f + g, 形成BinaryExprAST节点。上面代码最精彩的部分在ParseBinOpRHS()，这个函数解决了运算符优先级的问题，让我们来仔细分析下这个函数：

```
/// binoprhs
///   ::= ('+' primary)*
static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
                                              std::unique_ptr<ExprAST> LHS) {
  // If this is a binop, find its precedence.
  while (true) {
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;

    // Okay, we know this is a binop.
    int BinOp = CurTok;
    getNextToken(); // eat binop

    // Parse the primary expression after the binary operator.
    auto RHS = ParsePrimary();
    if (!RHS)
      return nullptr;

    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec + 1, std::move(RHS));
      if (!RHS)
        return nullptr;
    }

    // Merge LHS/RHS.
    LHS =
        llvm::make_unique<BinaryExprAST>(BinOp, std::move(LHS), std::move(RHS));
  }
}
```

`GetTokPrecedence()`函数得到当前运算符'+'的优先级为10

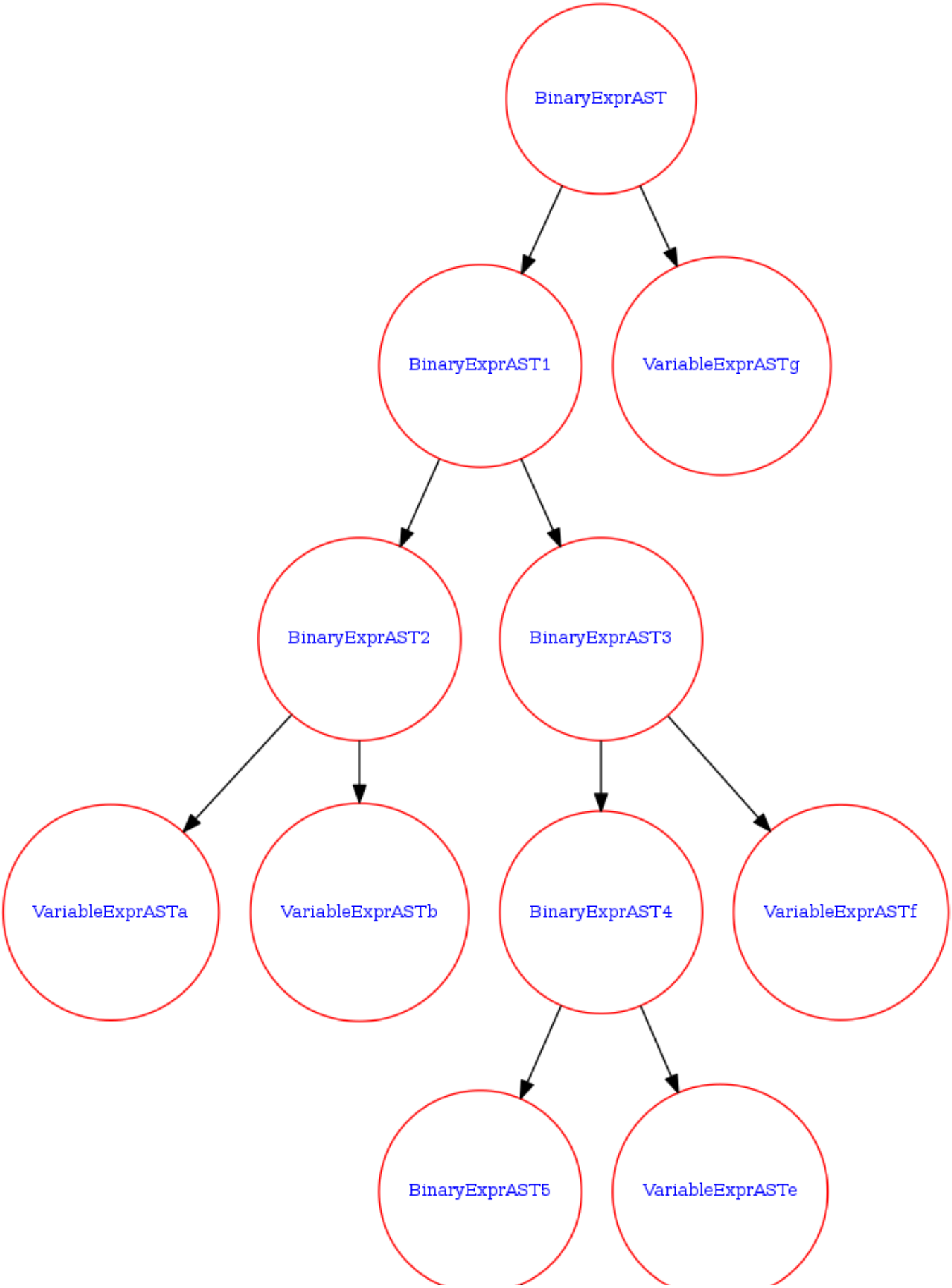
`if (TokPrec < ExprPrec) return LHS;` TokPrec 为当前运算符的优先级，ExprPrec为前面表达式的优先级的最大值。当前前者为10，后者为0，继续往下执行。这里的意思是ExprPrec表示ParseBinOpRHS函数能够处理最小运算符优先级的值。

