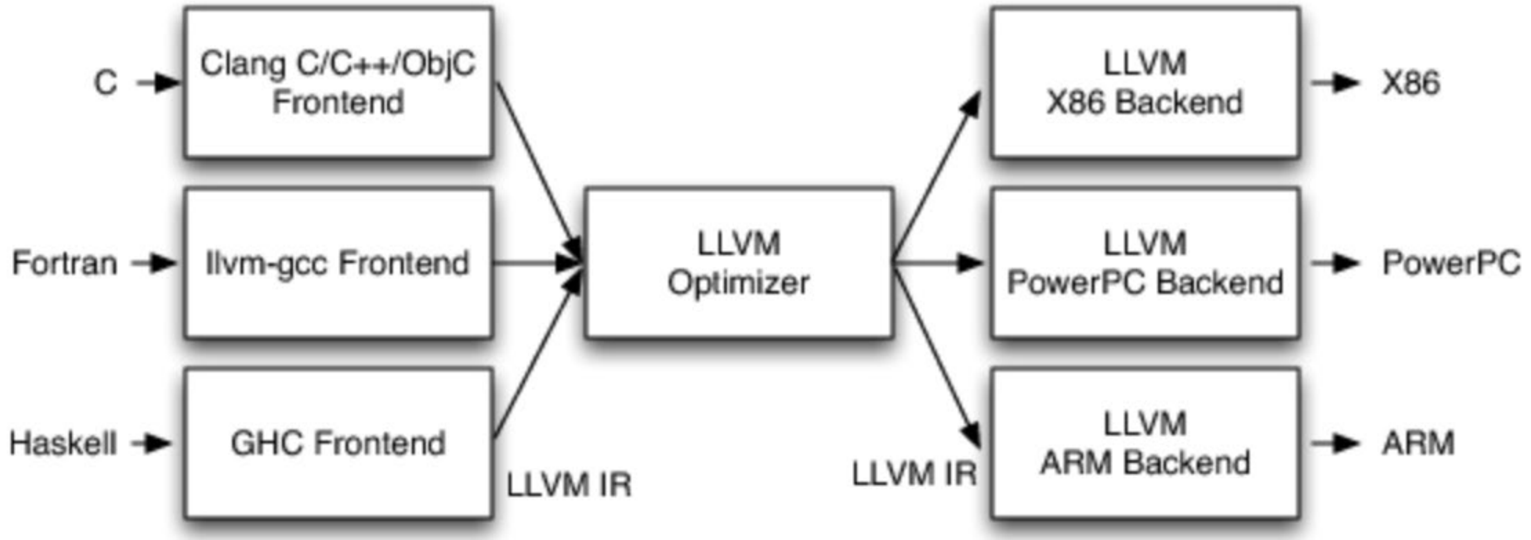


LLVM OPTIMIZATION

Andrew Miller and James Reboulet

Image: https://www.aosabook.org/en/llvm.html

AGENDA

1. opt: the modular LLVM optimizer
   - opt command and options
   - What is a Pass?
2. Writing a Pass
   - Declarations
   - Superclasses
3. Analysis Passes
   - Stack Safety Analysis
   - Region Analysis
   - Memory Dependance
   - Natural Loop Information
4. Transform Passes
   - Dead Code Elimination
   - Loop Optimization
5. Utility Passes
   - Module Verifier
   - View CFG of function.
   - View dominator trees.
6. LLVM Link-Time Optimizer
7. Additional libraries for opt
   - The LLVM gold plugin

opt [*options*] [*filename*]

Two main operations:

- opt –analyze performs some specified analysis on the LLVM source input, and prints the analysis results
- opt (without –analyze) aims to produce an output file that is optimized with a specified type of pass

Additional Options:

- -f enables opt to write raw bitcode to the terminal
- -S sets opt to write the output in LLVM IR rather than bitcode
- -load=<plugin> registers new types of passes via a dynamic library

-{passname} indicates the type of pass to execute on the given IR file

WHAT IS A PASS?

- In LLVM, a “pass” is a generic term for an operation on a unit of IR.
- The purpose of the pass is usually either analysis, or transformation.
- The operation a pass performs on a unit of IR can be a mutation of the IR or a calculation of some characteristic about the IR.

