Tutorial 6: LLVM Framework API in C++

LLVM IR stands for low-level virtual machine intermediate representation. LLVM is a Static Single Assignment (SSA) based representation in assembly language form. It provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. In assignment 5, use LLVM framework API to generate LLVM IR from Minic AST in `C++`.

Agenda

- Important LLVM IR Class description
- Typical LLVM Framework API
- Example for implementing in C++
- Example for control flow graph

Some good sources

- LLVM Tutorial: https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html
- LLVM Language Reference Manual: https://llvm.org/docs/LangRef.html
- llvm Namespace Reference: https://llvm.org/doxygen/namespacellvm.html

Name	Description
`llvm::Context`	Owns a lot of core LLVM data structures such as the type tables. You don't need to understand all of it, but it is one of the arguments for other API calls.
`llvm::Module`	Owns functions and global variables. It owns the contents of all of the IR.
`llvm::Builder`	A helper object of `llvm::Module` to generate LLVM instructions. It has methods to insert or create new instructions.
`llvm::Value`	A base class. Itself and Its inherited classes such as `llvm::Constant` can represent intliternal, expr, variable, function argument, etc.
`llvm::Type`	A base class of type. Itself and Its inherited classes such as `llvm::FunctionType` can represent function prototype, array type, etc.
`llvm::BasicBlock`	Basic block in control flow graph (CFG), which is mentioned in lecture. Instructions are inserted in the block. `llvm::Builder` manages CFG.

Table 1: Important LLVM Class Description

LLVM Framework API

Table 2 shows some typical API functions which are used in Assignment 5.

Return type & Methods under namespace llvm	**Description**
Type* Type::getInt32Ty(*TheContext) ArrayType* ArrayType::get(Type* a, uint64 b)	return 32-bit width "integer" type. return b type-a elements' ArrayType type.
FunctionType* FunctionType::get(Type* a, std::vector⟨Type*⟩b, **false**)	a is function return type, b is an array of params types, false means non-variadic function. The methods returns FunctionType
Function* Function::Create(FunctionType* a, **Function::ExternalLinkage**, std::string b, **TheModule.get()**);	a is function type, b is function name. It returns a function **and** sets the function into TheModule.
Constant* ConstantInt::get(Type* a, uint64 b, bool isSigned)	return boolean and integer constant. The constant type is a, value is b. It is signed if isSigned is true.
BasicBlock* BasicBlock::Create(***TheContext**, std::string a, Function* Parent)	Create a named "a" basic block which is a sequence of instructions. The block belongs to Function Parent.
void IRBuilder::SetInsertPoint(BasicBlock* bb) BasicBlock* IRBuilder::GetInsertBlock()	This specifies that created instructions should be appended to the end of bb. Get the inserted block bb.
Value* IRBuilder::CreateStore(Value* val, Value* ptr)	Store instruction to write val to ptr.
Value* IRBuilder::CreateLoad(Value* ptr)	Load instruction to load ptr.
CreateAdd CreateSub CreateMul ...	Add, Sub, Mul instructions ...
Value* IRBuilder::CreateBr(BasicBlock* Dest)	Unconditional 'br label X' instruction.
Value* IRBuilder::CreateCondBr(Value* Cond, BasicBlock* True, BasicBlock* False)	Conditional 'br Cond, TrueDest, FalseDest' instruction.
Value* IRBuilder::CreatePHI(Type* a, unsigned n)	Create a PHI node with n incoming edges.
void PHINode::addincoming(Value* a, BasicBlock* bb)	Add an incoming value a and its corresponding block bb.
Value* IRBuilder::CreateGEP(Value* a, std::vector⟨Value*⟩idxlst)	Create getelementptr instruction for variable a. **Do some research on it. Refer to A5 handout.**

Table 2: Typical LLVM Framework API Description

Examples

(a)

