

Assignment 5: LLVM IR Generation (22%)

Due: Mar. 18, 2022

In this assignment, you will learn how to use LLVM framework API to generate LLVM IR in C++. The completed assignment is likely to be around 300~400 lines of code.

LLVM IR Generation Workflow in MiniC (cont'd from Assignment 4 Workflow)

Similar to the `VerifyAndBuildSymbols` class, after constructing an `IRGenerator` object, it visits each AST node to build its LLVM module. The `IRGenerator` class is a derived class from `ASTVisitor` in order to visit each node. Finally, in `main.cpp`, the LLVM module outputs IR Bitcode and writes to a file corresponding to LLVM IR. For this assignment, we will focus on implementing the `IRGenerator.cpp` file to do this.

Implementation Instructions

The [LLVM Tutorial](#) (Ch.3 onwards) will be similar to the assignment, and we **strongly recommend** you to read it. You will get a better idea on how to implement the MiniC LLVM IR generation. In addition, here are the links for [LLVM Language Reference Manual](#) and [LLVM Namespace Reference](#). The links list and explain LLVM data structures and functions. You can search API functions from there if necessary.

The following section shows the related MiniC data structures and the instructions for implementing IR generation for each visited node.

(a) Related Data Structures

Table 1 shows some important member variables. You can add more helper variables for IR generation.

Name	Description
<code>IRGenerator::TheContext</code>	A pointer that 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.
<code>IRGenerator::TheModule</code>	A pointer that owns functions and global variables. It owns the contents of all of the IR.
<code>IRGenerator::TheBuilder</code>	A pointer to a helper object to generate LLVM instructions. It has methods to insert or create new instructions.
<code>IRGenerator::ModuleName</code>	The name of the generated IR as known as IR Bitcode file name.
<code>VarSymbolEntry::LLVMValue</code>	In the variable symbol table, each variable string matches its <code>llvm::Value</code> . We will explain how to use it later.
	<code>Expr</code> and <code>Parameter</code> will also interact with <code>llvm::Value</code> objects. You are free to add other member variables.

Table 1: Member Variable Description

(b) LLVM Framework API

Table 2 and Table 3 shows some important APIs you should notice. Table 4 and Table 5 shows how to use the API to implement IRGenerator member functions. Here are the recommendations for each function, but the implementation is ultimately up to you.

Tips

- It may be useful to add a function to convert `minicc::Type` to `llvm::Type`.
- It may be useful to add a `std::map` to store the pair of `Expr*` and `llvm::Value*`. You will store and read it when visiting the nodes.
- Design carefully how to use LLVM IR to implement `For loop`, `If statement` and `PHI node`.
- It may be useful to use the member functions of the class inherited from `ASTNode`.
- "`TheBuilder->GetInsertBlock()->getTerminator()`" can be used to check whether the current block has return instruction or not.
- Get an idea how to use `createGEP` to index an array. Here is "[The Often Misunderstood GEP Instruction](#)" which may be useful for you.

Compilation and Testing

Please make sure you are using `clang 11.0`. After compiling your MiniC compiler, you can run:

```
# Under build/ directory
# Generate IR Bitcode
src/minicc <YourInput.c> -o out.bc
# Generate executable file
clang out.bc minicio/libminicio.a -o out
# Run the executable file
./out
# Generate IR assembly code (Important for debugging!)
llvm-dis out.bc -o out.ll
```

You can read `out.ll` in any text editor.

Public Autotester

We have provided a public autotester `a5tester.zip` on Quercus. After compiling your code, change `asst5.py` to your minicc executable file path. Then simply run:

```
./asst5.py
OR in verbose mode
./asst5.py -v
```

The tester has several MiniC source files. Your compiler will compile the files and the generated executable files will run without any problems in the tester.

Deliverables

In project folder, compress the whole project content as a `zip` file. This zip file must match the structure of the skeleton code, and it must compile in our A1 environment, or you may receive a 0 on the automarker. On Markus, please submit:

- The `zip` file. We will directly unzip your submission, build it from source and run it against public and private tests.
- A brief explanation of your implementation and testing plan as a `txt` file.

