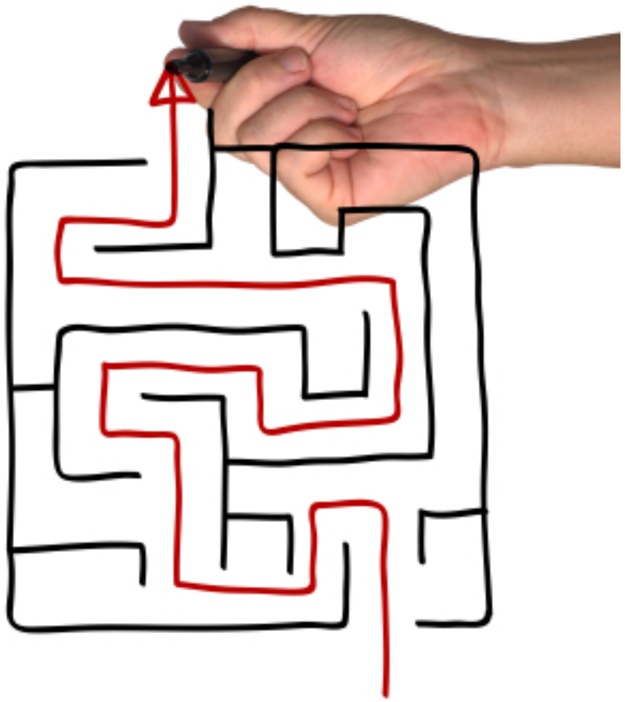
ITERATING AND CASTING

DCC 888

LLVM Provides a Rich Programming API

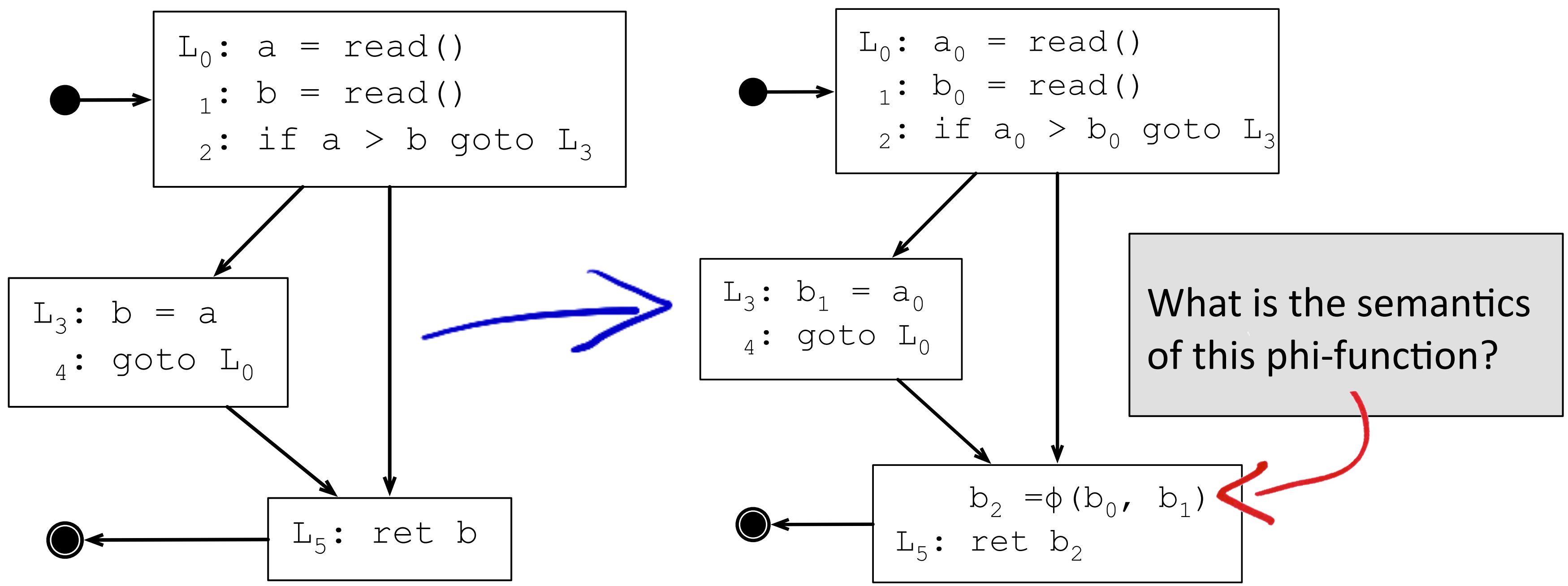
- Several ways to navigate through common structures:
  - instructions in a function
  - uses of an instruction
  - operands of instructions
  - blocks within functions
- Several type inference facilities:
  - Dynamic casts
  - Instance-of test
- Several ways to change the CFG
  - Add/remove instructions
  - Add/remove basic blocks
- The best reference is the programmer's manual[†].



