How an Optimizing Compiler Works

Rewriting code with simple data structures and algorithms

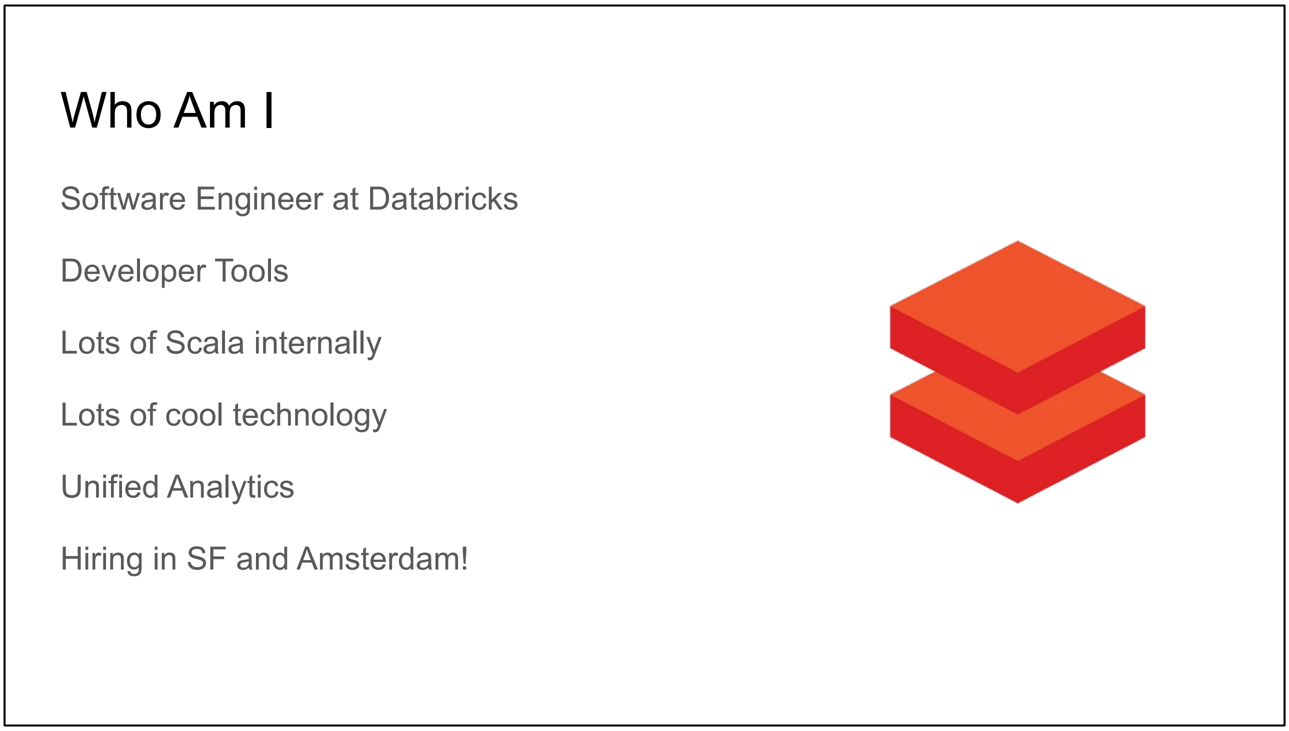
Li Haoyi, Scaladays 12 June 2019

Hello everyone. My name is Haoyi, and this talk is going to be about How an Optimizing Compiler Works

This topic is a recent interest of mine: how could we write an optimizing compiler remove the “Scala Tax” that stops idiomatic Scala programs from being as efficient as their Java equivalents?

There are a number of optimizers in the Scala ecosystem: one in each of the Scala’s JVM, Javascript, and Native-LLVM backends.

For most people, an optimizer is a black box: code goes in one end, and faster code comes out the other. The goal of this talk is to open up that black box, and understand how simple data structures and algorithms are enough to perform useful optimizations on our programs.



Who Am I

Software Engineer at Databricks

Developer Tools

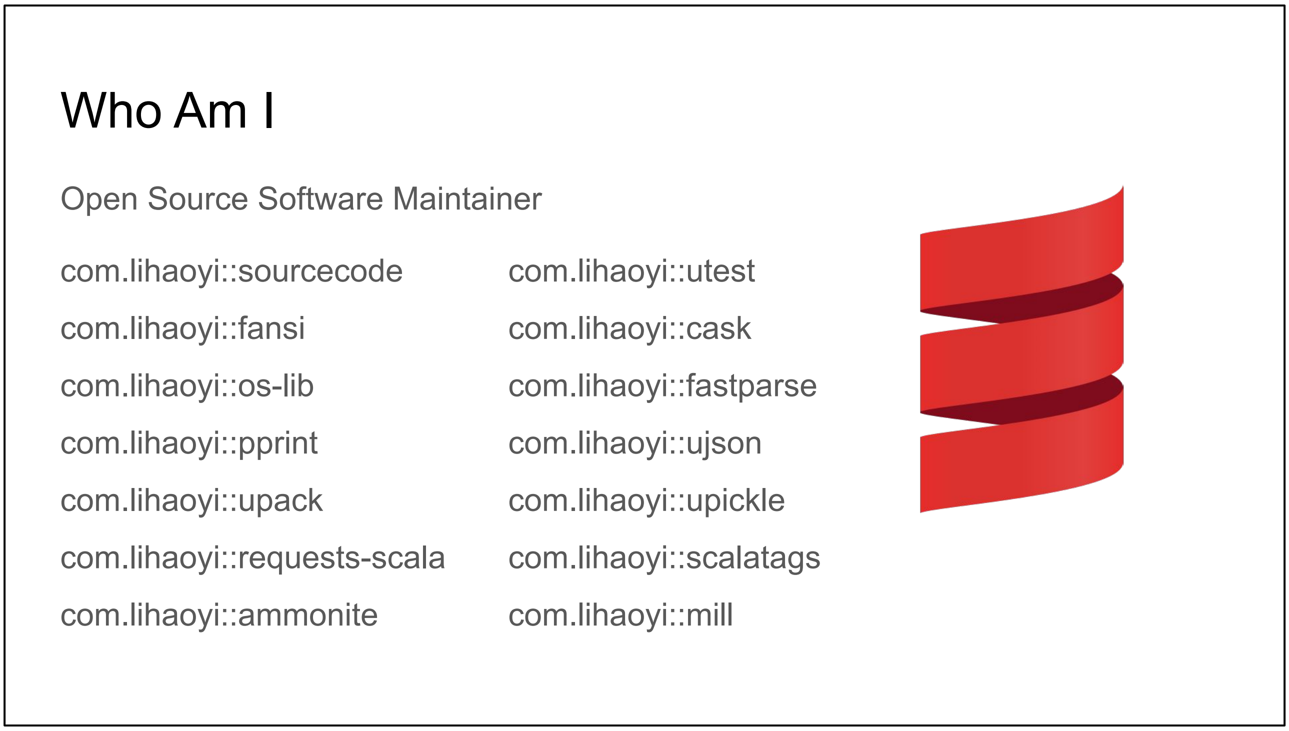
Lots of Scala internally

Lots of cool technology

Unified Analytics

Hiring in SF and Amsterdam!

About myself, I’m a software engineer at Databricks working on developer tools. We have a lot of Scala internally, and a lot of cool technology that we’re building to create the unified analytics platform of the future. Databricks isn’t your run-of-the-mill CRUD app, so if you’re interested in Scala and hard technology problems, we’re hiring in our offices in San Francisco and Amsterdam, so come talk to me after!