Return type & Methods under namespace llvm	Description
Type* Type::getVoidTy(*TheContext)	return "void" type.
Type* Type::getInt1Ty(*TheContext)	return 1-bit width "integer" type.
Type* Type::getInt32Ty(*TheContext)	return 32-bit width "integer" type.
ArrayType* ArrayType::get(Type* a, uint64 b)	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.
Argument* Function::getArg(unsigned i)	return a param indexed at i. class Argument is derived from llvm::Value.
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.
ConstantAggregateZero* ConstantAggregateZero::get(ArrayType* a)	Return all zero aggregate value, which means a constant array.
GlobalVariable* GlobalVariable(* TheModule , Type* a, false , GlobalVariable::CommonLinkage , Constant* c, std::string d)	Return a non-constant global variable with type a, initialized as c, name d. The variable is stored in TheModule as well.
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.
Instruction* BasicBlock::getTerminator()	Returns the terminator instruction if the block is well formed or null if the block is not well formed.
Function* Module::getFunction(std::string a)	Return a Function with name a from module.
Value* IRBuilder::CreateAlloca(Type* a, Value* ArraySize=nullptr, std::string b="")	Allocate a local variable with Type a, name b. ArraySize is set if it is an array.
Value* IRBuilder::CreateStore(Value* val, Value* ptr)	Store instruction to write val to ptr.
Value* IRBuilder::CreateLoad(Value* ptr)	Load instruction to load ptr.
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::CreateNeg(Value* val)	Create unary '-' for val.
Value* IRBuilder::CreateNot(Value* val)	Create unary '~' for val.
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::CreateAdd(Value* a, Value* b)	Create a+b.
Value* IRBuilder::CreateSub(Value* a, Value* b)	Create a-b.
Value* IRBuilder::CreateMul(Value* a, Value* b)	Create a*b.
Value* IRBuilder::CreateSDiv(Value* a, Value* b)	Create a÷b.
Value* IRBuilder::CreateICmpEQ(Value* a, Value* b)	Create a==b.
Value* IRBuilder::CreateICmpNE(Value* a, Value* b)	Create a!=b.
Value* IRBuilder::CreateICmpSLT(Value* a, Value* b)	Create a<b.
Value* IRBuilder::CreateICmpSLE(Value* a, Value* b)	Create a<=b.
Value* IRBuilder::CreateICmpSGT(Value* a, Value* b)	Create a>b.
Value* IRBuilder::CreateICmpSGE(Value* a, Value* b)	Create a>=b.

Table 2: LLVM Framework API Description


Value* IRBuilder::CreateCall(Function* a, std::vector Value* b)	Create function "a" call with params b.
Value* IRBuilder::CreateGEP(Value* a, std::vector Value* idxlst)	Create getelementptr instruction for variable a.
Value* IRBuilder::CreateRet(Value* a)	Create return instruction with a.
Value* IRBuilder::CreateRetVoid()	Create return void instruction.
void IRBuilder::SetInsertPoint(BasicBlock* bb)	This specifies that created instructions should be appended to the end of bb.
BasicBlock* IRBuilder::GetInsertBlock()	Get the inserted block bb.  == gives you the current basic block that the builder is on right now

Table 3: LLVM Framework API Description (Cont'd)

Function	Key Instructions
visitProgram	Insert all of functions into TheModule
visitVarDecl	<ul style="list-style-type: none"> • Check the variables are global or local. • Check it is array or not. • Create llvm::Value and set them into variable symbol table.
visitFuncDecl	<ul style="list-style-type: none"> • Get the corresponding llvm::Function object. • Check the function declaration has body or not. If so, allocate parameter variables and set LLVM in symbol table. • If having body, a entry basic block should be created for the function and inserted in TheBuilder. • If having body but no return expr in void function, create a void return for it.
visitIfStmt	<ul style="list-style-type: none"> • If having "else" statement, three basic blocks are created to represent "then" block, "else" block, "after" block. • CreateCondBr is needed.
visitForStmt	<ul style="list-style-type: none"> • Three basic blocks are created to represent "cond" block, "body" block, "exit" block. • First visit init expr. • Second visit cond expr and do CreateCondBr or CreateBr • Third visit "for" body and iter expr. Note that there are maybe break or return statements inside. You should jump to the end or create return instruction. • Finally set "exit" block.

Table 4: Function Implementation Instructions

visitReturnStmt	CreateRet or CreateRetVoid
visitBreakStmt	Invoke CreateBr() to directly jump to the end of for loop.
visitUnaryExpr	CreateNeg or CreateNot
visitBinaryExpr	<ul style="list-style-type: none"> • If binary op is not "AND" or "OR", create the corresponding instructions. • If A "AND" B, three blocks are created. <ul style="list-style-type: none"> – In "current" block, use CreateCondBr to check A llvm::Value. If 1, go to "slow" block; If 0, go to "out" block. – In "slow" block, check B llvm::Value and jump to "out" block. – In "out" block, create a PHI node and two incoming blocks "slow" and "current". "current" is coming with Value 0 and "slow" with Value 1. • If A "OR" B, <ul style="list-style-type: none"> – In "current" block, use CreateCondBr to check A llvm::Value. If 0, go to "slow" block; If 1, go to "out" block. – In "slow" block, check B llvm::Value and jump to "out" block. – In "out" block, create a PHI node and two incoming blocks "slow" and "current". "current" is coming with Value 1 and "slow" with Value 0.
visitCallExpr	Get Function from TheModule and createCall
visitVarExpr	<ul style="list-style-type: none"> • Acquire llvm::Value for variable. • CreateGEP for array. • CreateLoad
visitAssignmentExpr	<ul style="list-style-type: none"> • Get variable llvm::Value • CreateGEP if it is array. • CreateStore to assign the right value to the variable.
visitIntLiteralExpr	Create a 32-bit Constant object.
visitBoolLiteralExpr	Create a 1-bit Constant object.
visitScope	Before visiting child nodes, note that there may be a "return" statement in the middle of a scope.

Table 5: Function Implementation Instructions (Cont'd)