University of Toronto

CSC 488 / CSC2107 Compilers and Interpreters Winter 2022

Tutorial 7: Assignment 5 Q/A Session

You can put your questions in the chat box if you have

Some notes

(a) Possible Compiling Issues

1.

```
# Under build/ directory
# Generate executable file
clang out.bc minicio/libminicio.a -o out
# OR you can do
clang minicio/libminicio.a out.bc -o out
```

2.
Several students using sample solution code got error "`Compare operators cannot associate with each other`". It seems that your `antlr-runtime` cannot initialize a local variable correctly. **Comment the line 18 & 19 in** `GrammarCompareChecker.cpp`.

(b) Implementation issues and tips

- Each time of creating block, think that whether there could be "return" or "break" inside.
- Make sure each function should have a entry bb. Otherwise, `clang` cannot find entry point and report `undefined reference` error.
- The llvm API on A5 handout provides some default values. You can use it the same as mentioned in handouts.
- DEBUGGING: gdb (make sure `cmake -DCMAKE_BUILD_TYPE=Debug`), `std::cout`, look at `*.ll`. Compare your output `*.ll` with LLVM Language Reference Manual.
- Think about WHILE statement. **WHILE is not shown in the a5 handout**. It is similar with FOR statement.
- Looks good no one get some problems for IF, FOR, AND .... till now. Read carefully bb arrangement for such stmts/exprs in A5 handout.
- In private test case, first check simple and small input minic file, then more complicated.

Other questions?

Return type & Methods under namespace llvm	**Description**

Type* Type::getVoidTy(*TheContext)
Type* Type::getInt1Ty(*TheContext)
Type* Type::getInt32Ty(*TheContext)
ArrayType* ArrayType::get(Type* a, uint64 b)
	return "void" type.
	return 1-bit width "integer" type.
	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.
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) Value* IRBuilder::CreateSub(Value* a, Value* b) Value* IRBuilder::CreateMul(Value* a, Value* b) Value* IRBuilder::CreateSDiv(Value* a, Value* b) Value* IRBuilder::CreateICmpEQ(Value* a, Value* b) Value* IRBuilder::CreateICmpNE(Value* a, Value* b) Value* IRBuilder::CreateICmpSLT(Value* a, Value* b) Value* IRBuilder::CreateICmpSLE(Value* a, Value* b) Value* IRBuilder::CreateICmpSGT(Value* a, Value* b) Value* IRBuilder::CreateICmpSGE(Value* a, Value* b)	Create a+b. Create a-b. Create a*b. Create a÷b. Create a==b. Create a! =b. Create a<b. Create a<=b. Create a>b. Create a>=b.

Table 1: 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) BasicBlock* IRBuilder::GetInsertBlock()	This specifies that created instructions should be appended to the end of bb. Get the inserted block bb.

Table 2: LLVM Framework API Description (Cont'd)

Function	**Key Instructions**
visitProgram	Insert all of functions into TheModule
visitVarDecl	• Check the variables are global or local. • Check it is array or not. • Create llvm::Value and set them into variable symbol table.
visitFuncDecl	• 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	• If having "else" statement, three basic blocks are created to represent "then" block, "else" block, "after" block. • CreateCondBr is needed.
visitForStmt	• 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 3: Function Implementation Instructions

visitReturnStmt	CreateRet or CreateRetVoid
visitBreakStmt	Invoke CreateBr() to directly jump to the end of for loop.
visitUnaryExpr	CreateNeg or CreateNot
visitBinaryExpr	• If binary op is not "AND" or "OR", create the corresponding instructions. • If A "AND" B, three blocks are created. – 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, – 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	• Acquire llvm:Value for variable. • CreateGEP for array. • CreateLoad
visitAssignmentExpr	• 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 4: Function Implementation Instructions (Cont'd)