Kaleidoscope

代码解释(2)

万花筒语言 - LLVM 新手入门教程

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html

PLCT - SSC

环境准备

- Linux 环境 (Ubuntu)
- 下载LLVM最新源码
  - git clone https://github.com/llvm/llvm-project.git
- Cmake生成Makefile
  - mkdir build && cd build
  - cmake -G “Unix Makefiles” ../llvm
- Make编译
  - make Kaleidoscope
- 运行Kaleidoscope
  - bin/Kaleidoscope-Ch3

环境准备2

- Windows
- VS 社区版
  - 安装对应的 桌面版C++
- Cmake
  - 生成 VS的项目管理
- VS
  - 启动项目管理文件sln
  - 选定Kaleidoscope-ch3为启动项目

预处理、命名空间

```
#include "llvm/ADT/APFloat.h"      // llvm 头文件，须在llvm环境下编译
#include "llvm/ADT/STLExtras.h"    // 或者使用
// clang++ -g -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -o toy
#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/Module.h"
#include "llvm/IR/Type.h"
#include "llvm/IR/Verifier.h"
```

```
using namespace llvm; // llvm的命名空间,省略llvm::
```

全局变量

```
static LLVMContext TheContext;
// LLVMContext针对每一个线程记录了线程本地的变量，即对于每一个LLVM的线程，
// 都对应了这样一个context的实例。
static IRBuilder<> Builder(TheContext);
// Builder是用于简化LLVM指令生成的辅助对象。IRBuilder类模板的实例
// 可用于跟踪当前插入指令的位置，同时还带有用于生成新指令的方法。
static std::unique_ptr<Module> TheModule;
// 其中TheModule是LLVM中用于存放代码段中所有函数和全局变量的结构。
// 从某种意义上讲，可以把它当作LLVM IR代码的顶层容器。
static std::map<std::string, Value *> NamedValues;
// NamedValues映射表用于记录代码的符号表。在这一版的Kaleidoscope中，
// 可引用的变量只有函数的参数。因此，在生成函数体的代码时，函数的参数就存放在这张表中。
```

异常处理

```
std::unique_ptr<ExprAST> LogError(const char *Str) {
  fprintf(stderr, "Error: %s\n", Str);
  return nullptr;
}
```

```
std::unique_ptr<PrototypeAST> LogErrorP(const char *Str) {
  LogError(Str);
  return nullptr;
}
```

```
Value *LogErrorV(const char *Str) {
  LogError(Str);
  return nullptr;
}
```

AST、codegen

```
class ExprAST {
public:
  virtual ~ExprAST() = default;

  virtual Value *codegen() = 0; // 代码生成对应代码，llvm::Value类型
                                // 下有详细定义
};
```

AST、codegen

```
class NumberExprAST : public ExprAST {
  double Val;

public:
  NumberExprAST(double Val) : Val(Val) {}

  Value *codegen() override;
};
```

```
Value *NumberExprAST::codegen() {
  return ConstantFP::get(TheContext, APFloat(Val));
  // LLVM IR中的数值常量是由ConstantFP类表示的。
  // 在其内部，具体数值由APFloat（Arbitrary Precision Float，可用于存储任意精度的浮点数常量）表示。
  // 在LLVM IR内部，常量都只有一份，并且是共享的。
}
```

```
ConstantFP* ConstantFP::get(LLVMContext &Context, const APFloat& V) {
  LLVMContextImpl* pImpl = Context.pImpl;

  std::unique_ptr<ConstantFP> &Slot = pImpl->FPConstants[V];

  if (!Slot) {
    Type *Ty;
    if (&V.getSemantics() == &APFloat::IEEEhalf())
      Ty = Type::getHalfTy(Context);
    else if (&V.getSemantics() == &APFloat::BFloat())
      Ty = Type::getBFloatTy(Context);
    else if (&V.getSemantics() == &APFloat::IEEEsingle())
      Ty = Type::getFloatTy(Context);
    else if (&V.getSemantics() == &APFloat::IEEEdouble())
      Ty = Type::getDoubleTy(Context);
    else if (&V.getSemantics() == &APFloat::x87DoubleExtended())
      Ty = Type::getX86_FP80Ty(Context);
    else if (&V.getSemantics() == &APFloat::IEEEquad())
      Ty = Type::getFP128Ty(Context);
    else {
      assert(&V.getSemantics() == &APFloat::PPCDoubleDouble() &&
             "Unknown FP format");
      Ty = Type::getPPC_FP128Ty(Context);
    }
    Slot.reset(new ConstantFP(Ty, V));
  }

  return Slot.get();
}
```

