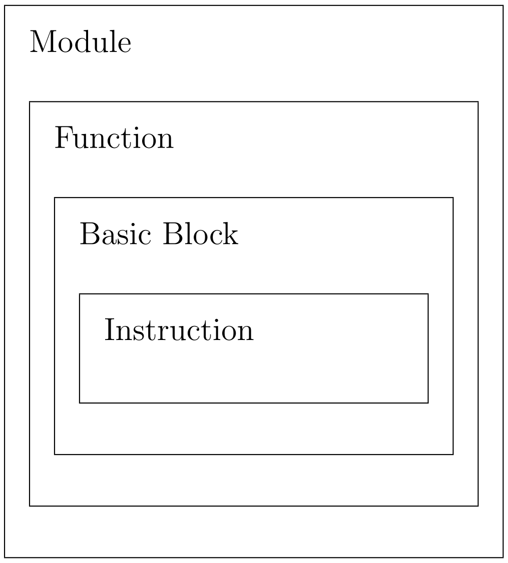
# Introduction

## LLVM Hierarchy

LLVM programs are composed of Modules, each of which is a translation unit of the input programs. Each module consists of functions, global variables, and symbol table entries.

A function definition contains a list of basic blocks, forming the CFG (Control Flow Graph) for the function. Each basic block may optionally start with a label (giving the basic block a symbol table entry), contains a list of instructions, and **ends with a terminator instruction** (such as a branch or function return).



Modules contain Functions, which contain BasicBlocks, which contain Instructions. Everything but Module descends from Value.

For more on the LLVM structure:
`https://www.cs.cornell.edu/~asampson/blog/llvm.html`

`https://llvm.org/docs/LangRef.html#high-level-structure`

## Notes on Data Types

**IRBuilder:**

The IRBuilder object is a convenience interface for creating instructions and appending them to the end of a block. Instructions can be created through their constructors as well, but some of their interfaces are quite complicated. Unless you need a lot of control, using IRBuilder will make your life simpler.

**Value:**

This is a very important LLVM class. It is the base class of all values computed by a program that may be used as operands to other values. Value is the super class of other important classes such as Instruction and Function. All Values have a Type. Type is not a subclass of Value. Some values can have a name and they belong to some Module. Setting the name on the Value automatically updates the module's symbol table.

Additional Information:
Every value has a "use list" that keeps track of which other Values are using this Value. A Value can also have an arbitrary number of ValueHandle objects that watch it and listen to RAUW and Destroy events. See llvm/IR/ValueHandle.h for details.

**BasicBlock:**

A basic block is simply a container of instructions that execute sequentially. Basic blocks are Values because they are referenced by instructions such as branches and switch tables. The type of a BasicBlock is "Type::LabelTy" because the basic block represents a label to which a branch can jump.

Additional Information:
A well formed basic block is formed of a list of non-terminating instructions followed by a single terminator instruction. Terminator instructions may not occur in the middle of basic blocks, and must terminate the blocks. The BasicBlock class allows malformed basic blocks to occur because it may be useful in the intermediate stage of constructing or modifying a program. However, the verifier will ensure that basic blocks are "well formed"

**Twine:**

A lightweight data structure for efficiently representing the concatenation of temporary values as strings.

**MDNode:**

Metadata Node. These are not needed for this lab.

## IRBuilder Functions

### Navigating Basic Blocks

```
// Get the current BasicBlock that the IRBuilder is writing to
BasicBlock*
GetInsertBlock() const;

// Get the location within a BasicBlock where the IRBuilder
// is currently inserting instructions
BasicBlock::iterator
GetInsertPoint() const;
```

```
// This specifies that created instructions should be appended to the
// end of the specified block.
void
SetInsertPoint(BasicBlock *TheBB);

// This specifies that created instructions should be inserted at the
// specified point.
void
SetInsertPoint(BasicBlock *TheBB, BasicBlock::iterator IP);
```

## Call/Return Instructions

```
// Create a function call instruction
CallInst*
CreateCall(Value *Callee, const Twine &Name="");

// Create a 'ret <val>' instruction.
ReturnInst*
CreateRet(Value *V);

// Create a 'ret void' instruction.
ReturnInst*
CreateRetVoid();
```

## Branching Instructions

```
// Create an unconditional 'br label X' instruction.
BranchInst*
CreateBr(BasicBlock *Dest);

// Create a conditional 'br Cond, TrueDest, FalseDest'
// instruction.
BranchInst*
CreateCondBr(Value *Cond, BasicBlock *True, BasicBlock *False,
             MDNode *BranchWeights = nullptr,
             MDNode *Unpredictable = nullptr);
```

## Memory Instructions

```
// Create an instruction to read an object of type Ty
// from memory at location Ptr
LoadInst*
CreateLoad(Type *Ty, Value *Ptr, const Twine &Name = "");

// Create an instruction to store Val in memory
// at location Ptr
StoreInst*
CreateStore(Value *Val, Value *Ptr, bool isVolatile = false);

// Create a Get Element Pointer (Pointer computation) instruction
Value*
CreateGEP(Value *Ptr, ArrayRef<Value *> IdxList,
          const Twine &Name = "");
```

For more information on the GEP (Get Element Pointer) Instruction:
`http://llvm.org/docs/GetElementPtr.html`