Who Am I

Open Source Software Maintainer

com.lihaoyi::sourcecode

com.lihaoyi::fansi

com.lihaoyi::os-lib

com.lihaoyi::pprint

com.lihaoyi::upack

com.lihaoyi::requests-scala

com.lihaoyi::ammonite

com.lihaoyi::utest

com.lihaoyi::cask

com.lihaoyi::fastparse

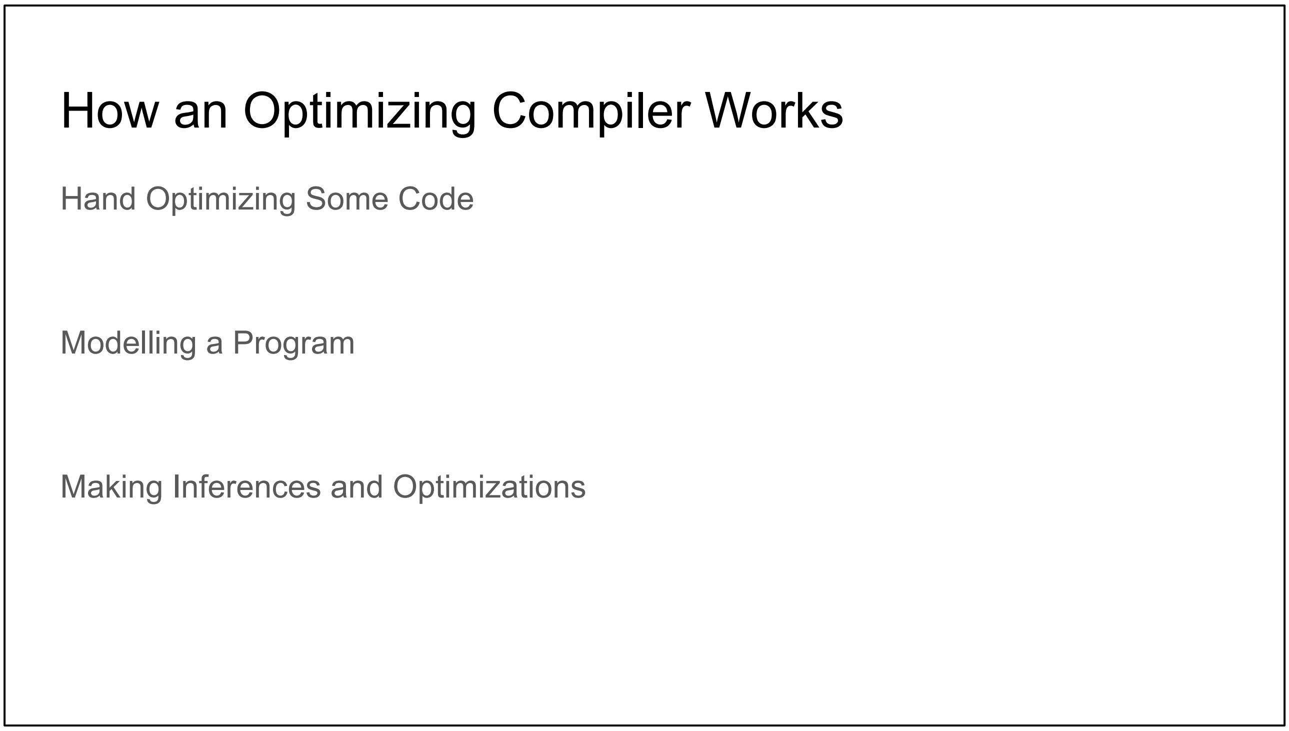
com.lihaoyi::ujson

com.lihaoyi::upickle

com.lihaoyi::scalatags

com.lihaoyi::mill

I also maintain quite an extensive suite of open source Scala libraries and tools.



How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

Making Inferences and Optimizations

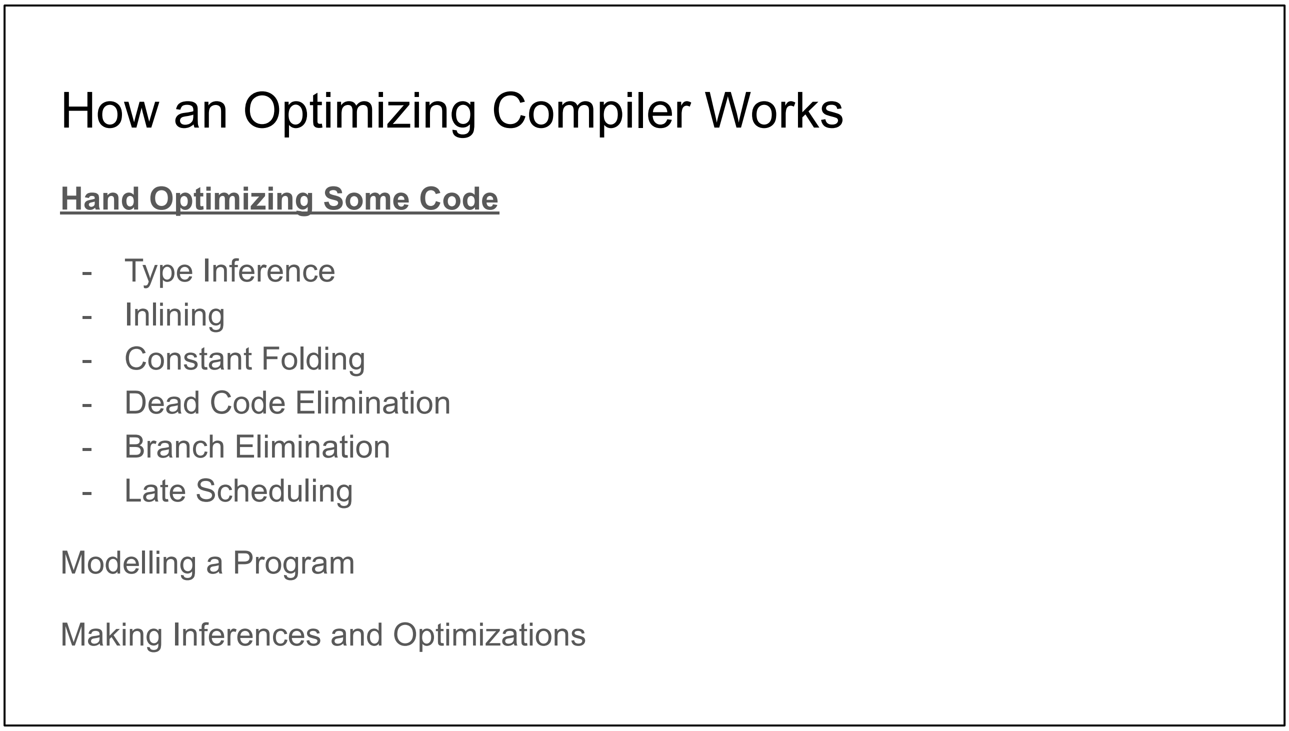
But this talk is not about Databricks, or my open source work, but about optimizing compilation.

We will go through three main sections.

First, we will optimize some sample code by hand

Next, the different ways in which your optimizer can model programs in memory

Lastly, how to perform the same optimizations we had done by-hand, but automatically



How an Optimizing Compiler Works

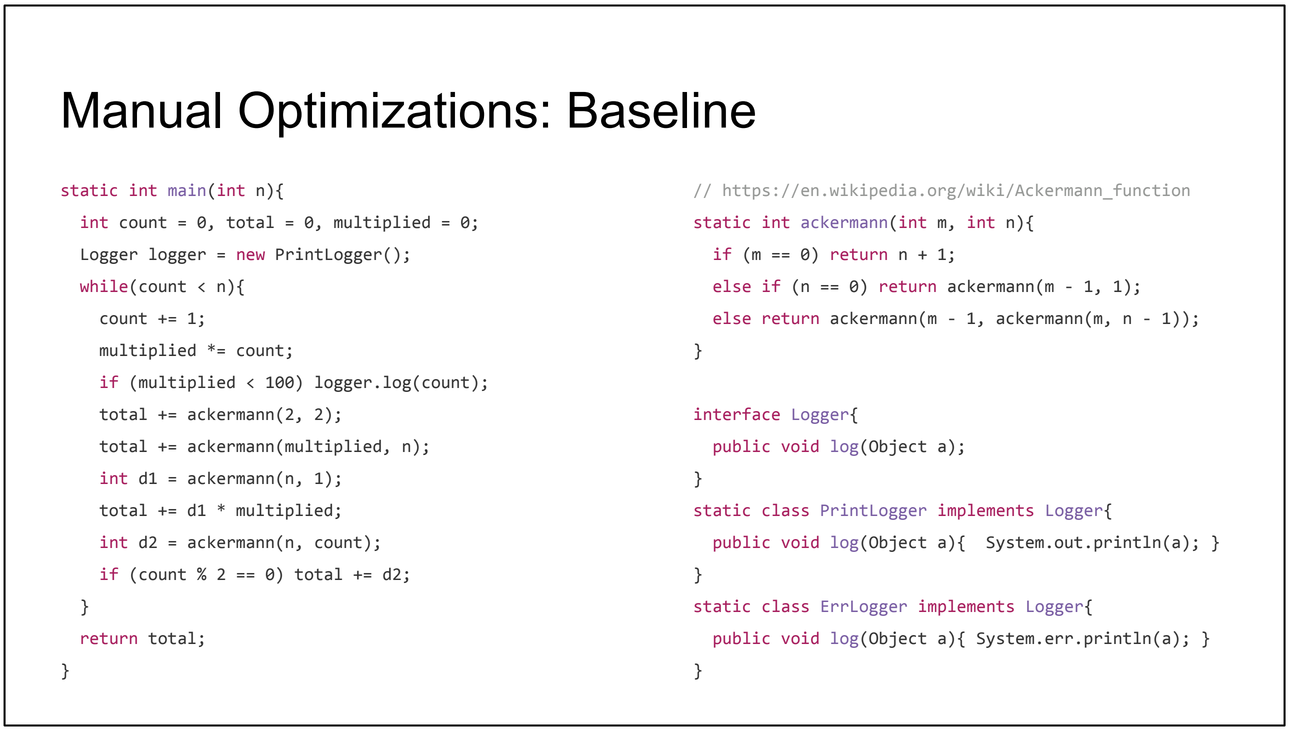
Hand Optimizing Some Code

- Type Inference
- Inlining
- Constant Folding
- Dead Code Elimination
- Branch Elimination
- Late Scheduling

Modelling a Program

Making Inferences and Optimizations

To begin with, let’s walk through some simple optimizations.



