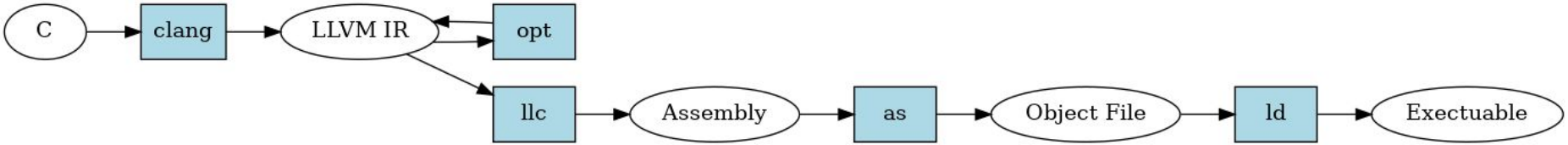
LLVM

Workshop

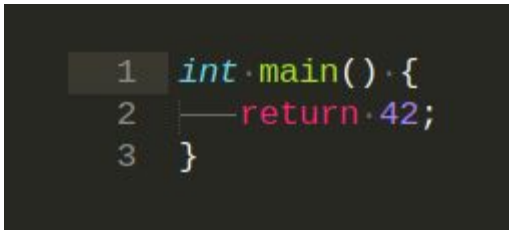
LLVM Pipeline

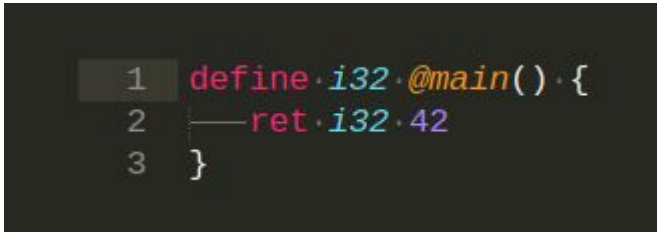


Reference commands:

```
clang -S -emit-llvm -o foo.ll foo.c
lli foo.ll
opt -S --mem2reg -o foo2.ll foo.ll
clang -o foo foo2.ll
```

Short intro to LLVM IR





LLVM Language Reference Manual: https://llvm.org/docs/LangRef.html

Another example

```
int x = 42;

void f(int a, int b) {
    int sum = a + b;
    if (sum < 5) {
        x = 0;
    }
    x = sum;
}

int main() {
    f(3, 5);
    return x;
}

```

```
@x = global i32 42

define void @f(i32 %a, i32 %b) {
    %sum = add i32 %a, %b
    %cond = icmp slt i32 %sum, 5
    br i1 %cond, label %body, label %then

body:
    store i32 0, i32* @x
    br label %then

then:
    store i32 %sum, i32* @x
    ret void
}

define i32 @main() {
    call void @f(i32 3, i32 5)
    %1 = load i32, i32* @x
    ret i32 %1
}

```

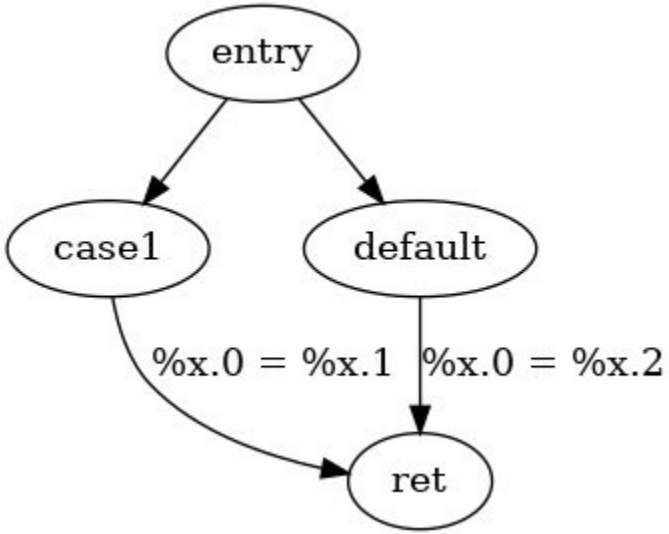
Overview of LLVM IR

- Module *(all lines)*
  - Globals *(1)*
  - Functions *(3 and 17)*
    - Basic blocks *(4, 8, 12, 18)*
      - Instructions *(4, 5, 9, 18, 19)*
      - Terminating instruction *(6, 10, 14, 20)*

```
1  @x = global i32 42
2
3  define void @f(i32 %a, i32 %b) {
4      %sum = add i32 %a, %b
5      %cond = icmp slt i32 %sum, 5
6      br i1 %cond, label %body, label %then
7
8  body:
9      store i32 0, i32* @x
10     br label %then
11
12 then:
13     store i32 %sum, i32* @x
14     ret void
15 }
16
17 define i32 @main() {
18     call void @f(i32 3, i32 5)
19     %1 = load i32, i32* @x
20     ret i32 %1
21 }
22
```