[†]: LLVM Programmer's manual: `http://llvm.org/docs/ProgrammersManual.html`

Example: Printing Phi-Nodes

- LLVM adopts the Static Single Assignment form as its internal representation[†].
  - Each program variable has only one definition site.
  - This representation simplifies many analyses.



[†]: *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, 1999

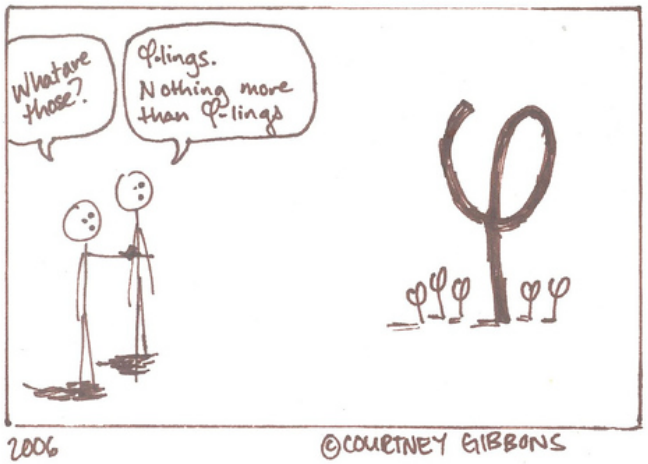
Printing Phi-Nodes

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

using namespace llvm;
namespace {
  struct Count_Phis : public FunctionPass {
    static char ID;
    Count_Phis() : FunctionPass(ID) {}
    virtual bool runOnFunction(Function &F) {
      errs() << "Function " << F.getName() << '\n';
      for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
        if (isa<PHINode>(*I))
          errs() << *I << "\n";
      }
      return false;
    }
  };
}
char Count_Phis::ID = 0;
static RegisterPass<Count_Phis> X("countphis",
    "Counts phi-instructions per function");
```

1) How do we go over the instructions in the function?
2) How can we find out that a given instruction is a phi-function?

This pass prints the phi-instructions in each function of a module.



Running the Pass

```
int foo(int n, int m) {
  int sum = 0;
  int c0;
  for (c0 = n; c0 > 0; c0--) {
    int c1 = m;
    for (; c1 > 0; c1--) {
      sum += c0 > c1 ? 1 : 0;
    }
  }
  return sum;
}
```

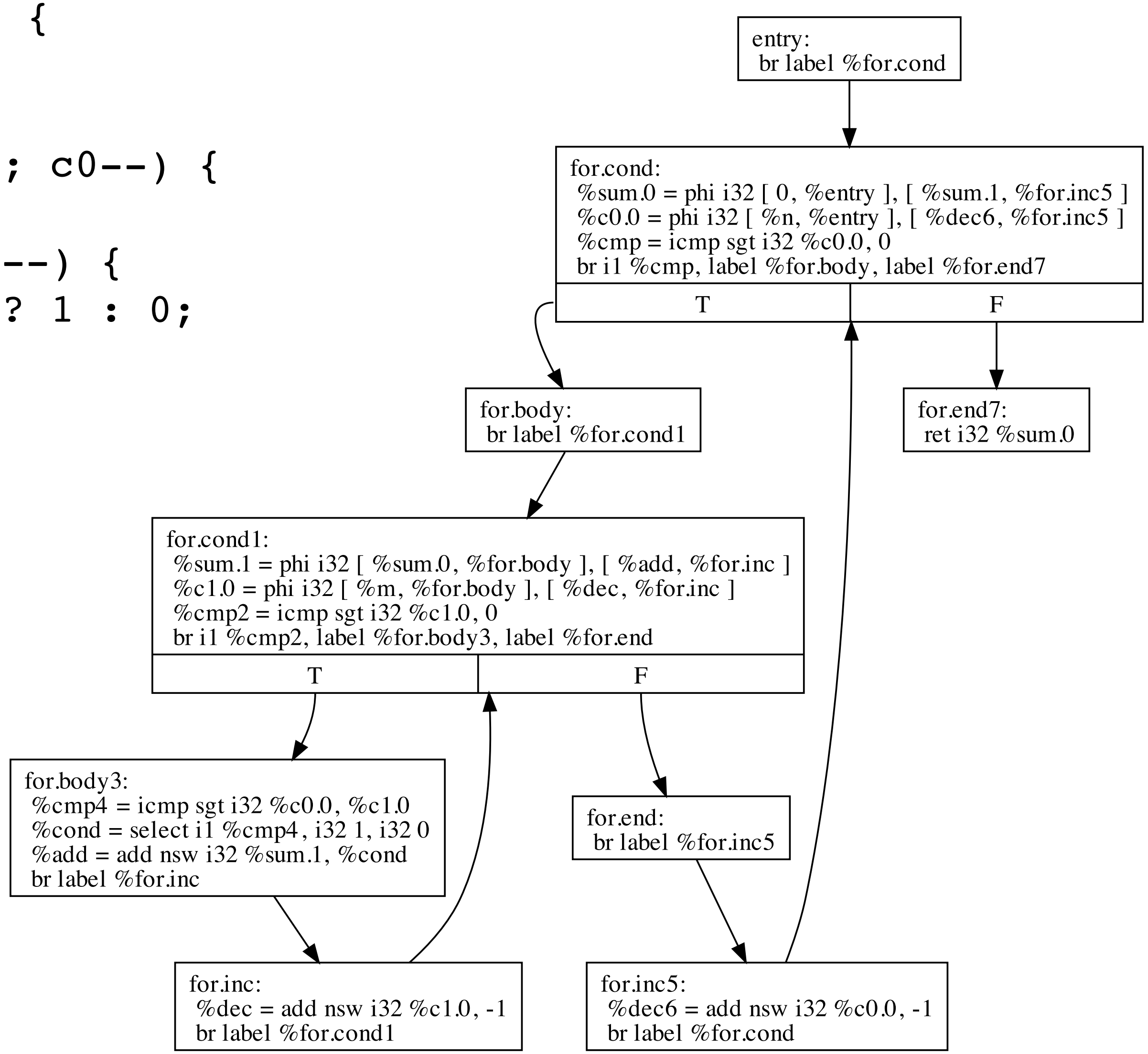
1) How to generate bytecodes for this program?

2) How to convert it to SSA form?

3) Can you guess how many phi-functions we will have for this program?

Running the Pass