First example is a simplified version from online LLVM tutorial.
After generating AST,

```
using namespace llvm;
static std::unique_ptr<LLVMContext> TheContext;
static std::unique_ptr<Module> TheModule;
static std::unique_ptr<IRBuilder<>> Builder;
static std::map<std::string, Value *> NamedValues;

static void InitializeModule() {
  // Open a new context and module.
  TheContext = std::make_unique<LLVMContext>();
  TheModule = std::make_unique<Module>("output.bc", *TheContext);

  // Create a new builder for the module.
  Builder = std::make_unique<IRBuilder<>>(*TheContext);
}
//Classes for node: NumberExprAST, BinaryExprAST, CallExprAST ...
...
//return value representing numeric literals
```

```
Value *NumberExprAST::codegen() {
  return ConstantFP::get(TheContext, APFloat(Val));
}
//Look this variable up in the function.
Value *VariableExprAST::codegen() {
  return NamedValues[Name];
}
//Impl IR for binary expr
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;

  switch (Op) {
  case '+':
    return Builder.CreateFAdd(L, R);
  case '-':
    return Builder.CreateFSub(L, R);
  case '*':
    return Builder.CreateFMul(L, R);
  case '<':
    L = Builder.CreateFCmpULT(L, R);
    // Convert bool 0/1 to double 0.0 or 1.0
    // You don't need to know it.
    return Builder.CreateUIToFP(L, Type::getDoubleTy(TheContext));
  }
}
//Impl IR for call expr
Value *CallExprAST::codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);

  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);
}
//Generate function prototype
Function *PrototypeAST::codegen() {
  // Make the function type:  double(double,double, .....) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(TheContext));
  FunctionType *FT =
    FunctionType::get(Type::getDoubleTy(TheContext), Doubles, false);

  Function *F =
    Function::Create(FT, Function::ExternalLinkage, Name, TheModule.get());

    return F;
}
//Visit Func declaration
Function *FunctionAST::codegen() {
```

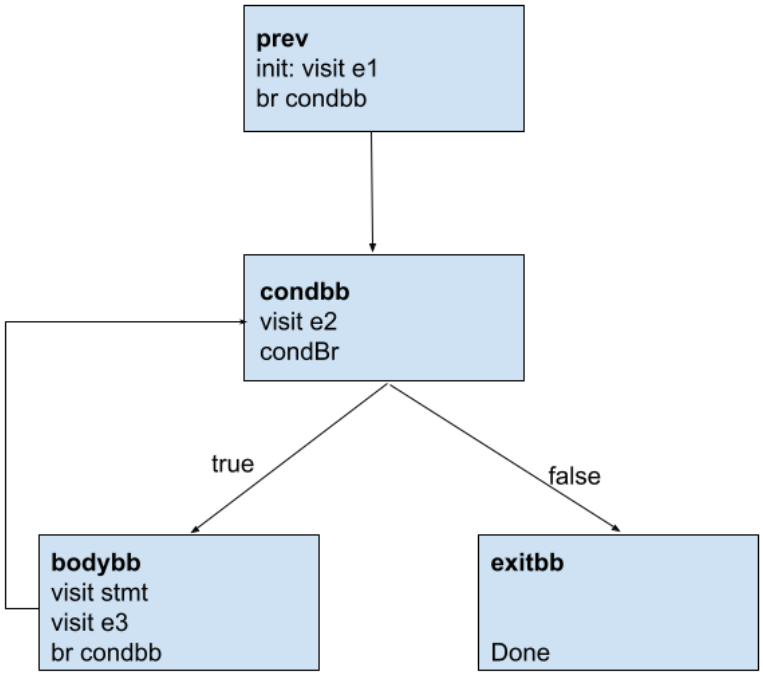
```
  // First, check for an existing function from a previous 'extern' declaration.
  Function *TheFunction = TheModule->getFunction(Proto->getName());
  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(*TheContext, "entry", TheFunction);
  Builder->SetInsertPoint(BB);
  // Record the function arguments in the NamedValues map.
  // Then allocate them in other functions
  NamedValues.clear();
  for (auto &Arg : TheFunction->args())
    NamedValues[std::string(Arg.getName())] = &Arg;

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

  return TheFunction;
}
```

Listing 1: Example function implementations

(b)

How to draw "For" statement CFG in general?

'for' '(' e1=expropt ';' e2=expropt ';' e3=expropt ')' stmt


prev
init: visit e1
br condbb
condbb
visit e2
condBr
true
false
bodybb
visit stmt
visit e3
br condbb
exitbb
Done