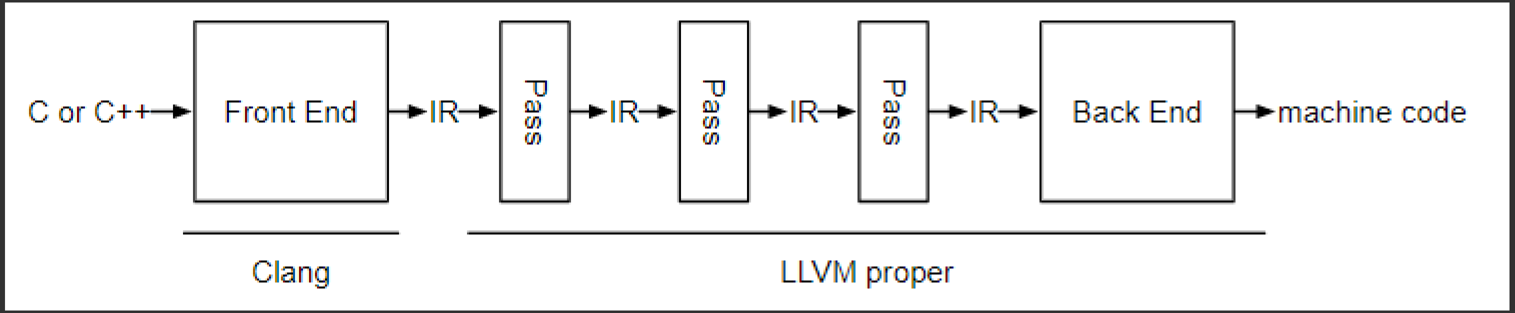


Image: https://www.cs.cornell.edu/~asampson/blog/llvm.html

WRITING A PASS

- Include these headers:
  - `llvm/Pass.h`
  - `llvm/IR/Function.h`
  - `llvm/Support/raw_ostream.h`

These are necessary because we are going to be operating on IR “Functions” and we need to be able to write an output

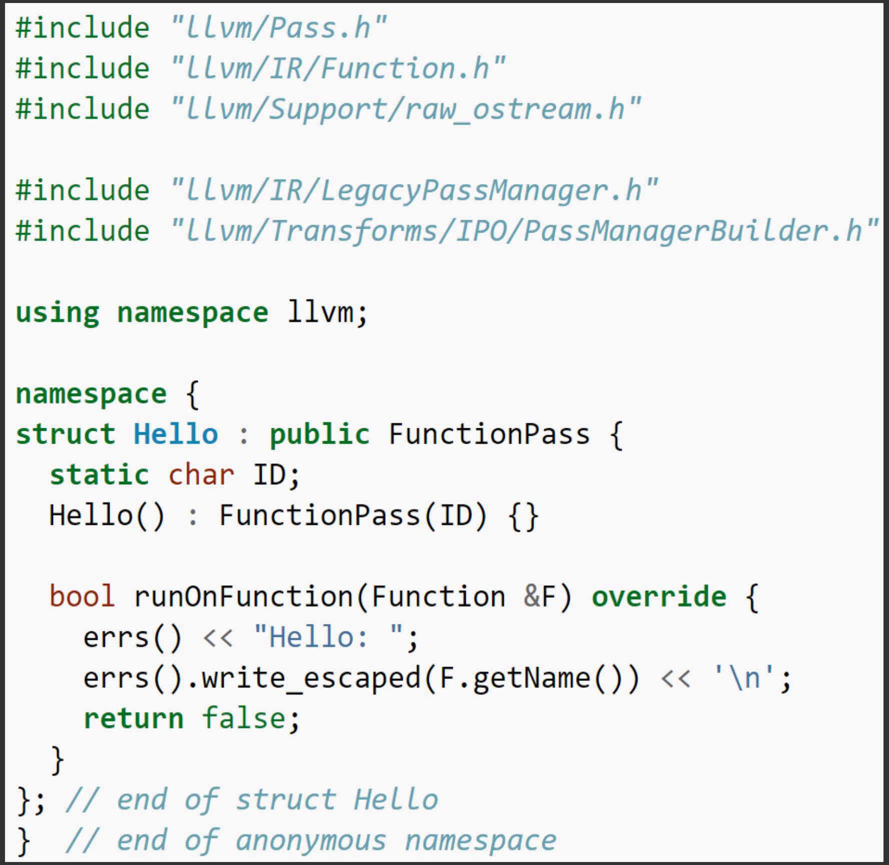
- It is recommended to develop a pass with “`using namespace llvm;`” and in an anonymous namespace
- The new pass should be a subclass of `FunctionPass`

WRITING A PASS

- All child classes of `FunctionClass` need to implement this:
  `bool runOnFunction(Function &F)`
  which overrides an abstract virtual method from `FunctionClass`
- `runOnFunction` is where the actual meat of the pass exists. This defines the behavior of the pass on a function-by-function basis as it optimizes or otherwise analyzes the IR.