```c
int foo(int n, int m) {
  int sum = 0;
  int c0;
  for (c0 = n; c0 > 0; c0--) {
    int c1 = m;
    for (; c1 > 0; c1--) {
      sum += c0 > c1 ? 1 : 0;
    }
  }
  return sum;
}
```



1) How to produce this CFG out of the program above?

2) How to run our pass on this prog?

Running the Pass

```
$> clang -c -emit-llvm c.c -o c.bc

$> opt -mem2reg c.bc -o c.rbc

$> opt -load dcc888.dylib -countphis -disable-output c.rbc
```

```
Function foo
  %sum.0 = phi i32 [ 0, %entry ], [ %sum.1, %for.inc5 ]
  %c0.0 = phi i32 [ %n, %entry ], [ %dec6, %for.inc5 ]
  %sum.1 = phi i32 [ %sum.0, %for.body ], [ %add, %for.inc ]
  %c1.0 = phi i32 [ %m, %for.body ], [ %dec, %for.inc ]
```

```
virtual bool runOnFunction(Function &F) {
  errs() << "Function " << F.getName() << '\n';
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    if (isa<PHINode>(*I))
      errs() << *I << "\n";
  }
  return false;
}
```

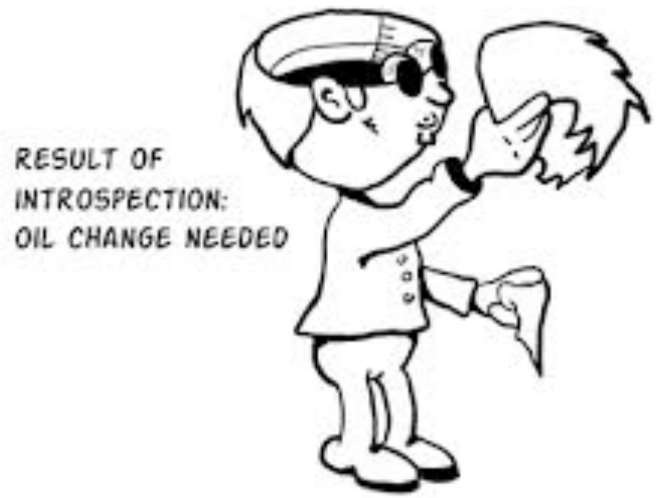
Iterating over Instructions

```
virtual bool runOnFunction(Function &F) {
  errs() << "Function " << F.getName() << '\n';
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I)
   if (isa<PHINode>(*I))
    errs() << *I << "\n";
  return false;
 }
};
```

There are two basic ways to iterate over instructions. Either we grab an inst_iterator from a function, or we iterate through the instructions in the basic blocks using a BasicBlock::iterator.