Static Single Assignment (SSA)

Variables may be assigned only once.




```
define i32 @f(i32 %a) {
    switch i32 %a, label %default [
        i32 42, label %case1
    ]
case1:
    %x.1 = mul nsw i32 %a, 2
    br label %ret
default:
    %x.2 = mul nsw i32 %a, 3
    br label %ret
ret:
    %x.0 = phi i32 [ %x.2, %default ], [ %x.1, %case1 ]
    ret i32 %x.0
}

define i32 @main(i32, i8**) {
    %3 = call i32 @f(i32 %0)
    ret i32 %3
}
```

Task 1 and 2

Play with the LLVM sample using the tools.

See the command line cheatsheet at page 9.

Introduction to the LLVM API

Skeleton provided for how to make a function pass for the LLVM middle-end.

```
1  struct FooPass : public FunctionPass {
2    // ...
3    bool runOnFunction(Function &F) override {
4      errs() << "f: " << F.getName() << "\n";
5      // return true if we've modified the IR.
6      return false;
7    }
8  };
```

LLVM Programmer's Manual: http://llvm.org/docs/ProgrammersManual.html
Writing an LLVM pass: http://llvm.org/docs/WritingAnLLVMPass.html

Module pass

Iterate over functions from the module pass.

```
1  struct QuxPass : public ModulePass {
2    // ...
3    bool runOnModule(Module &Module) override {
4      errs() << "m: " << Module.getModuleIdentifier() << "\n";
5      for (Function &F : Module.functions()) {
6        errs() << "  f: " << F.getName() << "\n";
7      }
8      // return true if we've modified the IR.
9      return false;
10   }
11 };
12
```

Iterating over the basic blocks of a function

```
1  Function &F = ...
2  for (BasicBlock &BB : F) {
3  ──// do stuff with basic block.
4  ──errs() << "bb: " << BB.getName() << "\n";
5  }
6
```

Iterating over the instructions of a basic block

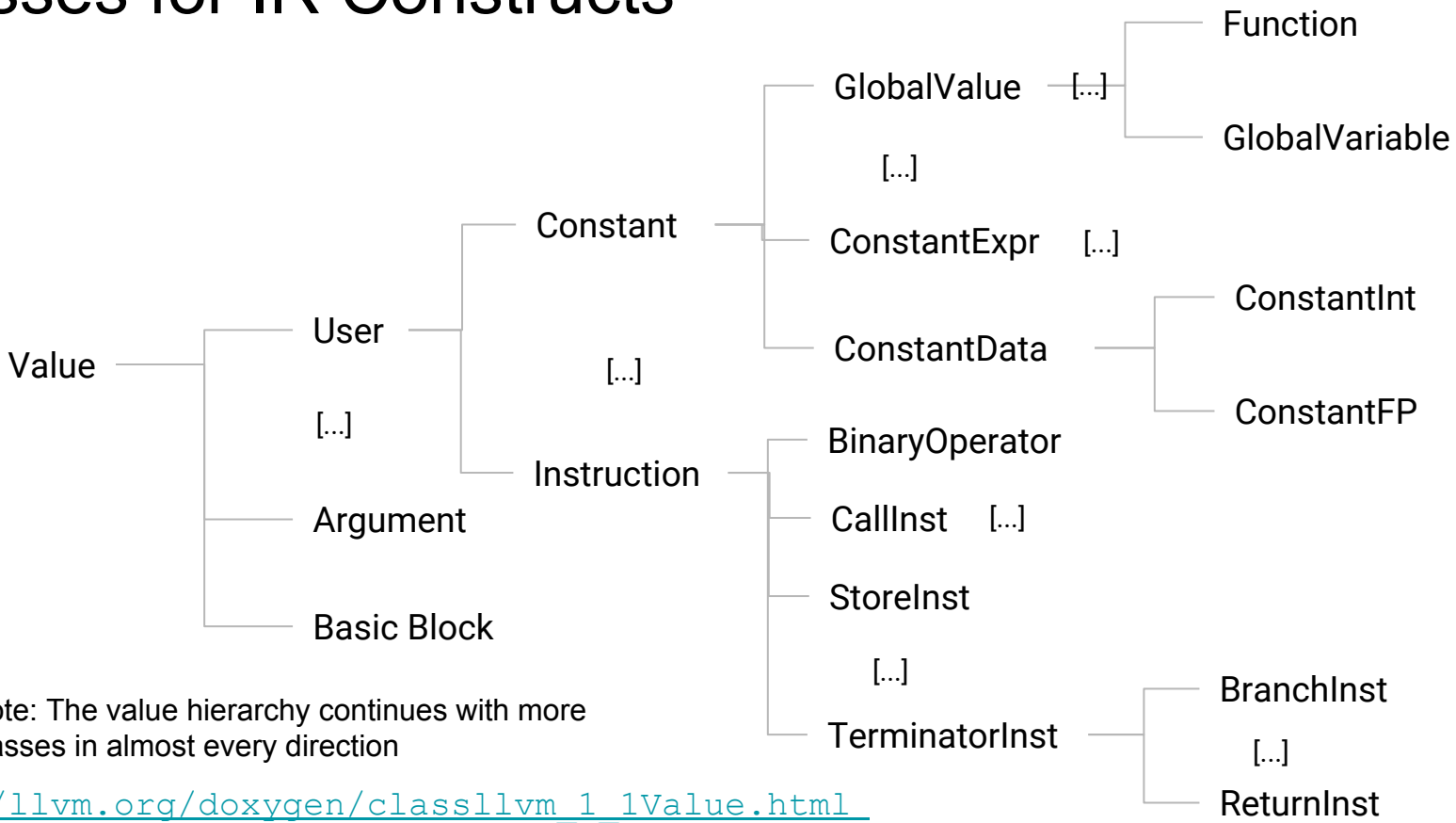
```
1  BasicBlock &BB = ...
2  for (Instruction &I : BB) {
3    // do stuff with instruction.
4    errs() << "inst: " << I << "\n";
5  }
6
```