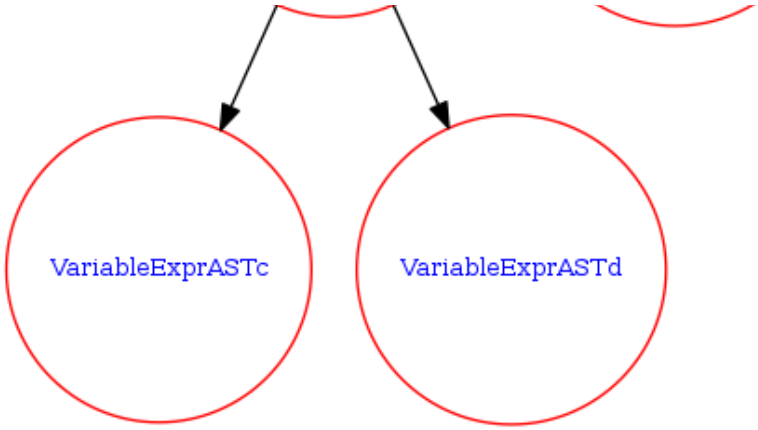
首先是LHS:a+b RHS:调用`RHS = ParseBinOpRHS(TokPrec+1, std::move(RHS))`合并右表达式，结果为:(c+d)*e。

然后下一个运算符为'*'，大于当前在b后面的'+'，因此调用`RHS = ParseBinOpRHS(TokPrec+1, std::move(RHS))`合并右表达式，结果为`RHS:(c+d)*e*f`。当前LHS:`a+b` RHS:`(c+d)*e*f`

然后下一个运算符为'+'，合并左右表达式，当前LHS：`a+b+(c+d)*e*f`。RHS：`g`。

总的构造的节点如图所示:





然后就开始递归的codegen产生IR语句

```
auto FnAST = ParseDefinition();
auto *FnIR = FnAST->codegen() -> Function *FunctionAST::codegen() -> Body-
>codegen() -> BinaryExprAST->codegen() -> 递归调用 BinaryExprAST->codegen() 生成该
节点的IR语句
```

下面我们按照上面的思路一步一步分析： 首先看看`Function *FunctionAST::codegen()`函数

```
  // First, check for an existing function from a previous 'extern' declaration.
  Function *TheFunction = TheModule->getFunction(Proto->getName());

  if (!TheFunction)
    TheFunction = Proto->codegen();

  if (!TheFunction)
    return nullptr;
```

上面代码是查看当前函数是否在之前使用'extern'关键字声明，也就是在指定模块查找特定函数，如果不存在就返回null,然后自己使用函数原型来自己生成函数声明的IR语句。

```
// Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
  Builder.SetInsertPoint(BB);
```

创建entry块，并且现在所有创建的IR都要创建到这一块中。

```
  // Record the function arguments in the NamedValues map.
  NamedValues.clear();
  for (auto &Arg : TheFunction->args())
    NamedValues[Arg.getName()] = &Arg;
```

这里记录函数参数的名字，便于后面VariableAST->codegen()的时候产生的IR变量名与函数参数的名字相同，便于阅读。

```
  if (Value *RetVal = Body->codegen()) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    return TheFunction;
  }
```

生成函数体的IR。根据上面生成的AST树表明，Body实质上是一个BinaryExprAST节点，因此调用Value *BinaryExprAST::codegen()，然后递归的解析LHS和RHS，因为左右表达式都是BinaryExprAST节点。并且获取左右节点的返回值，用一个变量存储。

然后调用`return Builder.CreateFAdd(L, R, "addtmp");`返回最后的返回值变量，该变量用于最后调用`Builder.CreateRet(RetVal);`生成ret指令。

`verifyFunction(*TheFunction);`用于验证生成的代码，检查生成代码的一致性。

`TheFunction->eraseFromParent();`如果函数体body生成失败，会从该模型中将函数声明删除掉。

```
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;

  switch (Op) {
  case '+':
    return Builder.CreateFAdd(L, R, "addtmp");
  case '-':
    return Builder.CreateFSub(L, R, "subtmp");
  case '*':
    return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(TheContext), "booltmp");
  default:
    return LogErrorV("invalid binary operator");
  }
}
```

至此整个函数定义的代码IR的生成就完成了。最后贴一下产生的IR的形式。

```
define double @foo(double %a, double %b, double %c, double %d, double %e, double
%f, double %g) {
entry:
  %addtmp = fadd double %a, %b
  %addtmp1 = fadd double %c, %d
  %multmp = fmul double %addtmp1, %e
  %multmp2 = fmul double %multmp, %f
  %addtmp3 = fadd double %addtmp, %multmp2
  %addtmp4 = fadd double %addtmp3, %g
  ret double %addtmp4
}
```