```
for(Function::iterator bb = F.begin(), e = F.end(); bb != e; ++bb)
  for(BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e; ++i)
    Instruction* inst = i;
```

Runtime Type Introspection




```
virtual bool runOnFunction(Function &F) {
  errs() << "Function " << F.getName() << '\n';
  for (inst_iterator I = inst_begin(F),
                     E = inst_end(F); I != E; ++I)
   if (isa<PHINode>(*I))
    errs() << *I << "\n";
  return false;
 }
};
```

LLVM provides a very expressive API for runtime type inference (RTTI). The isa<> template is a way to know the dynamic type of a value. The test isa<T>(V) returns true if V is an instance of type T, and false otherwise. This template is part of the LLVM library, and not part of the C++ Standard Library. As such, instances of `Value` and `Instruction`, in LLVM, implement a `classof` method, which makes the isa<> test possible.

More on RTTI

- LLVM has five operations to ask the runtime type of a value, but three are particularly used:
  - isa<T>(V) which we just saw
  - cast<T>(V), which works like a checked type coercion
    - It causes an assertion failure if applied on a wrong type
  - V' = dyn_cast<T>(V), which either converts V to V', or returns NULL

In addition to these three operations, LLVM also provides cast_or_null and dyn_cast_or_null<>, which can handle null pointers, contrary to cast<> and dyn_cast<>

Example of Verified Cast

```
virtual bool runOnFunction(Function &F) {
  errs() << "Function " << F.getName() << '\n';
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    if (isa<PHINode>(*I)) {
      errs() << *I << "\n";
      errs() << "  - has " <<
        cast<PHINode>(*I).getNumIncomingValues() << " arguments.\n";
    }
  }
  return false;
}
```

This method uses a dynamic cast to invoke on *I, which is an instance of PHINode, a method that is defined in that class. Notice that the cast is necessary, for getNUMIncomingValues is not defined on inst_iterators.

What does this program print?

The Static Cast in Action

Assume that we have modified our pass countphis with the previous method runOnFunction.

```
$> clang -c -emit-llvm c.c -o c.bc

$> opt -mem2reg c.bc -o c.rbc

$> opt -load dcc888.dylib -countphis -disable-output c.rbc
```

```
Function foo
  %sum.0 = phi i32 [ 0, %entry ], [ %sum.1, %for.inc5 ]
  - has 2 arguments.
  %c0.0 = phi i32 [ %n, %entry ], [ %dec6, %for.inc5 ]
  - has 2 arguments.
  %sum.1 = phi i32 [ %sum.0, %for.body ], [ %add, %for.inc ]
  - has 2 arguments.
  %c1.0 = phi i32 [ %m, %for.body ], [ %dec, %for.inc ]
  - has 2 arguments.
```

Count_Phis3.cpp

Example of Dynamic Cast

```
virtual bool runOnFunction(Function &F) {
  errs() << "Function " << F.getName() << '\n';
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    if (PHINode *PN = dyn_cast<PHINode>(&*I)) {
      errs() << *PN << "\n";
      int numArgs = PN->getNumIncomingValues();
      errs() << "  - has " << numArgs << " parameters\n";
      for (int arg = 0; arg < numArgs; arg++) {
        errs() << "    Argument " << arg << ":\n";
        errs() << "    " << PN->getIncomingBlock(arg)->getName() << ": " <<
          *(PN->getIncomingValue(arg)) << "\n";
      }
    }
  }
  return false;
}
```

1) What do you think is the semantics of a V' = dyn_cast<T>(V)?

2) Can you tell what is this method doing?

Running the Pass with the Dynamic Cast

```
$> clang -c -emit-llvm c.c -o c.bc

$> opt -mem2reg c.bc -o c.rbc

$> opt -load dcc888.dylib -countphis -disable-output c.rbc
```

These are the arguments of the phi-function