Handling specific instruction types

```
1  Instruction &I = ...
2  if (CallInst *CI = dyn_cast<CallInst>(&I)) {
3    const Function *callee = CI->getCalledFunction();
4    errs() << "callee: " << callee->getName() << "\n";
5  }
6
```

List of instruction types: http://llvm.org/doxygen/classllvm_1_1Instruction.html

Classes for IR Constructs



Note: The value hierarchy continues with more classes in almost every direction

http://llvm.org/doxygen/classllvm_1_1Value.html

Debug output

```
1  #define DEBUG_TYPE "skeleton"
2
3  namespace {
4  struct SkeletonPass : public FunctionPass {
5      // ...
6      bool runOnFunction(Function &F) override {
7          DEBUG(dbgs() << "I saw a function called " << F.getName() << "!\n");
8          return false;
9      }
10 }
```

```
$ opt -load libSkeletonPass.so -S -o foo2.ll foo.ll -skeleton -debug-only=skeleton

I saw a function called main!
```

Task 3

Write you first LLVM middle-end analysis pass.

See the API cheatsheet at page 10.

Inserting new function declarations

```
1  Function &F = ...
2  LLVMContext &Context = F.getContext();
3  Module *Module = F.getParent();
4  FunctionType *GType = FunctionType::get(
5      Type::getVoidTy(Context),
6      {
7          Type::getInt8PtrTy(Context),
8          Type::getInt32Ty(Context)
9      },
10     false);
11 Constant *GFunc = Module->getOrInsertFunction("g", GType);
12
```

```
1  declare void @g(i8*, i32)
2
```

Insert new instructions

```
Instruction &I = ...
if (ReturnInst *RI = dyn_cast<ReturnInst>(&I)) {
    IntegerType *i32 = Type::getInt32Ty(Context);
    IRBuilder<> Builder(RI);
    Builder.CreateCall(
        cast<Function>(GFunc),
        {
            Builder.CreateGlobalStringPtr("test", ".str"),
            ConstantInt::get(i32, 3),
        }
    );
}
```

```
@.str = private unnamed_addr constant [5 x i8] c"test\00"

define i32 @f() {
    call void @g(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str, i32 0, i32 0), i32 3)
    ret i32 42
}
```

IR Builder: http://llvm.org/doxygen/classllvm_1_1IRBuilder.html

Replacing values

```
1  CallInst CI = ...
2  IRBuilder<> Builder(CI);
3  CI->replaceAllUsesWith(Builder.getInt32(42));
4
```

```
1  ; before
2  ——%call = call i32 @f()
3  ——ret i32 %call
4
5  ; after
6  ——%call = call i32 @f()
7  ——ret i32 42
```

Task 4, 5 and 6

Write middle-end passes to do transformation and instrumentation.