Manual Optimizations: Baseline

```java
static int main(int n){
  int count = 0, total = 0, multiplied = 0;
  Logger logger = new PrintLogger();
  while(count < n){
    count += 1;
    multiplied *= count;
    if (multiplied < 100) logger.log(count);
    total += ackermann(2, 2);
    total += ackermann(multiplied, n);
    int d1 = ackermann(n, 1);
    total += d1 * multiplied;
    int d2 = ackermann(n, count);
    if (count % 2 == 0) total += d2;
  }
  return total;
}
```

```java
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}

interface Logger{
  public void log(Object a);
}
static class PrintLogger implements Logger{
  public void log(Object a){  System.out.println(a); }
}
static class ErrLogger implements Logger{
  public void log(Object a){ System.err.println(a); }
}
```

This is a small snippet of Java source code. Java is a well-known, relatively simple language. Scala code usually translates straightforwardly to Java.

This **main** function takes an argument, performs some computation, function calls, and logging, before returning a value. Assume that this code is run in isolation, so what you see is all there is.

Let’s say our job is to optimize this code, ignoring whether what it does is useful or not. What can we do to make this code smaller, simpler, and faster?

Manual Optimizations: Type Inference

```
static int main(int n){
  int count = 0, total = 0, multiplied = 0;
- Logger logger = new PrintLogger();
+ PrintLogger logger = new PrintLogger();
  while(count < n){
    count += 1;
    multiplied *= count;
    if (multiplied < 100) logger.log(count);
    total += ackermann(2, 2);
    total += ackermann(multiplied, n);
    int d1 = ackermann(n, 1);
    total += d1 * multiplied;
    int d2 = ackermann(n, count);
    if (count % 2 == 0) total += d2;
  }
  return total;
}
```

```
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}

interface Logger{
  public void log(Object a);
}
static class PrintLogger implements Logger{
  public void log(Object a){  System.out.println(a); }
}
static class ErrLogger implements Logger{
  public void log(Object a){ System.err.println(a); }
}
```

First, we can see that the **logger** variable is typed less specifically than it could be. We know it is a concrete **PrintLogger**, not just any **Logger**

Manual Optimizations: Inlining

```
static int main(int n){
  int count = 0, total = 0, multiplied = 0;
  PrintLogger logger = new PrintLogger();
  while(count < n){
    count += 1;
    multiplied *= count;
-   if (multiplied < 100) logger.log(count);
+   if (multiplied < 100) System.out.println(count);
    total += ackermann(2, 2);
    total += ackermann(multiplied, n);
    int d1 = ackermann(n, 1);
    total += d1 * multiplied;
    int d2 = ackermann(n, count);
    if (count % 2 == 0) total += d2;
  }
  return total;
}
```

```
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}

interface Logger{
  public void log(Object a);
}
static class PrintLogger implements Logger{
  public void log(Object a){  System.out.println(a); }
}
static class ErrLogger implements Logger{
  public void log(Object a){ System.err.println(a); }
}
```

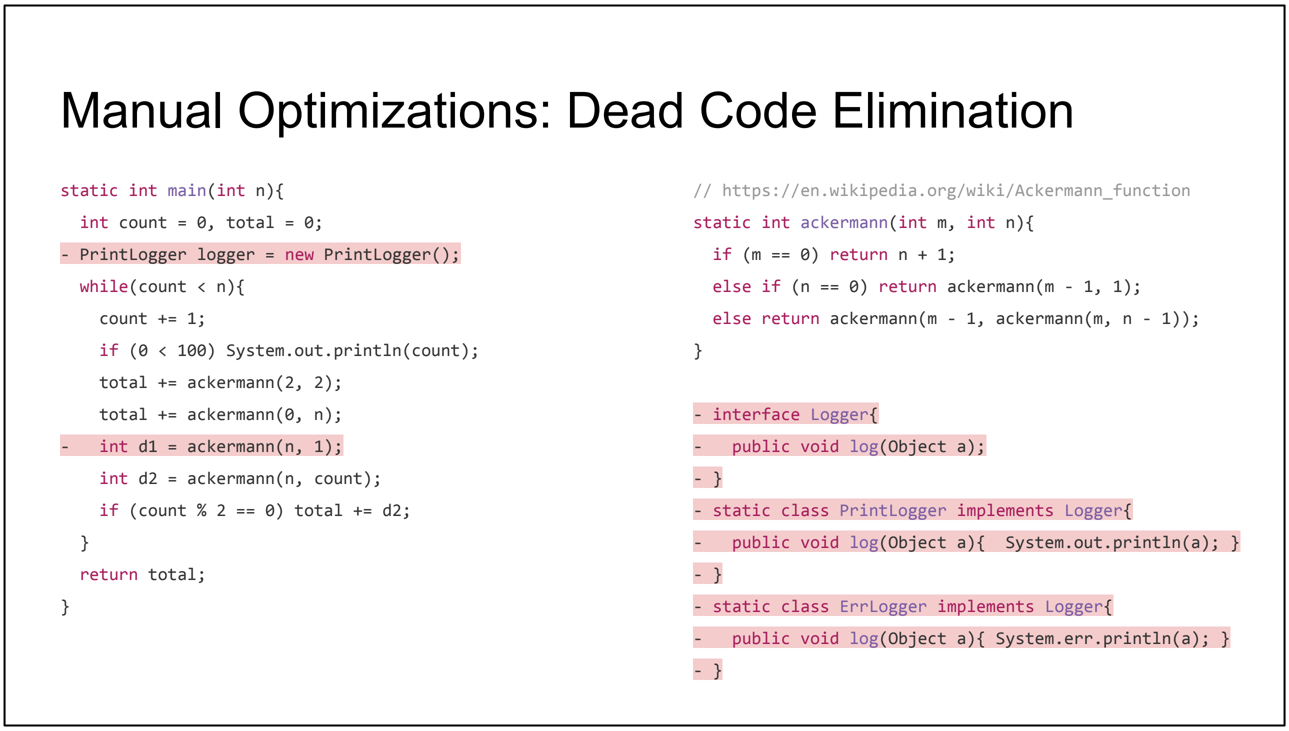
We can then infer the call to **logger.log** can only go to one implementation, **System.out.println**, and can simply inline it.

Manual Optimizations: Constant Folding

```
static int main(int n){
- int count = 0, total = 0, multiplied = 0;
+ int count = 0, total = 0;
  PrintLogger logger = new PrintLogger();
  while(count < n){
    count += 1;
-   multiplied *= count;
-   if (multiplied < 100) System.out.println(count);
+   if (0 < 100) System.out.println(count);
    total += ackermann(2, 2);
-   total += ackermann(multiplied, n);
+   total += ackermann(0, n);
    int d1 = ackermann(n, 1);
-   total += d1 * multiplied;
    int d2 = ackermann(n, count);
    if (count % 2 == 0) total += d2;
  }
```

```
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}

interface Logger{
  public void log(Object a);
}
static class PrintLogger implements Logger{
  public void log(Object a){  System.out.println(a); }
}
static class ErrLogger implements Logger{
  public void log(Object a){ System.err.println(a); }
}
```

Next, the **multiplied** variable starts off **0**, and since it only gets multiplied, it remains **0** throughout. We can thus discard the variable and just put **0** everywhere it is used.



Manual Optimizations: Dead Code Elimination

```
static int main(int n){
  int count = 0, total = 0;
- PrintLogger logger = new PrintLogger();
  while(count < n){
    count += 1;
    if (0 < 100) System.out.println(count);
    total += ackermann(2, 2);
    total += ackermann(0, n);
-   int d1 = ackermann(n, 1);
    int d2 = ackermann(n, count);
    if (count % 2 == 0) total += d2;
  }
  return total;
}
```

```
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}

- interface Logger{
-   public void log(Object a);
- }
- static class PrintLogger implements Logger{
-   public void log(Object a){  System.out.println(a); }
- }
- static class ErrLogger implements Logger{
-   public void log(Object a){ System.err.println(a); }
- }
```