```
Function foo
  %sum.0 = phi i32 [ 0, %entry ], [ %sum.1, %for.inc5 ]
  - has 2 parameters
    Argument 0:
    entry: i32 0
    Argument 1:
    for.inc5:   %sum.1 = phi i32 [ %sum.0, %for.body ], [ %add, %for.inc ]
  %c0.0 = phi i32 [ %n, %entry ], [ %dec6, %for.inc5 ]
  - has 2 parameters ...
```

What do these args represent?

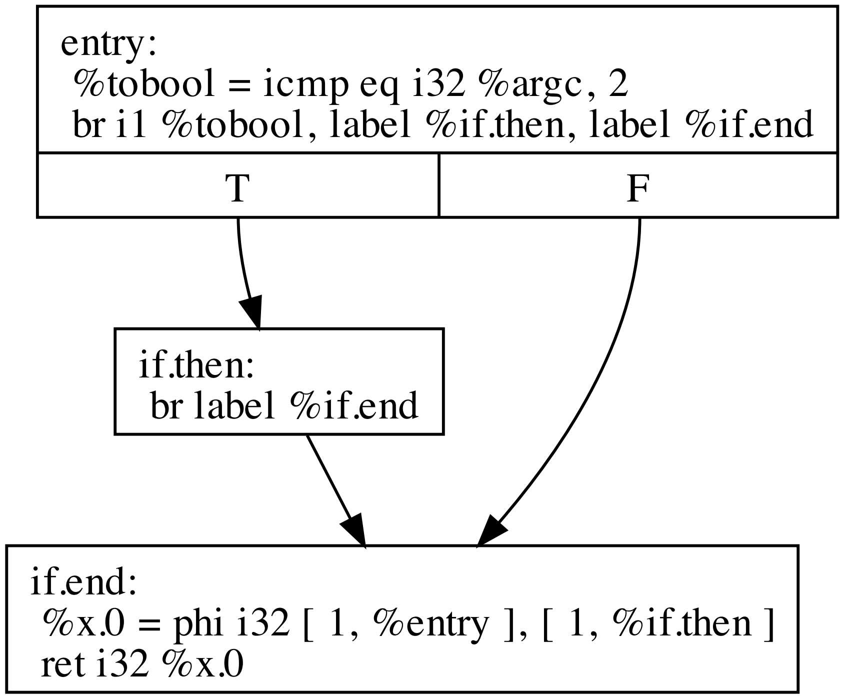
Transforming the Code

- LLVM lets the developer to change the code.
- Usually changes are performed to **optimize** the program.
- There are basically three kinds of transformations that we can do:
  - Add/Remove instructions
  - Add/Remove basic blocks
  - Add/Remove functions

Today we shall see how to play with instructions!

Optimizing Phi-Functions with Constant Args

One of the first optimizations that LLVM does is to replace phi-functions that have equal arguments by the argument itself.



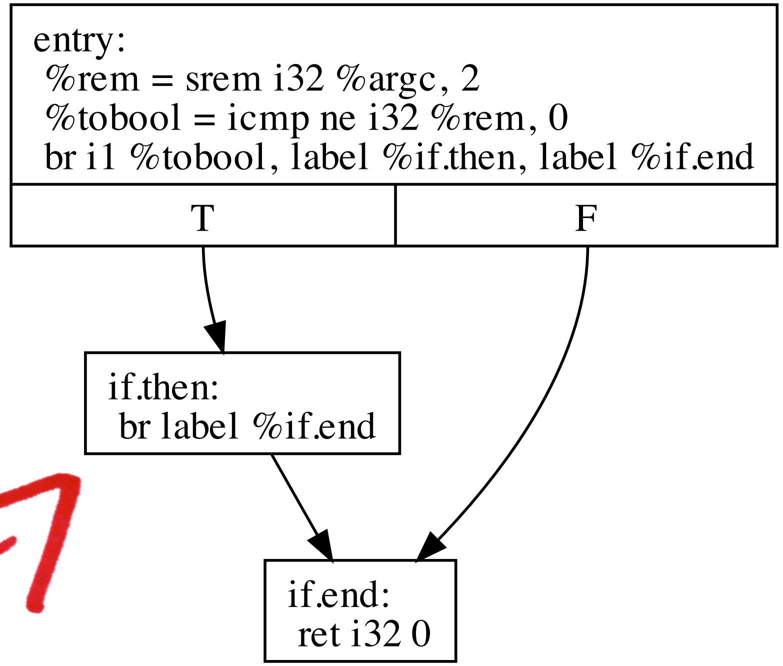
1) How can we have phi-functions having the same value as different arguments? It is not that easy to build such a thing.

2) How can we optimize programs that have phi-functions like these?

Optimizing Phi-Functions with Constant Args

It is really not easy to get a phi-function with constant args in LLVM. For instance, the program below produces the CFG on the right.