AST、codegen

```
class VariableExprAST : public ExprAST {
  std::string Name;

public:
  VariableExprAST(const std::string &Name) : Name(Name) {}

  Value *codegen() override;
};
```

```
Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  // 表达式被转为匿名函数，所以变量都是函数的参数
  if (!V)
    return LogErrorV("Unknown variable name");
  return V;
}
```

AST、codegen

```
class BinaryExprAST : public ExprAST {
  char Op;
  std::unique_ptr<ExprAST> LHS, RHS;

public:
  BinaryExprAST(char Op, std::unique_ptr<ExprAST> LHS,
                std::unique_ptr<ExprAST> RHS)
      : Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}

  Value *codegen() override;
};
```

```
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");
```

```cpp
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");
    // 重名时会追加一个自增的唯一数字后缀
    // 类型相同所以代码得以简化
  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");
  }
}
```

AST、codegen

```
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<std::unique_ptr<ExprAST>> Args;

public:
  CallExprAST(const std::string &Callee,
              std::vector<std::unique_ptr<ExprAST>> Args)
      : Callee(Callee), Args(std::move(Args)) {}

  Value *codegen() override;
};
```

```
Value *CallExprAST::codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (!CalleeF)
    return LogErrorV("Unknown function referenced");

  // If argument mismatch error.
```

```
Value *CallExprAST::codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (!CalleeF)
    return LogErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return LogErrorV("Incorrect # arguments passed");

  std::vector<Value *> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->codegen());
    if (!ArgsV.back())
      return nullptr;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
```

AST、codegen

```
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;

public:
  PrototypeAST(const std::string &Name, std::vector<std::string> Args)
      : Name(Name), Args(std::move(Args)) {}

  Function *codegen();
  const std::string &getName() const { return Name; }
};
```

```
Function *PrototypeAST::codegen() {
  // n个double
  std::vector<Type *> Doubles(Args.size(), Type::getDoubleTy(TheContext));
  FunctionType *FT =
      FunctionType::get(Type::getDoubleTy(TheContext), Doubles, false);
      // 返回值类型,参数列表,参数个数不可变

  Function *F =
```

AST、codegen

```
Function *PrototypeAST::codegen() {
  // n个double
  std::vector<Type *> Doubles(Args.size(), Type::getDoubleTy(TheContext));
  FunctionType *FT =
      FunctionType::get(Type::getDoubleTy(TheContext), Doubles, false);
      // 返回值类型,参数列表,参数个数不可变

  Function *F =
      Function::Create(FT, Function::ExternalLinkage, Name, TheModule.get());
      // 函数信息，链接方式，函数名，注册在TheModule的符号表

  // Set names for all arguments.
  unsigned Idx = 0;
  for (auto &Arg : F->args())
    Arg.setName(Args[Idx++]);
  // 给函数F赋以对应的参数名称

  return F;
}
```

AST、codegen

```
class FunctionAST {
  std::unique_ptr<PrototypeAST> Proto;
  std::unique_ptr<ExprAST> Body;

public:
  FunctionAST(std::unique_ptr<PrototypeAST> Proto,
              std::unique_ptr<ExprAST> Body)
      : Proto(std::move(Proto)), Body(std::move(Body)) {}

  Function *codegen();
};
```

```
Function *FunctionAST::codegen() {
  // 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;
```

```
  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
  Builder.SetInsertPoint(BB); // 新指令插入到尾部

  // Record the function arguments in the NamedValues map.
  NamedValues.clear();
  for (auto &Arg : TheFunction->args())
    NamedValues[std::string(Arg.getName())] = &Arg;
  // 参数列表插入

  if (Value *RetVal = Body->codegen()) {
    // 进行函数的生成
    // Finish off the function.
    Builder.CreateRet(RetVal);

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

    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return nullptr;
}
```