WRITING A PASS – DECLARATION EXAMPLE

- The pass, itself, is actually a self-contained struct.
- The following is an example of writing a pass for a “Hello World” pass struct declaration:



```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"

using namespace llvm;

namespace {
struct Hello : public FunctionPass {
  static char ID;
  Hello() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override {
    errs() << "Hello: ";
    errs().write_escaped(F.getName()) << '\n';
    return false;
  }
}; // end of struct Hello
}  // end of anonymous namespace
```

WRITING A PASS

- After the declaration of the new pass, the code requires a template `static llvm::RegisterPass< passClassName > X(<args>)` function with the following arguments:
  - A string representing the command line argument to use the pass
  - A string with the full name of the pass
  - Two additional bool arguments describing the behavior of the pass:
  - Argument three is set to true if the pass walks a given Context-Free Grammar without modifying it.
  - Argument four is set to true if the pass is an analysis pass.

WRITING A PASS – REGISTER PASS CODE

- The code for the function RegisterPass() and the methods following it are static and added to the same header file.
- The following code is inserted within the header file after the pass struct is defined:

```
char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);
```

WRITING A PASS – CHOOSING PASS SUPER-CLASSES

- There are additional super classes besides the FunctionPass superclass which was previously described.
- Depending on which superclass is selected, its subclasses are allowed to abide by different requirements.
- When choosing a superclass to derive from, the most specific one which meets all of the individual developer's specified requirements should be selected.
- If not, the compiler cannot optimize how individual passes are run. For example, if all of our passes are inheriting from ModulePass and we run all ModulePasses on every function in our program, the entire program has to be checked every time we run a pass.
  - It would have been better to use FunctionPass, so that each function could have been independently optimized or the passes could have been pipelined together, as long as the passes had no dependencies (i.e. run all passes on FunctionX() along with those on FunctionY()).
- Examples of these superclasses will be presented on the next three slides.

WRITING A PASS - SUPER CLASSES – IMMUTABLE CLASS

- The immutable class has simple requirements: All of its derived passes:

1) Do not run.
2) Do not change state.
3) Never need to be updated.

- If they are used, they are used for providing information about the target machine that the IR code is being compiled for, or other static information that can affect various transformations.
- Although they don’t run or modify IR, they are not useless.

WRITING A PASS - SUPER CLASSES – MODULEPASS CLASS

- ❑ The ModulePass class is the most general of all super classes that can be used.
- ❑ Since derived classes treat the entire program as a unit: i.e.
  - Function bodies are not referred to in any particular order.
  - Functions cannot be added or removed.
- ❑ Result: No optimization for this function’s execution takes place, since nothing is known about the behavior of this class and its derivatives.
- ❑ Module passes can use function-level passes (denominators), if a particular function does not require a module or any immutable passes.
  - ▪ However, the analysis must have previously run on those functions.
- ➢ If you want to optimize an entire program module and potentially be able to optimize it by adding or removing functions, derive from this superclass.
- ➢ The FunctionPass class does not enable this feature.
- ➢ In addition, to use this class, you have to implement the virtual bool runOnModule(Module &M) method (to do the actual “work” of the class).

WRITING A PASS – OTHER SUPERCLASSES

- There are other various classes, such as the
- CallGraphSCCPass class
- LoopPass (for optimizing individual loops within a function, independent of other loops)
- RegionPass (for optimizing various nested exit or entry points within a particular function)
- MachineFunctionPass class (which is derived from FunctionPass, but has additional restrictions – more specific)
- Each superclass has various virtual methods which can be and/or are required to be implemented for them to perform their optimization work.
- More information is available on the link:
- Writing an LLVM Pass — LLVM 12 documentation .
- We will now move on to explain the various types of passes.