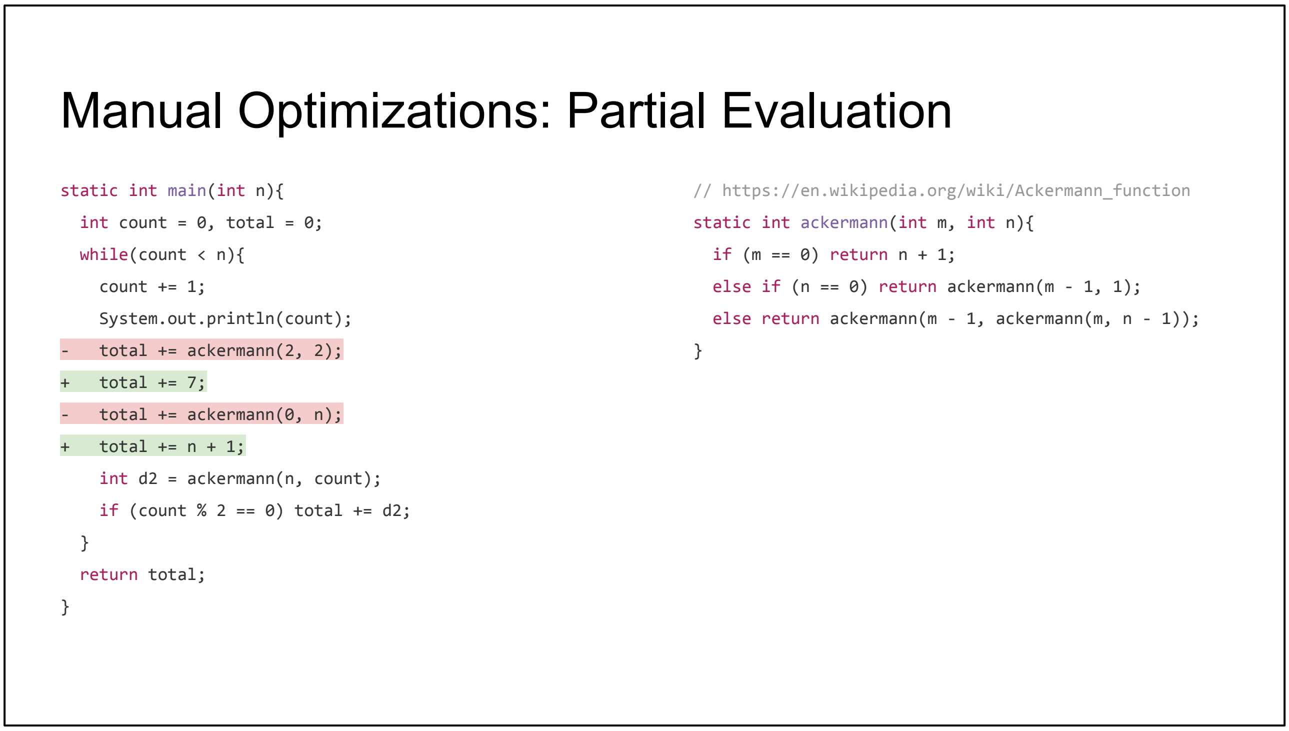
The previous optimizations mean that the **d1** variable is now completely unused, as is the **logger** variable, and all the **Logger** classes, so they can be removed.

Manual Optimizations: Branch Elimination

```
static int main(int n){
  int count = 0, total = 0;
  while(count < n){
    count += 1;
-   if (0 < 100) System.out.println(count);
+   System.out.println(count);
    total += ackermann(2, 2);
    total += ackermann(0, n);
    int d2 = ackermann(n, count);
    if (count % 2 == 0) total += d2;
  }
  return total;
}
```

```
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

The **if (0 < 100)** always returns true, so it can be removed.

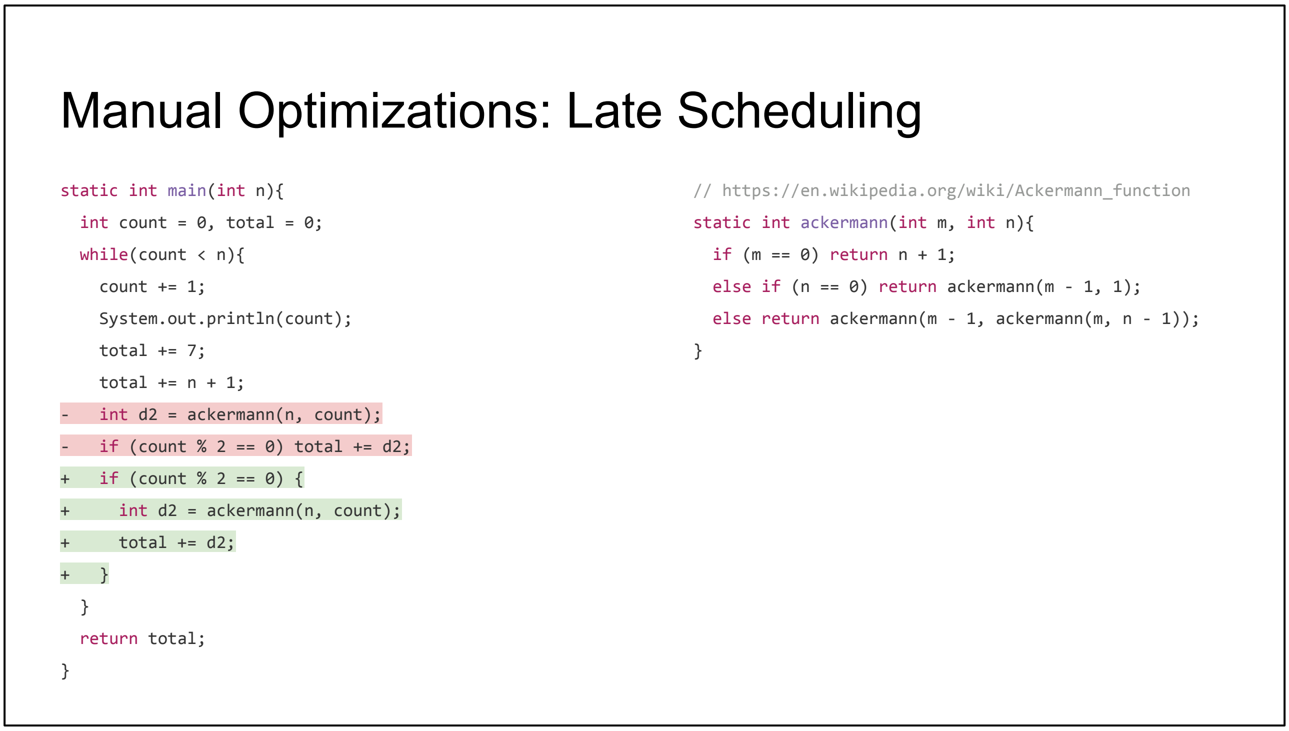


Manual Optimizations: Partial Evaluation

```
static int main(int n){
  int count = 0, total = 0;
  while(count < n){
    count += 1;
    System.out.println(count);
-   total += ackermann(2, 2);
+   total += 7;
-   total += ackermann(0, n);
+   total += n + 1;
    int d2 = ackermann(n, count);
    if (count % 2 == 0) total += d2;
  }
  return total;
}
```

```
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

Two of the calls to **ackermann** have fixed parameters: the first takes two constant **2**s as arguments and always returns the constant **7**, while the second taking a constant **0** and an unknown integer **n** will always return **n + 1**