顶层解析

```
static void HandleDefinition() {
  if (auto FnAST = ParseDefinition()) {
    if (auto *FnIR = FnAST->codegen()) {
      fprintf(stderr, "Read function definition:");
      FnIR->print(errs());
      fprintf(stderr, "\n");
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}
```

```
  if (auto ProtoAST = ParseExtern()) {
    if (auto *FnIR = ProtoAST->codegen()) {
      fprintf(stderr, "Read extern: ");
      FnIR->print(errs());
      fprintf(stderr, "\n");
    }
```

```
  if (auto FnAST = ParseTopLevelExpr()) {
    if (auto *FnIR = FnAST->codegen()) {
      fprintf(stderr, "Read top-level expression:");
      FnIR->print(errs());
      fprintf(stderr, "\n");
    }
```

主函数

```
  TheModule = std::make_unique<Module>("my cool jit", TheContext);

  // Print out all of the generated code.
  TheModule->print(errs(), nullptr);
```

def foo(a b) a+b;

```
ready> def foo(a b) a+b;
ready> Read function definition:define double
@foo(double %a, double %b) {
entry:
  %addtmp = fadd double %a, %b
  ret double %addtmp
}

ready>
```

def foo(a b) a+b;

1. main() 初始化
    1. MainLoop() 判断表达式或声明或定义
        1. HandleDefinition() 判断是否函数合规
            1. ParseDefinition() 调用处理def，函数原型，函数体
                1. ParsePrototype() 处理函数原型
                    1. ArgNames 存储相应的参数名称 a,b
                    2. return make_unique<PrototypeAST>(FnName, move(ArgNames));
                2. ParseExpression() 处理第一个函数体
                    1. ParsePrimary() 判断变量，数字，左括号
                        1. ParseIdentifierExpr() 解析变量或函数
                            1. return make_unique<VariableExprAST>(IdName);
                    2. ParseBinOpRHS(0, std::move(LHS)) 解析下一个 Op,RHS
                        1. RHS = ParsePrimary() 判断变量，数字，左括号
                        2. LHS = make_unique<BinaryExprAST>(BinOp, move(LHS), move(RHS)); 解析好的表达式转为LHS
                        3. return LHS;
                3. return make_unique<FunctionAST>(move(Proto), move(E)); 返回生成好的语法树
            2. FunctionAST::codegen() 函数生成
                1. Proto->getName() 获取函数名
                    1. TheModule->getFunction() 查找是否定义过
                2. Proto->codegen() 原型代码生成
                    1. FunctionType::get(Type::getDoubleTy(TheContext), Doubles, false); 返回值,参数,参数个数不变
                    2. Function::Create(FT, ExternalLinkage, Name, TheModule.get()); 原型,链接,名称,Module
                    3. Arg.setName() 参数名赋值
                    4. return F;

1. ParsePrimary() 判断变量，数字，左括号
    1. ParseIdentifierExpr() 解析变量或函数
        1. return make_unique<VariableExprAST>(IdName);
    2. ParseBinOpRHS(0, std::move(LHS)) 解析下一个 Op,RHS
        1. RHS = ParsePrimary() 判断变量，数字，左括号
        2. LHS = make_unique<BinaryExprAST>(BinOp, move(LHS), move(RHS)); 解析好的表达式转为LHS
        3. return LHS;
3. return make_unique<FunctionAST>(move(Proto), move(E)); 返回生成好的语法树
2. FunctionAST::codegen() 函数生成
    1. Proto->getName() 获取函数名
        1. TheModule->getFunction() 查找是否定义过
    2. Proto->codegen() 原型代码生成
        1. FunctionType::get(Type::getDoubleTy(TheContext), Doubles, false); 返回值,参数,参数个数不变
        2. Function::Create(FT, ExternalLinkage, Name, TheModule.get()); 原型,链接,名称,Module
        3. Arg.setName() 参数名赋值
        4. return F;
    3. BasicBlock::Create(TheContext, “entry”, TheFunction); 代码块存储
    4. NamedValues[string(Arg.getName())] = &Arg; 参数名列表
    5. Body->codegen() 函数体生成
        1. LHS->codegen(); a
        2. RHS->codegen(); b
        3. return Builder.CreateFAdd(L, R, "addtmp"); +
    6. Builder.CreateRet(RetVal); 返回值生成
    7. verifyFunction(*TheFunction); 函数生成校验
3. FnIR->print(errs()); 输出代码