ANALYSIS PASSES

- LLVM includes over 40 different Passes for use with the "-analysis" option
- These passes compute information relating to a unit of IR without altering the input
- Analysis passes can take the output from one pass and use it as input for another

EXAMPLES OF ANALYSIS PASSES

-instcount: counts all instructions in the source IR and reports the result

-domtree: constructs a dominator tree for finding forward dominators

-stack-safety: determines if variables are safe from memory access bugs

-regions: detects regions in a function with one entry and one exit

-memdep: finds dependencies for memory operations

-loops: finds natural loops in the IR

EXAMPLES OF ANALYSIS PASSES – STACK SAFETY PASS

- -stack-safety ensures variables are "safe"
- "safe" means that the variable is not used out-of-scope or accessed out-of-bounds

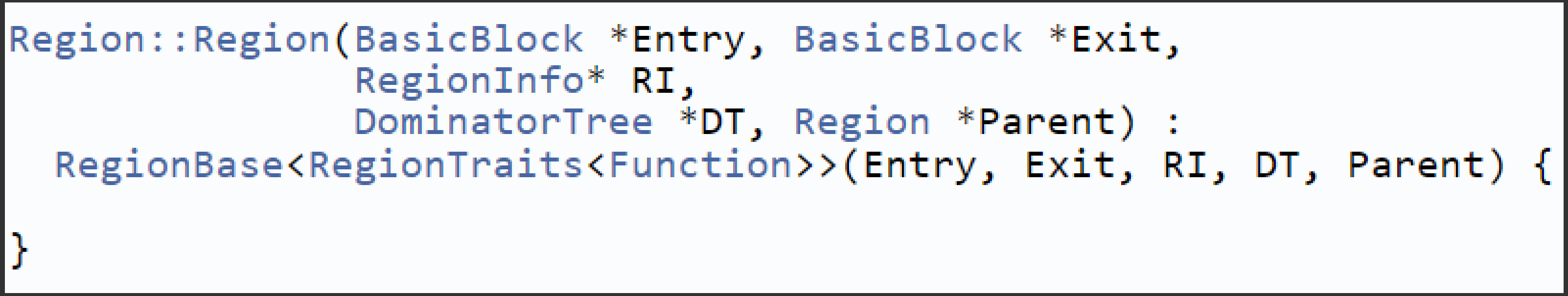
Implementation:

1.) The "local" (intra-procedural) stage searches the Functions depth-first to collect uses of each variable

2.) The "global" (inter-procedural) resolves what happens to variables after they are passed as function arguments by looking at a single module and performing a depth-first-search on the function calls

EXAMPLES OF ANALYSIS PASSES - REGIONS

- -regions searches the module's structure for subgraphs with a single entry and a single exit point
- Depends heavily on the dominator tree of the module, as it scans this tree to find the regions. Most passes maintain this tree.



```
Region::Region(BasicBlock *Entry, BasicBlock *Exit,
               RegionInfo* RI,
               DominatorTree *DT, Region *Parent) :
  RegionBase<RegionTraits<Function>>(Entry, Exit, RI, DT, Parent) {

}
```

EXAMPLES OF ANALYSIS PASSES – MEMORY DEPENDENCE ANALYSIS

- -memdep uses the AliasAnalysis superclass
- It provides high-level dependence information regarding instructions that use memory
- For each load operation, it will show the “store” instructions that that address is populated by
- Used by Dead Store Elimination, Global Value Numbering, and MemCpy optimization passes

EXAMPLES OF ANALYSIS PASSES – NATURAL LOOP INFORMATION

- -loops makes use of LLVM’s LoopInfo class to detect natural loops in the control-flow graph
- “natural loops” are defined as a subset of nodes in the CFG that meet the following criteria:
  1. The subgraph of every node in the proposed loop is strongly connected
  2. All edges going into the subset come in through a single node, known as the header
  3. There is no greater subset with these properties – no other nodes could be added in to fit the criteria
- This pass is liable to detect separate natural loops that share a header node as a single large loop

TRANSFORMATION PASSES

- LLVM is also equipped with numerous transformation passes, which are to be used without the “-analyze” option
- Transformation passes mutate the IR, to optimize it
- They cannot be daisy-chained: this would cause new forms of the original IR to be passed around, unlike with analysis where the IR remains the same.