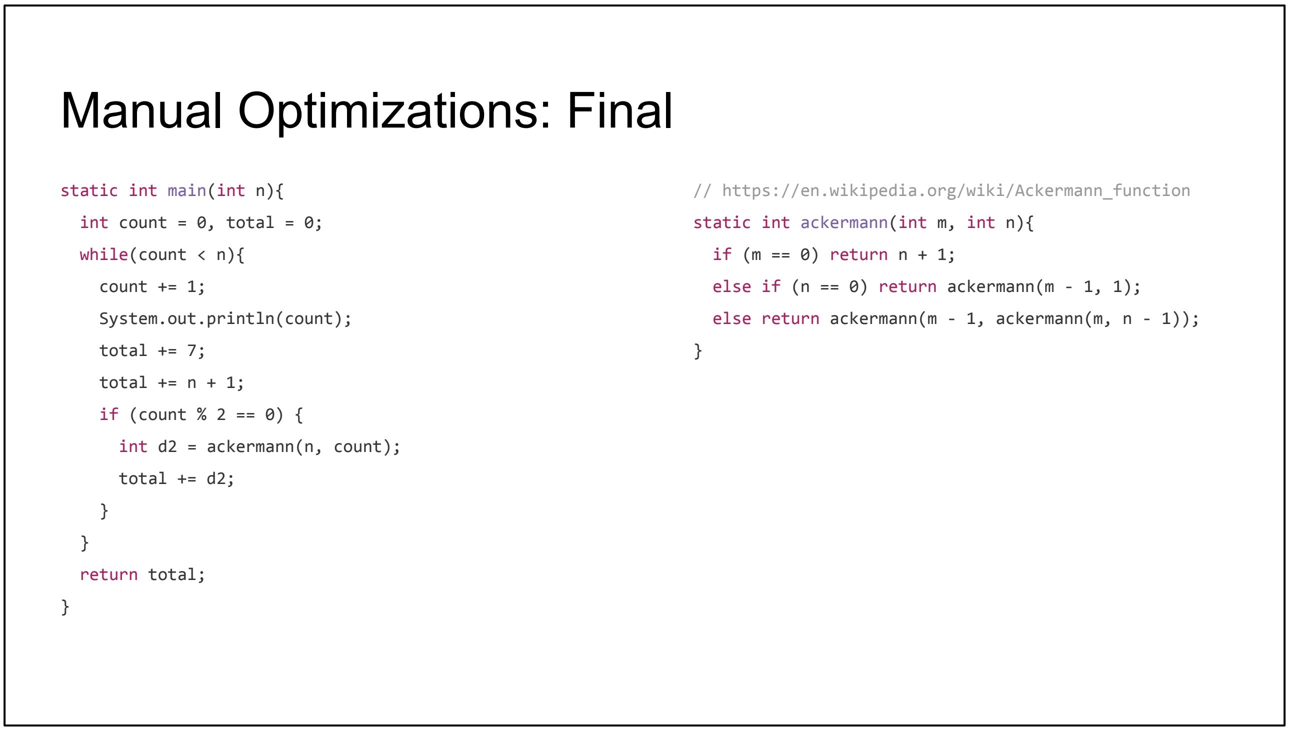


Manual Optimizations: Late Scheduling

```
static int main(int n){
  int count = 0, total = 0;
  while(count < n){
    count += 1;
    System.out.println(count);
    total += 7;
    total += n + 1;
-   int d2 = ackermann(n, count);
-   if (count % 2 == 0) total += d2;
+   if (count % 2 == 0) {
+     int d2 = ackermann(n, count);
+     total += d2;
+   }
  }
  return total;
}
```

```
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

The remaining call to **ackermann** taking two unknown Integers **n** and **count** is only used in the body of the remaining **if**. We can thus move it inside the conditional block so it is only computed when necessary.



Manual Optimizations: Final

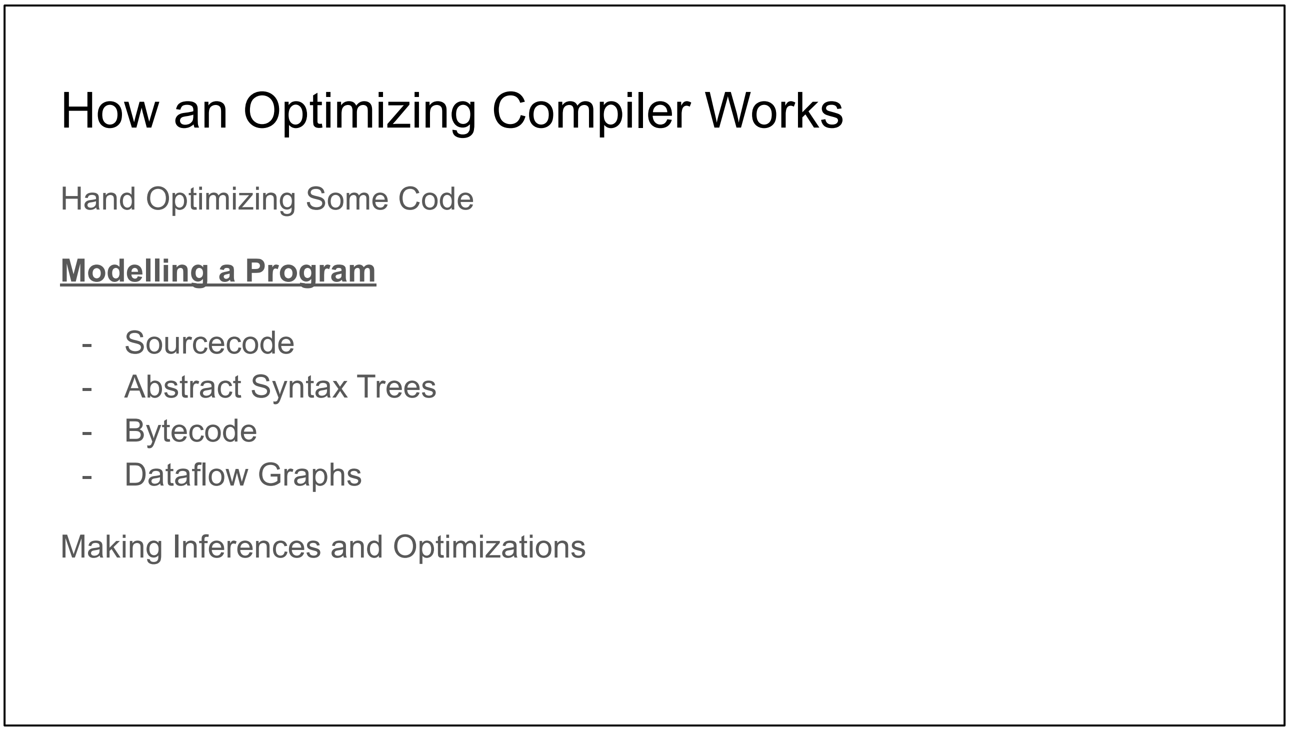
```
static int main(int n){
  int count = 0, total = 0;
  while(count < n){
    count += 1;
    System.out.println(count);
    total += 7;
    total += n + 1;
    if (count % 2 == 0) {
      int d2 = ackermann(n, count);
      total += d2;
    }
  }
  return total;
}
```

```
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

The final simplified code looks like this. We have taken the somewhat convoluted original program, and created a simplified version that does the same thing, in less code, and probably faster. Optimizations often make your program both faster and simpler as layers of indirection and redundancy are stripped away to reveal the core logic of your code.

Automated Optimizations

These simplifications are pretty mechanical, and should be doable automatically by an optimizing compiler. Here’s a demo.



How an Optimizing Compiler Works

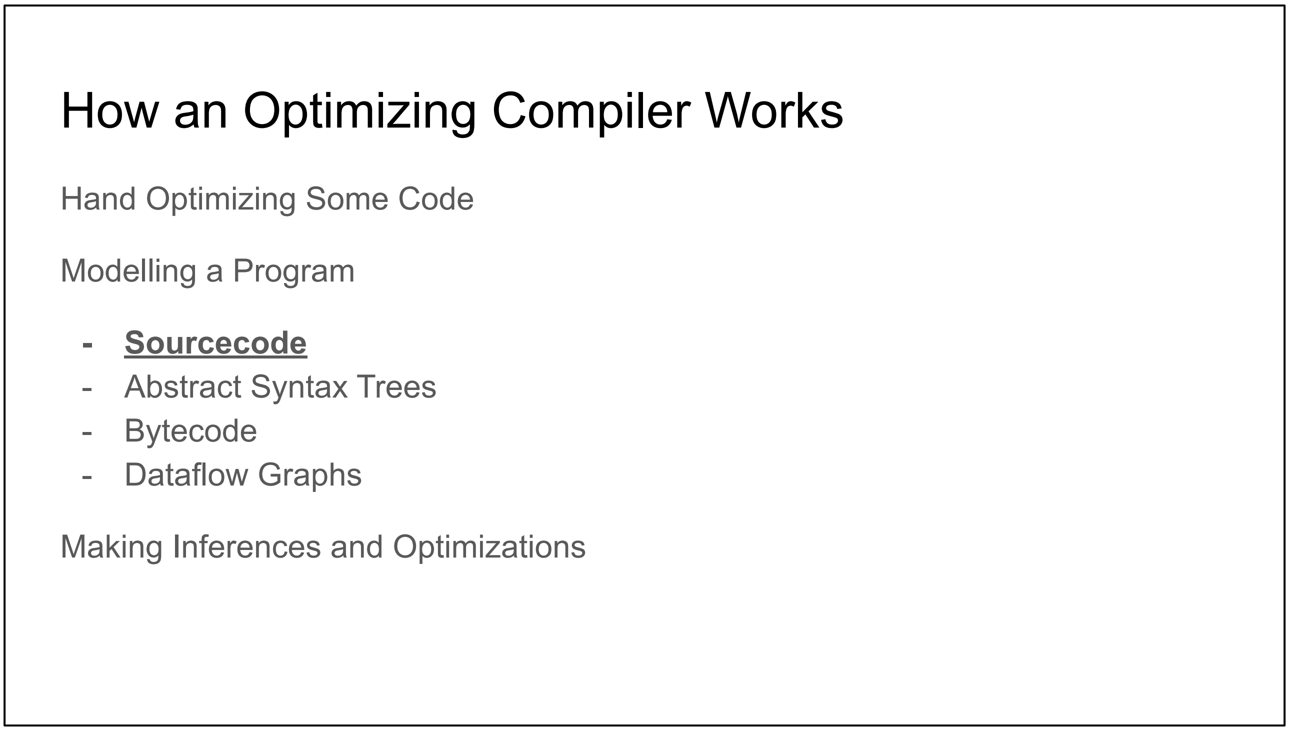
Hand Optimizing Some Code

Modelling a Program

- Sourcecode
- Abstract Syntax Trees
- Bytecode
- Dataflow Graphs

Making Inferences and Optimizations

We'll now take a look at how an optimizer like the one I just demoed can model your program in memory, before going into the algorithms for doing inference and the optimizations themselves.