## Cast/Conversion Instructions

```
// Create a cast instruction on value V from a Floating Point
// Type to a Signed Integer Type
Value*
CreateFPToSI(Value *V, Type *DestTy, const Twine &Name = "");

// Create a cast instruction on value V from a Signed Integer
// Type to a Floating Point Type
Value*
CreateSIToFP(Value *V, Type *DestTy, const Twine &Name = "");

// Create an instruction to convert value V to type DestTy
// without changing any bits
Value*
CreateBitCast(Value *V, Type *DestTy, const Twine &Name = "");
```

### LLVM Builtin Primitive Types

LLVM representations of primitive objects can be accessed through the LLVM Type class (`#include "llvm/IR/Type.h"`) with Type functions such as the following:

```
static Type *getVoidTy(LLVMContext &C);
static Type *getFloatTy(LLVMContext &C);
static Type *getDoubleTy(LLVMContext &C);
static IntegerType *getInt32Ty(LLVMContext &C);
static IntegerType *getInt64Ty(LLVMContext &C);
```

## Arithmetic/Logic Instructions

```
// Create a 'not V' instruction
Value*
CreateNot(Value *V, const Twine &Name = "");

// Create a 'negate V' instruction
Value*
CreateNeg(Value *V, const Twine &Name = "",
          bool HasNUW = false, bool HasNSW = false);

// Create a 'negate V' instruction for a floating point
// value V
Value*
CreateFNeg(Value *V, const Twine &Name = "",
           MDNode *FPMathTag = nullptr);
```

### Bitwise Instructions

```
// Create a bitwise logical 'and' instruction of LHS and RHS
Value*
CreateAnd(Value *LHS, Value *RHS, const Twine &Name = "");

// Create a bitwise logical 'inclusive or' instruction of LHS and RHS
Value*
CreateOr(Value *LHS, Value *RHS, const Twine &Name = "");

// Create a bitwise logical 'exclusive or' instruction of LHS and RHS
Value*
CreateXor(Value *LHS, Value *RHS, const Twine &Name = "");

// Create an arithmetic shift right instruction of LHS and RHS
Value*
CreateAShr(Value *LHS, Value *RHS, const Twine &Name = "",
           bool isExact = false);

// Create a logical shift right instruction of LHS and RHS
Value*
CreateLShr(Value *LHS, Value *RHS, const Twine &Name = "",
           bool isExact = false);

// Create a shift left instruction of LHS and RHS
Value*
CreateShl(Value *LHS, Value *RHS, const Twine &Name = "",
          bool HasNUW = false, bool HasNSW = false);
```

For more on Bitwise Binary Operations:
`http://llvm.org/docs/LangRef.html#bitwise-binary-operations`

## Relational Instructions

### Integer (icmp) Instructions:

IRBuilder Functions that create integer comparison instructions take the form

```
CreateICmp[opcode](Value *LHS, Value *RHS, const Twine &Name = "");
```

Where "`[opcode]`" is replaced by one of the following LLVM IR integer comparison instructions:

| | |
|---|---|
| eq | equal |
| ne | not equal |
| ugt | unsigned greater than |
| uge | unsigned greater or equal |
| ult | unsigned less than |
| ule | unsigned less or equal |
| sgt | signed greater than |
| sge | signed greater or equal |
| slt | signed less than |
| sle | signed less or equal |

For example:

```
// Create an Integer compare not equal instruction
Value*
CreateICmpNE(Value *LHS, Value *RHS, const Twine &Name = "");

// Create a Signed Integer compare greater than instruction
Value*
CreateICmpSGT(Value *LHS, Value *RHS, const Twine &Name = "");
```

For more on integer comparison instructions:
`https://llvm.org/docs/LangRef.html#icmp-instruction`

**Floating Point (fcmp) Instructions:**
IRBuilder Functions that create floating point comparison instructions take the form

```
CreateFCmp[opcode](Value *LHS, Value *RHS, const Twine &Name = "",
                   MDNode *FPMathTag = nullptr);
```

Where "`[opcode]`" is replaced by one of the following LLVM IR integer comparison instructions:

| | |
|---|---|
| false | no comparison, always returns false |
| oeq | ordered and equal |
| ogt | ordered and greater than |
| oge | ordered and greater than or equal |
| olt | ordered and less than |
| ole | ordered and less than or equal |
| one | ordered and not equal |
| ord | ordered (no nans) |
| ueq | unordered or equal |
| ugt | unordered or greater than |
| uge | unordered or greater than or equal |
| ult | unordered or less than |
| ule | unordered or less than or equal |
| une | unordered or not equal |
| uno | unordered (either nans) |
| true | no comparison, always returns true |

For example:

```
// Create an Ordered floating point compare equal instruction
Value*
CreateFCmpOEQ(Value *LHS, Value *RHS, const Twine &Name = "",
              MDNode *FPMathTag = nullptr);

// Create an Ordered floating point compare less than instruction
Value*
CreateFCmpOLT(Value *LHS, Value *RHS, const Twine &Name = "",
              MDNode *FPMathTag = nullptr);
```

For more on floating point comparison instructions:
`https://llvm.org/docs/LangRef.html#fcmp-instruction`

## Additional Resources

Please note that the above functions are a very small subset of the IRBuilder class functions. For a more complete list of functions, and for implementation details, check out `http://llvm.org/doxygen/IRBuilder_8h_source.html`

For more practice with LLVM, consider following this introductory JIT Compiler Tutorial: `http://releases.llvm.org/2.6/docs/tutorial/JITTutorial1.html`

For a more detailed explanation of LLVM features, check out the LLVM Language Reference Manual: `https://llvm.org/docs/LangRef.html`