```
int main(int argc, char** argv) {
  int x = 0;
  if (argc % 2) {
    x = 0;
  }
  return x;
}
```



1) What does LLVM do with phi-functions that have the same arguments?

2) So, how to get a phi-function with all the arguments the same?

Playing with Bytecodes

We can play with the bytecodes directly. For instance, this program on the right is valid LLVM code, and we can even compile it!

> Does this program on the right has a phi-function with the same two arguments?

```
target triple = "i386-apple-macosx10.5.0"

define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %tobool = icmp eq i32 %argc, 2
  br i1 %tobool, label %if.then, label %if.end

if.then:
  br label %if.end

if.end:
  %x.0 = phi i32 [ 1, %entry ], [ 1, %if.then ]
  ret i32 %x.0
}
```

```
$> clang -c -emit-llvm play.ll -o play.bc
$> opt -view-cfg play.bc
$> clang play.ll ; ./a.out ; echo $?
1
```

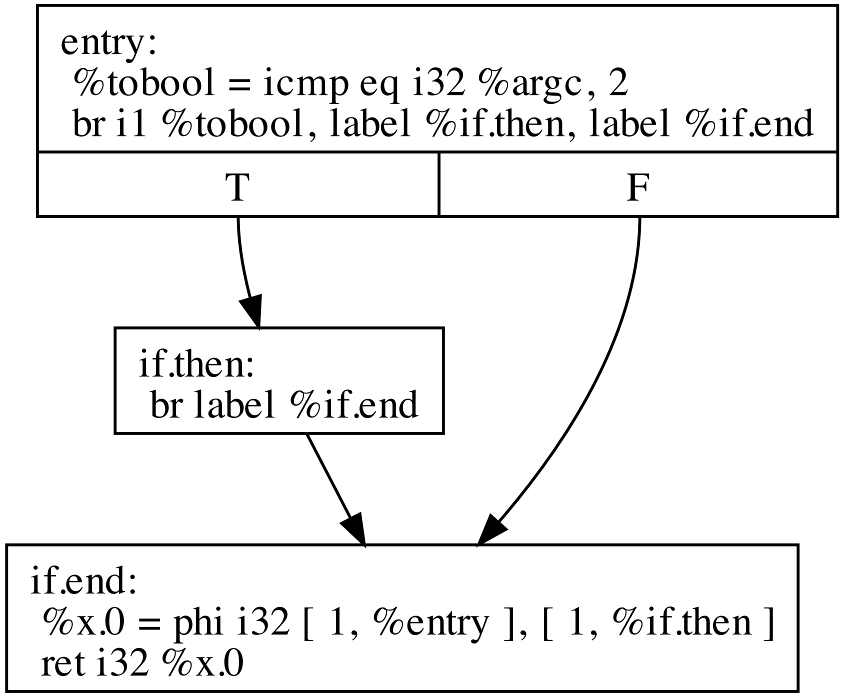
A Silly Example

```
target triple = "i386-apple-macosx10.5.0"

define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %tobool = icmp eq i32 %argc, 2
  br i1 %tobool, label %if.then, label %if.end

if.then:
  br label %if.end

if.end:
  %x.0 = phi i32 [ 1, %entry ], [ 1, %if.then ]
  ret i32 %x.0
}
```



Can you design an optimization that improves this kind of program?

Presto! Here is our program with the bad phi-function.

Writing the Optimizing Pass

```
struct Count_Phis : public FunctionPass {
  static char ID;
  Count_Phis () : FunctionPass(ID) {}
  virtual bool runOnFunction(Function &F) {

    // Collect the instructions that need to be removed

    // Remove the instructions collected previously

    return cutInstruction;
  }
};
```

1) Why can't we remove instructions once we find them in the iterator?
2) What is the return value of the runOnFunction pass?
3) Which value (**cutInstruction**) should we return?

Finding and Removing Instructions