EXAMPLES OF TRANSFORMATION PASSES – DEAD-CODE ELIMINATION

- Dead code can mean source code which is executed, but which is not utilized in any other computation. For example:

```
int func(int y){
int x = y +1;  //This is a "dead" line of code.
 y =  y + 2;
 return y;
```

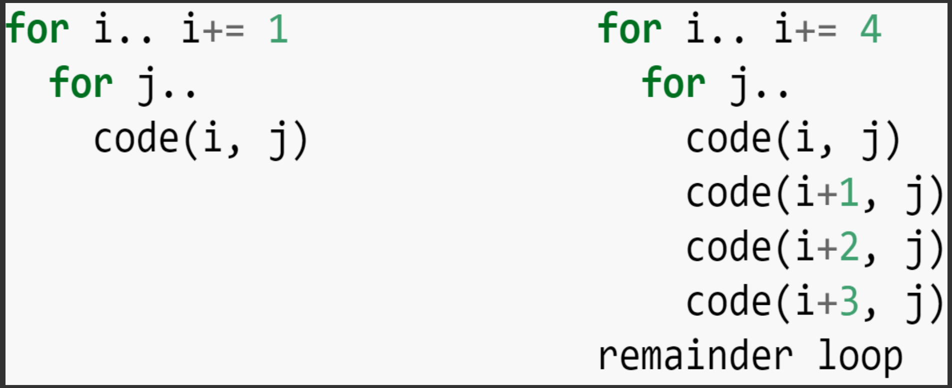
- If the dead code happened to be a loop that was defined and had run 100,000 times, but never affected the output of the program, this code would have affected runtime, even though it hadn't actually done anything, except consume memory and CPU clock cycles.
- LLVM, provides the ability to specify multiple types of transformation passes to remove dead code and optimize the IR code.
- However, the optimizer must be careful not to remove code which could, in some way, affect the final output of the program and its removal would result in unintended bugs.
- This is especially true if the code was incorrect, affected the output, and nobody knew it was there for 5 years, but designed the rest of the program around the buggy code. By removing it, it breaks 5 years worth of "product innovation." Sometimes it's better not to update the compiler, if the update were to remove this, supposedly, dead code, unless you are leaving the company and you want them to remember you.

TRANSFORMATION PASSES – DEAD-CODE ELIMINATION OPTIONS

- -die : Removes all dead instructions.
- -dce : Does the same thing as dead instruction elimination, except that checks other instructions that were potentially used by previously “dead” instructions to see if they have “died” as well.
- -deadargelim : Deletes dead arguments from internal functions.
- -adce : Aggressive Dead Code Elimination – simply assumes all values are dead until proven otherwise. It, therefore, tries to eliminate a lot of code.
- -globaldce : Dead Global Elimination – Aggressive algorithm to eliminate unreachable internal globals from a program.
- -strip-dead-prototypes : The pass loops over all functions in the input module, looks for dead declarations and removes them. This could be applicable to unused library functions or other functions which have no implementation available.
- -loop-deletion : Deletes dead loops.

TRANSFORMATION PASSES – LOOP OPTIMIZATION

- LLVM also enables various types of loop-optimization passes.
- The simplest optimization would be to unroll a loop, to eliminate branch calls which tend to utilize additional clock cycles. The caveat is, however, that the binary uses more file space.
- -loop-unroll : This pass simply unrolls the loop. If there are embedded loops, they are fully unrolled as well.
- -loop-unroll-and-jam : This method is fancier. It unrolls the outer loop fully, but within each outer loop iteration, the associated inner loop is "jammed" into the middle of an iteration.



```
for i.. i+= 1                 for i.. i+= 4
  for j..                       for j..
    code(i, j)                    code(i, j)
                                  code(i+1, j)
                                  code(i+2, j)
                                  code(i+3, j)
                              remainder loop
```

-When the variables are shared, as in the case of the variable, i , this prevents the issue of having to first unwind all of the j's for i =1, then i = 2, etc., branch at least twice each time code() is called, and create a new stack entry for each unique call of the function, code().

-This can save a great deal of time over a generic –loop-unroll .

TRANSFORMATION PASSES – LOOP OPTIMIZATION

- A couple of more interesting optimization passes for loops.
- -indvars: Canonicalizes loop induction variables. This is better illustrated by the following example:
- This
```
for (i = 7; i*i < 1000; ++i)
```
becomes the following:
```
for (i = 0; i != 25; ++i)
```
  - ✓Notice that instead of the value of i needing to be incremented by 1 , squared, 25 times and compared against the value of 1000 (since i = (7 + 25) * (7+25) = 1024). It makes life easier and saves CPU operations to initially set i to 0, remove the squaring of i, and simply terminate the loop when i = 25.
  - ✓However, do not use this if i is part of an expression that computes an exit value, or reducing the value of i could create issues. For instance, if the loop returns (i * i) in its result.
- -loop-unswitch: Transforms loops that have invariant conditions into multiple loops ???!

