使用LLVM实现一门语言（三）Code Generation to LLVM IR

终于开始codegen了，首先我们include一些LLVM头文件，定义一些全局变量

```
#include "llvm/ADT/APFloat.h"
#include "llvm/ADT/STLExtras.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Type.h"
#include "llvm/IR/Verifier.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Transforms/InstCombine/InstCombine.h"
#include "llvm/Transforms/Scalar.h"
#include "llvm/Transforms/Scalar/GVN.h"

// 记录了LLVM的核心数据结构，比如类型和常量表，不过我们不太需要关心它的内部
llvm::LLVMContext g_llvm_context;
// 用于创建LLVM指令
llvm::IRBuilder<> g_ir_builder(g_llvm_context);
// 用于管理函数和全局变量，可以粗浅地理解为类c++的编译单元(单个cpp文件)
llvm::Module g_module("my cool jit", g_llvm_context);
// 用于记录函数的变量参数
std::map<std::string, llvm::Value*> g_named_values;
```

然后给每个AST Class增加一个CodeGen接口

```
// 所有 `表达式` 节点的基类
class ExprAST {
 public:
  virtual ~ExprAST() {}
  virtual llvm::Value* CodeGen() = 0;
};

// 字面值表达式
class NumberExprAST : public ExprAST {
 public:
  NumberExprAST(double val) : val_(val) {}
```

```
  llvm::Value* CodeGen() override;

 private:
  double val_;
};
```

首先实现NumberExprAST的CodeGen

```
llvm::Value* NumberExprAST::CodeGen() {
  return llvm::ConstantFP::get(g_llvm_context, llvm::APFloat(val_));
}
```

由于Kaleidoscope只有一种数据类型FP64，所以直接调用ConstantFP传入即可，APFloat是llvm内部的数据结构，用于存储Arbitrary Precision Float. 在LLVM IR中，所有常量是唯一且共享的，所以这里使用的get而不是new/create.

然后实现VariableExprAST的CodeGen

```
llvm::Value* VariableExprAST::CodeGen() {
  return g_named_values.at(name_);
}
```

由于Kaleidoscope的VariableExpr只存在于函数内对函数参数的引用，我们假定函数参数已经被注册到g_name_values中，所以VariableExpr直接查表返回即可。

接着实现BinaryExprAST，分别codegen lhs，rhs然后创建指令处理lhs，rhs即可

```
llvm::Value* BinaryExprAST::CodeGen() {
  llvm::Value* lhs = lhs_->CodeGen();
  llvm::Value* rhs = rhs_->CodeGen();
  switch (op_) {
    case '<': {
      llvm::Value* tmp = g_ir_builder.CreateFCmpULT(lhs, rhs, "cmptmp");
      // 把 0/1 转为 0.0/1.0
      return g_ir_builder.CreateUIToFP(
          tmp, llvm::Type::getDoubleTy(g_llvm_context), "booltmp");
    }
    case '+': return g_ir_builder.CreateFAdd(lhs, rhs, "addtmp");
    case '-': return g_ir_builder.CreateFSub(lhs, rhs, "subtmp");
    case '*': return g_ir_builder.CreateFMul(lhs, rhs, "multmp");
    default: return nullptr;
  }
}
```

实现CallExprAST

```
llvm::Value* CallExprAST::CodeGen() {
  // g_module中存储了全局变量/函数等
  llvm::Function* callee = g_module.getFunction(callee_);

  std::vector<llvm::Value*> args;
  for (std::unique_ptr<ExprAST>& arg_expr : args_) {
    args.push_back(arg_expr->CodeGen());
  }
  return g_ir_builder.CreateCall(callee, args, "calltmp");
}
```