How an Optimizing Compiler Works

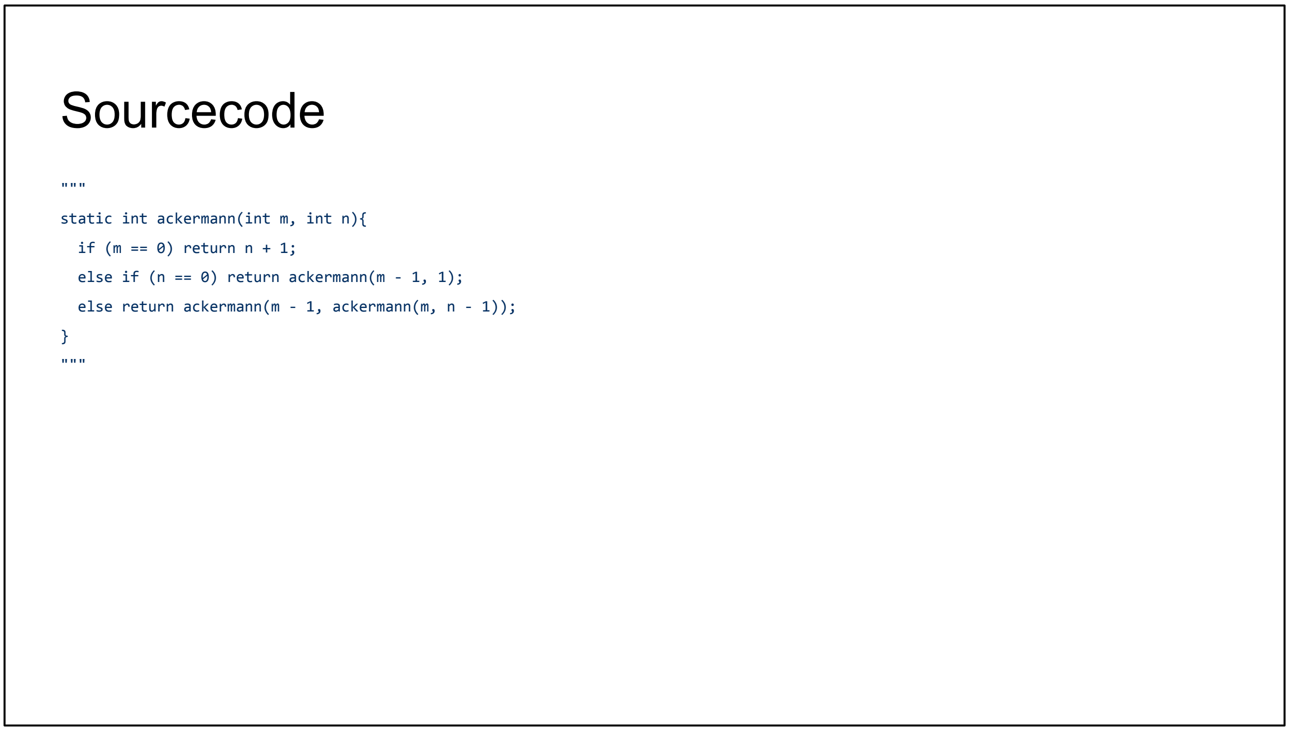
Hand Optimizing Some Code

Modelling a Program

- **Sourcecode**
- Abstract Syntax Trees
- Bytecode
- Dataflow Graphs

Making Inferences and Optimizations

To begin with, let’s consider source code: your program as a linear **String** of characters



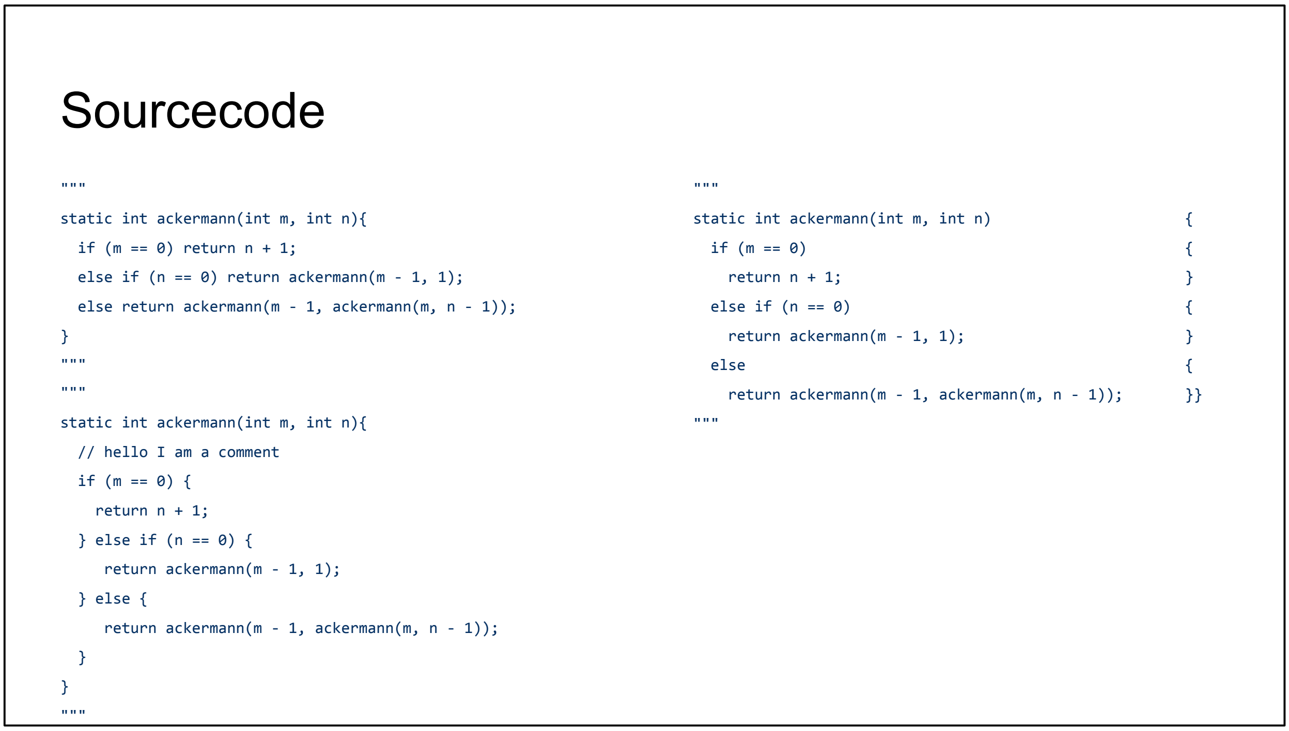
Sourcecode

```
"""
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
"""
```

Here is the **ackermann** function we saw earlier, as a string. As a string, it is human readable, but it has other less-convenient properties:

- It contains all sorts of naming and formatting details that are important to a user but meaningless to the computer

- There are many more invalid sourcecode strings than there are valid sourcecode strings



Sourcecode

```
"""
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
"""
"""
static int ackermann(int m, int n){
  // hello I am a comment
  if (m == 0) {
    return n + 1;
  } else if (n == 0) {
     return ackermann(m - 1, 1);
  } else {
     return ackermann(m - 1, ackermann(m, n - 1));
  }
}
"""
```

```
"""
static int ackermann(int m, int n)                       {
  if (m == 0)                                            {
    return n + 1;                                        }
  else if (n == 0)                                       {
    return ackermann(m - 1, 1);                          }
  else                                                   {
    return ackermann(m - 1, ackermann(m, n - 1));        }}
"""
```

This means that if you are trying to use string-based pattern matching to *analyze* your program, e.g. using Regexes, you need to be able to handle a wide range of possible formattings and layouts that mean the same thing.

And if you are using string-manipulation to *transform* your program, it’s very easy to accidentally generate total gibberish

This makes it difficult to work with programs as strings. Not impossible, but is generally limited to tools which can do their work under human supervision, e.g. Facebook’s CodeMod refactoring tool.

How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

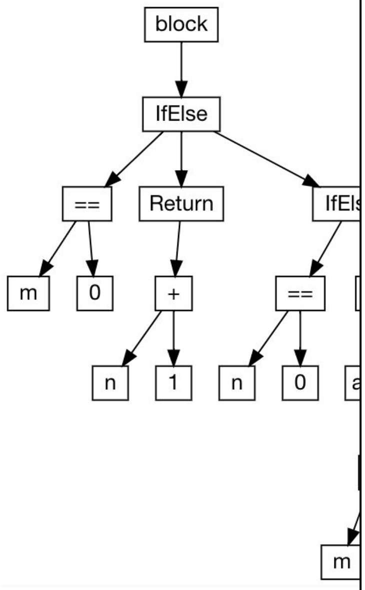
- Sourcecode
- **Abstract Syntax Trees**
- Bytecode
- Dataflow Graphs