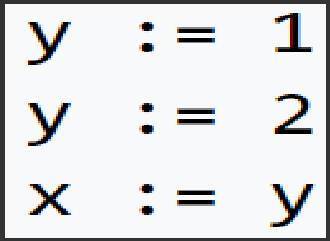
```
for (...)
    A
    if (lic)
        B
    C
```
becomes
```
if (lic)
    for (...)
        A; B; C
else
    for (...)
        A; C
```
  - This reduces the additional 2 branches which occur every time "lic" evaluates to true within the non-optimized for-loop. Many times, reducing branching saves time.
  - Be careful though. This pass can double the amount of code exponentially

ADDITIONAL USEFUL TRANSFORMATION PASSES

- -tailcallelim : Eliminates the tail call on a recursive function and converts the call into a branch to the beginning of the recursive function, thus effectively converting the recursive call into a loop.

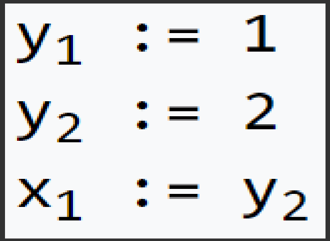
- -sroa : Scalar Replacement of Aggregates. This is a nice optimization when dealing with an array where multiple elements are repeatedly being assigned values temporally during program execution. For instance, in the image to the right, only the second value of y is used to compute x, so the memory location of y has to be written to twice. Instead, this transformation pass allocates y twice, so that it each value only needs to be assigned once. The “Optimized” version is in SSA form. This doesn’t seem useful, however, in a C++ vector- or set- variable-size array setting.

Not Optimized



```
y  :=  1
y  :=  2
x  :=  y
```

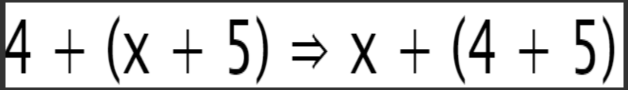
Optimized - SSA



$$y_1 := 1$$
$$y_2 := 2$$
$$x_1 := y_2$$

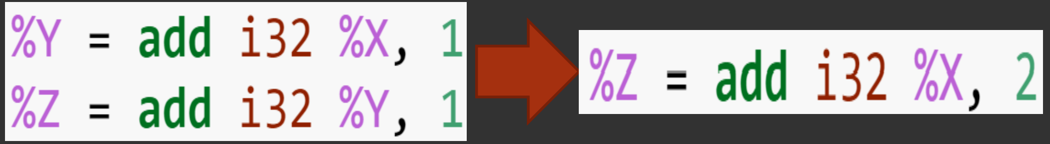
- -mergereturn: Ensures that functions only have at most one return instruction in them. James’ compiler code could stand to use this optimization.

- -reassociate : Reassociate expressions. Changes the order of commutative expressions, so that constants can better propagate through the code.



$$4 + (x + 5) \Rightarrow x + (4 + 5)$$

- -instcombine : Combines redundant instructions into a single instruction.



```
%Y = add i32 %X, 1
%Z = add i32 %Y, 1
```

```
%Z = add i32 %X, 2
```

UTILITY PASSES – MODULE VERIFIER

- -verify verifies that an LLVM IR code is valid
- Most useful after an optimization pass that is currently being tested
- Performs general semantic analysis of the input IR module:
  - Checks that all binary operators have matching types
  - Code is in valid static single assignment form (SSA)
  - No illegal labelling of structures or void values
- Does not provide full security verification, but is useful for making sure the code is properly formed

UTILITY PASSES – ADDITIONAL EXAMPLES

- View the CFG (Context-Free Grammar) of a given function.

 -view-cfg : You can view the CFG using the GraphViz tool with the function body omitted.

- View the Dominator Tree of IR code (shown below – referenced from text, p.657).

 -view-dom : You can view the dominator tree (parse tree) of the given IR source code. This may potentially be useful if –verify fails and you want to possibly figure out why.

 -view-dom-only: View dominance tree of function headers only with no function bodies.