```
virtual bool runOnFunction(Function &F) {
  bool cutInstruction = false;
  errs() << "Function " << F.getName() << '\n';
  SmallVector<PHINode*, 16> Worklist;
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    if (PHINode *PN = dyn_cast<PHINode>(&*I)) {
      if (PN->hasConstantValue()) {
        errs() << *PN << " has constant value.\n";
        // store for later elimination:
        Worklist.push_back(PN);
        cutInstruction = true;
      }
    }
  }
  // Eliminate the uses:
  while (!Worklist.empty()) {
    PHINode* PN = Worklist.pop_back_val();
    PN->replaceAllUsesWith(PN->getIncomingValue(0));
    PN->eraseFromParent();
  }
  return cutInstruction;
}
```

What do you think is **this** data structure good for?

We have to be a bit careful. We cannot change an iterator during the iteration. Removing elements from the iterator would invalidate it. That is why first we collect, and then we process the instructions.

What do you think **this** call, and **this** call do?

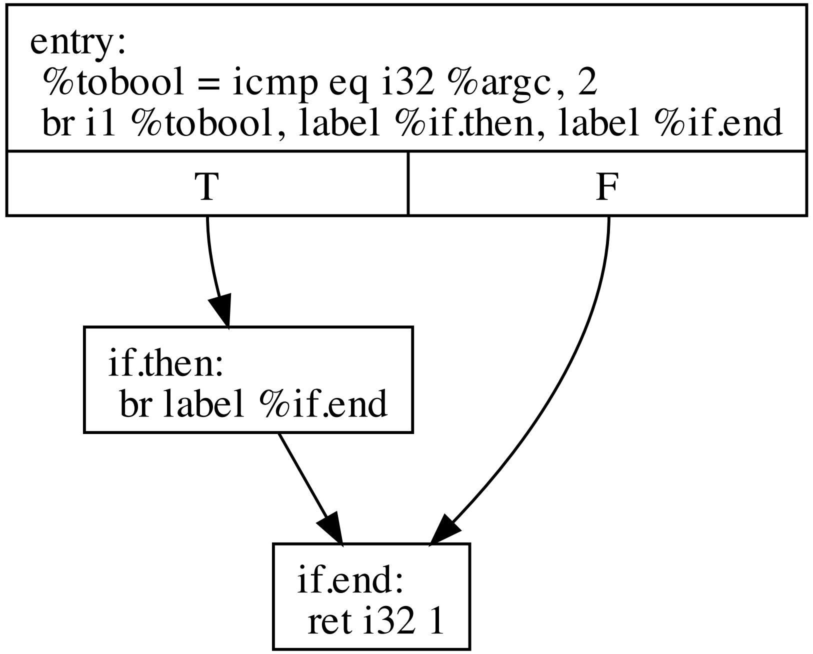
Our Optimizer in Action

```
; ModuleID = '<stdin>'
target triple = "i386-apple-macosx10.5.0"

define i32 @main(i32 %argc, i8** %argv) {
entry:
  %tobool = icmp eq i32 %argc, 2
  br i1 %tobool, label %if.then, label %if.end

if.then:
  br label %if.end

if.end:
  ret i32 1
}
```



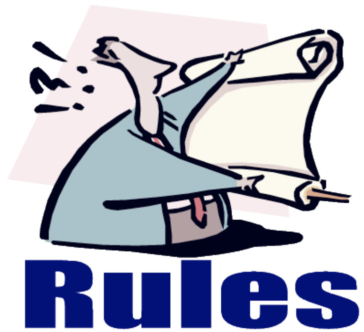
This time we no longer use the disable-output argument. Can you guess why?

```
$> opt -load dcc888.dylib -countphis  play.bc -o play2.bc
$> clang play2.ll ; ./a.out ; echo $?
1
```

Count_Phis4.cpp

Optimizing the Optimizer

```
virtual bool runOnFunction(Function &F) {
  bool cutInstruction = false;
  errs() << "Function " << F.getName() << '\n';
  SmallVector<PHINode*, 16> Worklist;
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    if (PHINode *PN = dyn_cast<PHINode>(&*I)) {
      if (PN->hasConstantValue()) {
        errs() << *PN << " has constant value.\n";
        // store for later elimination:
        Worklist.push_back(PN);
        cutInstruction = true;
      }
    }
  }
  // Eliminate the uses:
  while (!Worklist.empty()) {
    PHINode* PN = Worklist.pop_back_val();
    PN->replaceAllUsesWith(PN->getIncomingValue(0));
    PN->eraseFromParent();
  }
  return cutInstruction;
}
```



We can use some properties of the SSA program representation to improve our code. For instance, we know that phi-functions cannot be in the middle of basic blocks…

1) Why is the last sentence true?

2) How can we use this fact to speedup our optimizer?

Functions and Basic Blocks