Making Inferences and Optimizations

Another way we commonly model programs is as abstract syntax trees, or ASTs

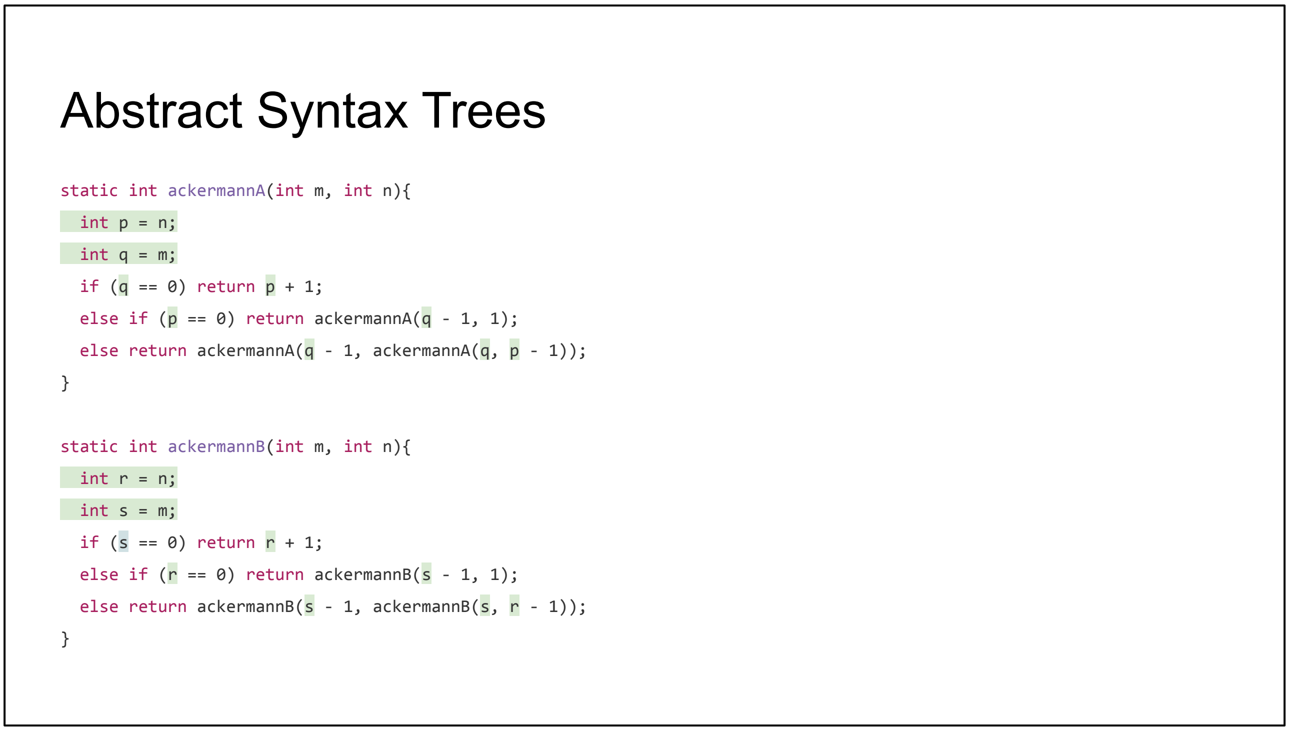
Abstract Syntax Trees

```
IfElse(
    cond = BinOp(Ident("m"), "==", Literal(0)),
    then = Return(BinOp(Ident("n"), "+", Literal(1)),
    else = IfElse(
        cond = BinOp(Ident("n"), "==", Literal(0)),
        then = Return(Call("ackermann", BinOp(Ident("m"), "-", Literal(1)), Literal(1)),
        else = Return(
            Call(
                "ackermann",
                BinOp(Ident("m"), "-", Literal(1)),
                Call("ackermann", Ident("m"), BinOp(Ident("n"), "-", Literal(1)))
            )
        )
    )
)
```



ASTs are a simplified representation of the program, as a tree, built directly from parsing the source code

Details like formatting, whitespace and comments are discarded, leaving only the structure behind. This makes it easier to work with ASTs than strings, and they are a common in many compilers and tools.



Abstract Syntax Trees

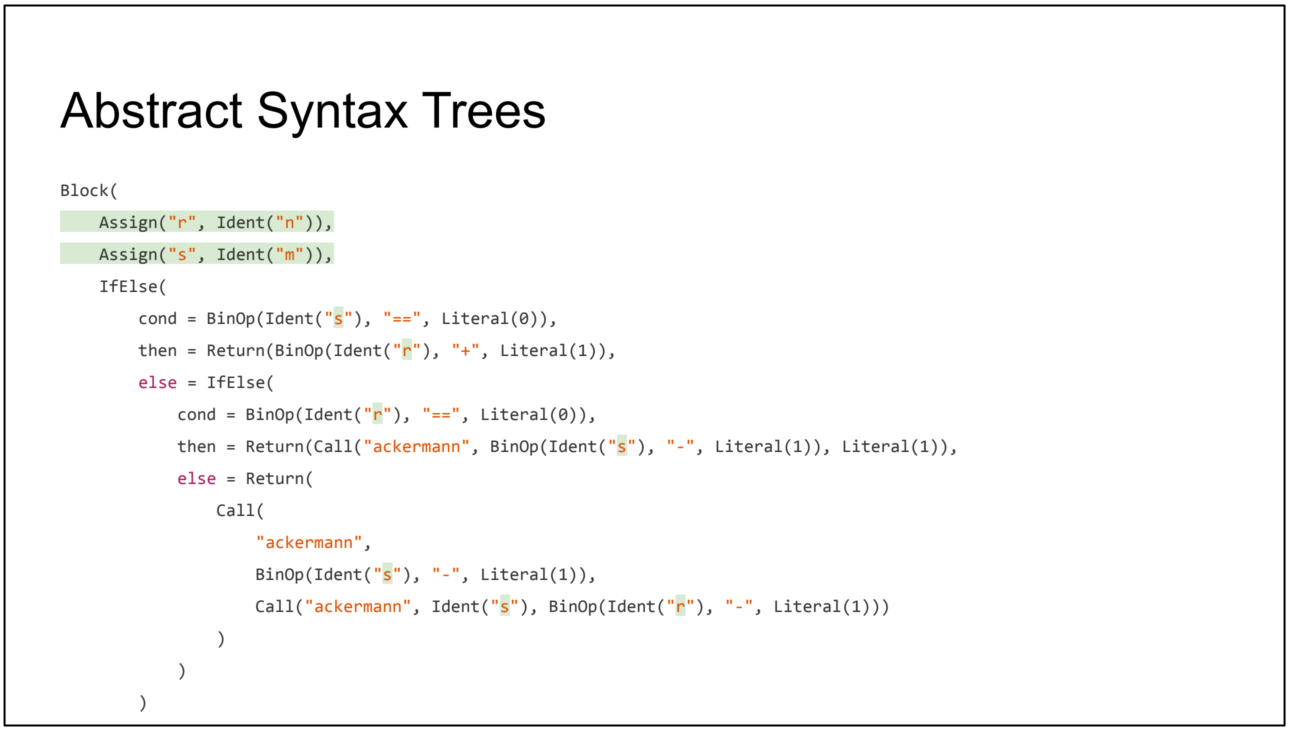
```
static int ackermannA(int m, int n){
  int p = n;
  int q = m;
  if (q == 0) return p + 1;
  else if (p == 0) return ackermannA(q - 1, 1);
  else return ackermannA(q - 1, ackermannA(q, p - 1));
}

static int ackermannB(int m, int n){
  int r = n;
  int s = m;
  if (s == 0) return r + 1;
  else if (r == 0) return ackermannB(s - 1, 1);
  else return ackermannB(s - 1, ackermannB(s, r - 1));
}
```

However, ASTs still contain some amount of irrelevant information. Consider these two variants on **ackermann** that differ from the original by assigning the parameters to locals, and differ from each other in the names chosen for those locals.

Abstract Syntax Trees

```
Block(
    Assign("p", Ident("n")),
    Assign("q", Ident("m")),
    IfElse(
        cond = BinOp(Ident("q"), "==", Literal(0)),
        then = Return(BinOp(Ident("p"), "+", Literal(1)),
        else = IfElse(
            cond = BinOp(Ident("p"), "==", Literal(0)),
            then = Return(Call("ackermann", BinOp(Ident("q"), "-", Literal(1)), Literal(1)),
            else = Return(
                Call(
                    "ackermann",
                    BinOp(Ident("q"), "-", Literal(1)),
                    Call("ackermann", Ident("q"), BinOp(Ident("p"), "-", Literal(1)))
                )
            )
        )
```