LLVM's Analysis and Transform Passes — LLVM 12 documentation

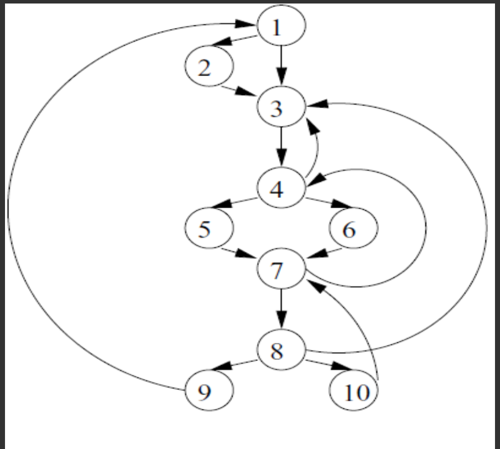
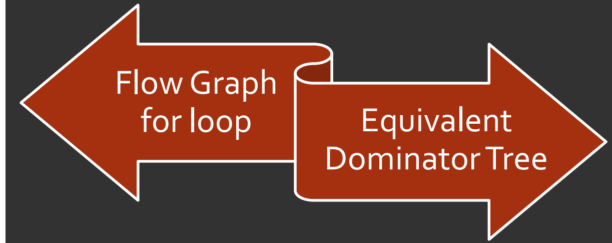
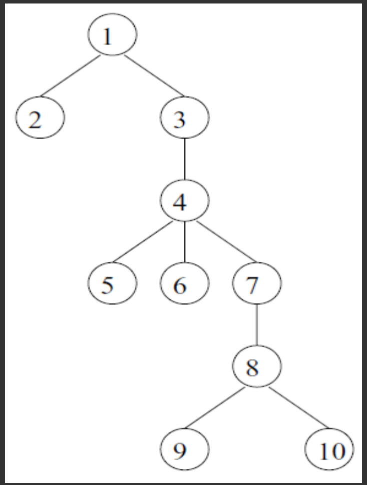


Figure 9.38: A flow graph



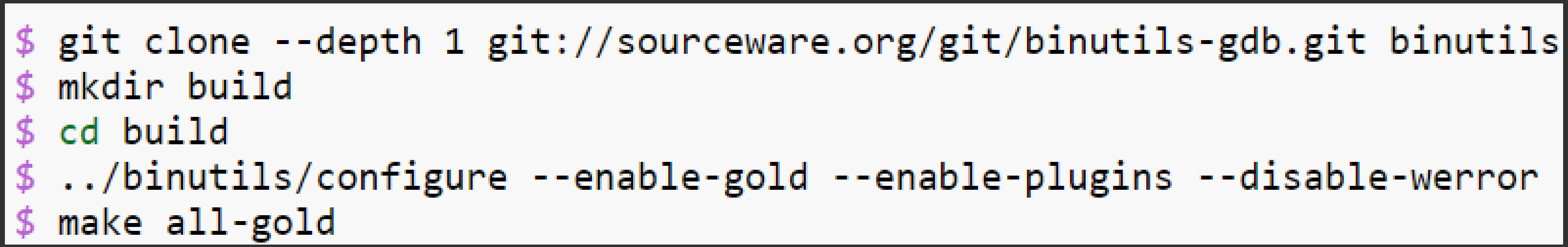


LLVM LINK-TIME OPTIMIZER

- The Linker uses `libLTO.so` to handle LLVM bitcode files
- `libLTO.so` provides an abstract C interface to allow use of the inter-procedural optimizer without exposing the internals of LLVM
- Multiple phases of communication between `libLTO` and the linker
  1. Read all object files and gather symbol information
  2. Resolve symbols with the global symbol table
  3. Optimization: lto_codegen_compile() combines modules, performs passes, and returns one object file
  4. Linker reads object file and updates the global symbol table with any changes

THE LLVM GOLD PLUGIN

- The gold plugin is required for LTO on Linux systems
  - Check whether gold is running with `/usr/bin/ld –v`
  - Check for plugin support with `/usr/bin/ld –plugin`
  - Download, Configure, and Build:



```
$ git clone --depth 1 git://sourceware.org/git/binutils-gdb.git binutils
$ mkdir build
$ cd build
$ ../binutils/configure --enable-gold --enable-plugins --disable-werror
$ make all-gold
```

To use gold after loading the plugin, call clang with the –flto option