实现ProtoTypeAST

```
llvm::Value* PrototypeAST::CodeGen() {
  // 创建kaleidoscope的函数类型 double (doube, double, ..., double)
  std::vector<llvm::Type*> doubles(args_.size(),
                                   llvm::Type::getDoubleTy(g_llvm_context));
  // 函数类型是唯一的，所以使用get而不是new/create
  llvm::FunctionType* function_type = llvm::FunctionType::get(
      llvm::Type::getDoubleTy(g_llvm_context), doubles, false);
  // 创建函数，ExternalLinkage意味着函数可能不在当前module中定义，在当前module
  // 即g_module中注册名字为name_，后面可以使用这个名字在g_module中查询
  llvm::Function* func = llvm::Function::Create(
      function_type, llvm::Function::ExternalLinkage, name_, &g_module);
  // 增加IR可读性，设置function的argument name
  int index = 0;
  for (auto& arg : func->args()) {
    arg.setName(args_[index++]);
  }
  return func;
}
```

实现FunctionAST

```
llvm::Value* FunctionAST::CodeGen() {
  // 检查函数声明是否已完成codegen(比如之前的extern声明)，如果没有则执行codegen
  llvm::Function* func = g_module.getFunction(proto_->name());
  if (func == nullptr) {
    func = proto_->CodeGen();
  }
  // 创建一个Block并且设置为指令插入位置。
  // llvm block用于定义control flow graph，由于我们暂不实现control flow，创建
  // 一个单独的block即可
  llvm::BasicBlock* block =
      llvm::BasicBlock::Create(g_llvm_context, "entry", func);
  g_ir_builder.SetInsertPoint(block);
  // 将函数参数注册到g_named_values中，让VariableExprAST可以codegen
  g_named_values.clear();
  for (llvm::Value& arg : func->args()) {
    g_named_values[arg.getName()] = &arg;
```

```
  }
  // codegen body然后return
  llvm::Value* ret_val = body_->CodeGen();
  g_ir_builder.CreateRet(ret_val);
  llvm::verifyFunction(*func);
  return func;
}
```

至此，所有codegen都已完成，修改main

```
int main() {
  GetNextToken();
  while (true) {
    switch (g_current_token) {
      case TOKEN_EOF: return 0;
      case TOKEN_DEF: {
        auto ast = ParseDefinition();
        std::cout << "parsed a function definition" << std::endl;
        ast->CodeGen()->print(llvm::errs());
        std::cerr << std::endl;
        break;
      }
      case TOKEN_EXTERN: {
        auto ast = ParseExtern();
        std::cout << "parsed a extern" << std::endl;
        ast->CodeGen()->print(llvm::errs());
        std::cerr << std::endl;
        break;
      }
      default: {
        auto ast = ParseTopLevelExpr();
        std::cout << "parsed a top level expr" << std::endl;
        ast->CodeGen()->print(llvm::errs());
        std::cerr << std::endl;
        break;
      }
    }
  }
  return 0;
}
```

输入测试

```
4 + 5

def foo(a b)
    a*a + 2*a*b + b*b

foo(2, 3)

def bar(a)
```

```
    foo(a, 4) + bar(31337)

extern cos(x)

cos(1.234)
```

得到输出

```
parsed a top level expr
define double @0() {
entry:
  ret double 9.000000e+00
}

parsed a function definition
define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a
  %multmp1 = fmul double 2.000000e+00, %a
  %multmp2 = fmul double %multmp1, %b
  %addtmp = fadd double %multmp, %multmp2
  %multmp3 = fmul double %b, %b
  %addtmp4 = fadd double %addtmp, %multmp3
  ret double %addtmp4
}

parsed a top level expr
define double @1() {
entry:
  %calltmp = call double @foo(double 2.000000e+00, double 3.000000e+00)
  ret double %calltmp
}

parsed a function definition
define double @bar(double %a) {
entry:
  %calltmp = call double @foo(double %a, double 4.000000e+00)
  %calltmp1 = call double @bar(double 3.133700e+04)
  %addtmp = fadd double %calltmp, %calltmp1
  ret double %addtmp
}

parsed a extern
declare double @cos(double)

parsed a top level expr
define double @2() {
entry:
  %calltmp = call double @cos(double 1.234000e+00)
```

```
  ret double %calltmp
}
```

至此，我们已成功将Parser输出的AST转为LLVM IR