```
virtual bool runOnFunction(Function &F) {
  bool cutInstruction = false;
  errs() << "Function " << F.getName() << '\n';
  SmallVector<PHINode*, 16> Worklist;
  for (Function::iterator B = F.begin(), EB = F.end(); B != EB; ++B) {
    for (BasicBlock::iterator I = B->begin(), EI = B->end(); I != EI; ++I) {
      if (PHINode *PN = dyn_cast<PHINode>(I)) {
        if (PN->hasConstantValue()) {
          Worklist.push_back(PN);
          cutInstruction = true;
        }
      } else {
        continue;
      }
    }
  }
  while (!Worklist.empty()) {
    PHINode* PN = Worklist.pop_back_val();
    PN->replaceAllUsesWith(PN->getIncomingValue(0));
    PN->eraseFromParent();
  }
  return cutInstruction;
}
```

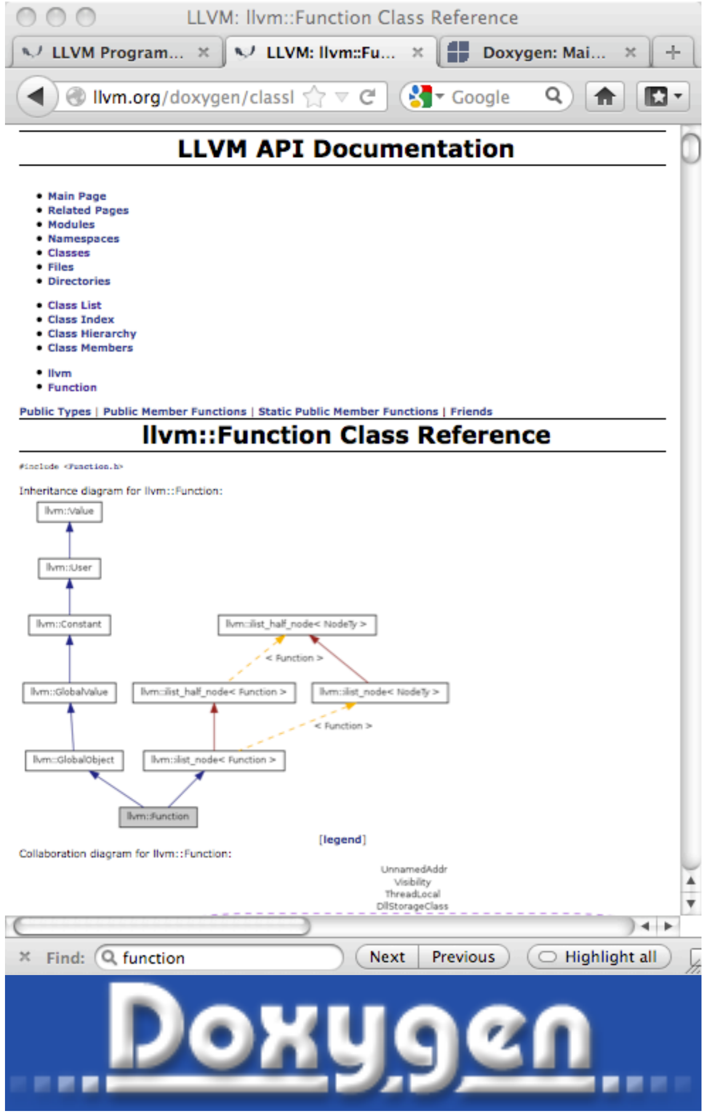
What are the elements stored in each of these iterators?

Remember: all the phi-functions are right in the beginning of the basic-blocks.

```
if.end:
 %x.0 = phi i8 [ %1, %if.then ], [ 0, %entry ]
 %y.0 = phi i8 [ %3, %if.then ], [ 0, %entry ]
 %z.0 = phi i8 [ %5, %if.then ], [ 0, %entry ]
 %w.0 = phi i8 [ %7, %if.then ], [ 0, %entry ]
 %conv = sext i8 %w.0 to i32
 %conv8 = sext i8 %x.0 to i32
 %add = add nsw i32 %conv, %conv8
 %conv9 = sext i8 %y.0 to i32
 %add10 = add nsw i32 %add, %conv9
 %conv11 = sext i8 %z.0 to i32
 %add12 = add nsw i32 %add10, %conv11
 ret i32 %add12
```

Doxygen

- The LLVM API is mostly described in the doxygen page, which is available on-line.
  – http://llvm.org/doxygen/
- *Doxygen* is the de facto standard tool for generating documentation from annotated C++ sources.
- Much can be learnt from the doxygen, and the source files in the LLVM distribution.



Final Remarks

- There are many different ways to use the LLVM API to analyze or optimize programs.
- A good way to learn about these tools is to read the LLVM source code.
  - Remember, `grep` is your friend:

```
$> cd llvm/lib/Analysis
$~Programs/llvm/lib/Analysis> grep -r inst_iterator  *

AliasAnalysisEvaluator.cpp:  for (inst_iterator I =
inst_begin(F), E = inst_end(F); I != E; ++I) {

AliasSetTracker.cpp:      for (inst_iterator I =
 inst_begin(F), E = inst_end(F); I != E; ++I)

...
```