While all three functions are semantically identical, they have different ASTs

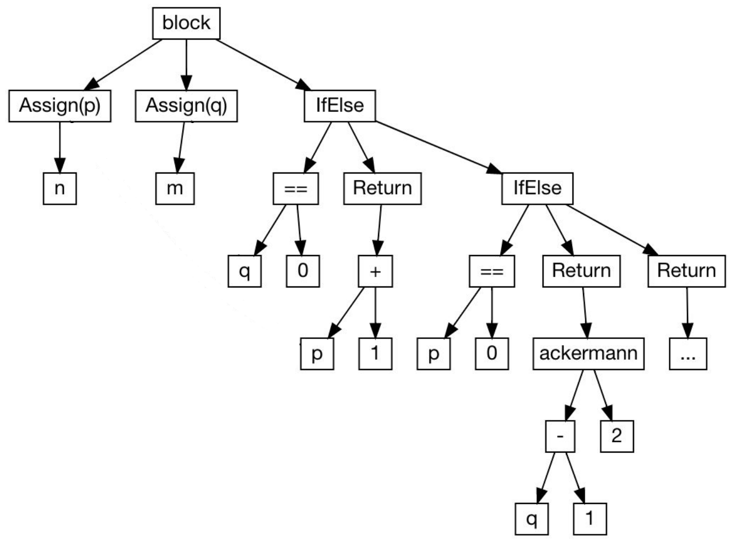


Abstract Syntax Trees

```
Block(
    Assign("r", Ident("n")),
    Assign("s", Ident("m")),
    IfElse(
        cond = BinOp(Ident("s"), "==", Literal(0)),
        then = Return(BinOp(Ident("r"), "+", Literal(1)),
        else = IfElse(
            cond = BinOp(Ident("r"), "==", Literal(0)),
            then = Return(Call("ackermann", BinOp(Ident("s"), "-", Literal(1)), Literal(1)),
            else = Return(
                Call(
                    "ackermann",
                    BinOp(Ident("s"), "-", Literal(1)),
                    Call("ackermann", Ident("s"), BinOp(Ident("r"), "-", Literal(1)))
                )
            )
        )
```

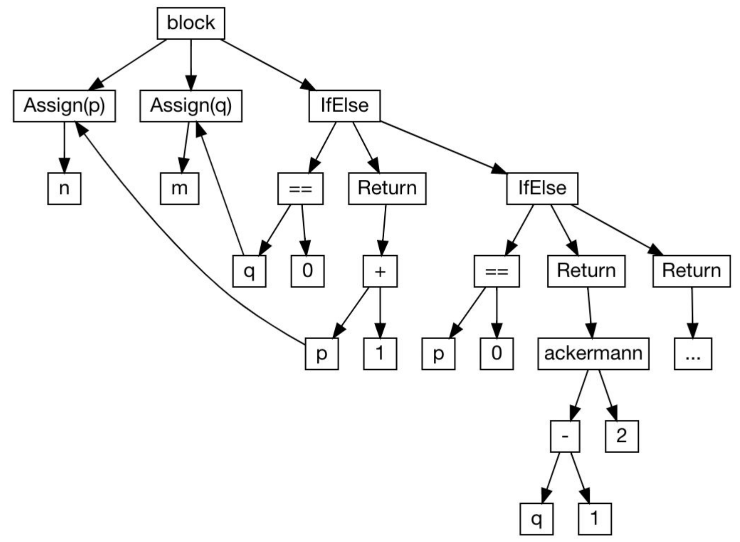
The key here is that while ASTs are structured as trees, some nodes semantically do not behave as trees: if you want to know something about **Ident("r")**, you cannot just look at its contents, but need to go hunt down the **Assign("r")** node somewhere else.

Abstract Syntax Trees

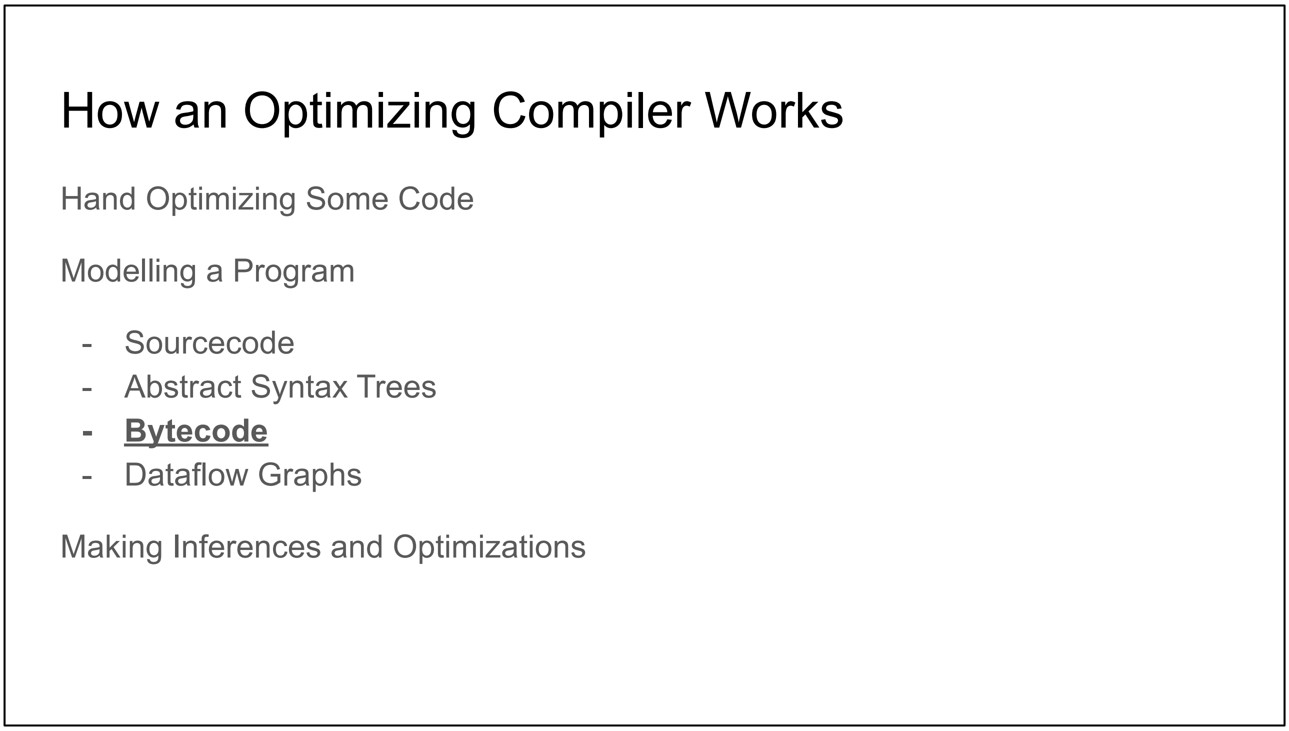


In effect, these **Ident**/**Assign** pairs serve as additional edges in the AST, which are just as important as those in the original tree structure

Abstract Syntax Trees



Our program is in fact a directed graph structure, possibly cyclic, embedded into a tree structure using these **Ident**/**Assign** pairs. This is an idea that we will come back to later.



How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

- Sourcecode
- Abstract Syntax Trees
- **Bytecode**
- Dataflow Graphs

Making Inferences and Optimizations

The next kind of program representation we will look at is Java Bytecode

```
BYTECODE

 0: iload_0
 1: ifne          8
 4: iload_1
 5: iconst_1
 6: iadd
 7: ireturn
 8: iload_1
 9: ifne          20
12: iload_0
13: iconst_1
14: isub
15: iconst_1
16: invokestatic ackermann:(II)I
19: ireturn
20: iload_0
21: iconst_1
22: isub
23: iload_0
24: iload_1
25: iconst_1
26: isub
27: invokestatic ackermann:(II)I
30: invokestatic ackermann:(II)I
33: ireturn
```

```java
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

A Java source program is compiled to a linear list of instructions called bytecode. Here, we can see the **ackermann** function on the right compiled to a linear bytecode on the left

```
BYTECODE                                LOCALS          STACK
                                       |a0|a1|         |
 0: iload_0                            |a0|a1|         |a0|
 1: ifne          8                    |a0|a1|         |
 4: iload_1                            |a0|a1|         |a1|
 5: iconst_1                           |a0|a1|         |a1| 1|
 6: iadd                               |a0|a1|         |v1|
 7: ireturn                            |a0|a1|         |
 8: iload_1                            |a0|a1|         |a1|
 9: ifne          20                   |a0|a1|         |
12: iload_0                            |a0|a1|         |a0|
13: iconst_1                           |a0|a1|         |a0| 1|
14: isub                               |a0|a1|         |v2|
15: iconst_1                           |a0|a1|         |v2| 1|
16: invokestatic ackermann:(II)I       |a0|a1|         |v3|
19: ireturn                            |a0|a1|         |
20: iload_0                            |a0|a1|         |a0|
21: iconst_1                           |a0|a1|         |a0| 1|
22: isub                               |a0|a1|         |v4|
23: iload_0                            |a0|a1|         |v4|a0|
24: iload_1                            |a0|a1|         |v4|a0|a1|
25: iconst_1                           |a0|a1|         |v4|a0|a1| 1|
26: isub                               |a0|a1|         |v4|a0|v5|
27: invokestatic ackermann:(II)I       |a0|a1|         |v4|v6|
30: invokestatic ackermann:(II)I       |a0|a1|         |v7|
33: ireturn                            |a0|a1|         |
```

```java
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

In Bytecode, there is an operand STACK where values can be placed to be operated on, and an array of LOCALS where values can be stored in between operations.

I have annotated the LOCALS and STACK here for clarity, so you can see the height of the stack growing as values are moved onto it, and shrinking as computations like **iadd** combine multiple values into one.

```
BYTECODE                              LOCALS          STACK
                                      |a0|a1|         |
 0: iload_0                           |a0|a1|         |a0|
 1: ifne          8                   |a0|a1|         |
 4: iload_1                           |a0|a1|         |a1|
 5: iconst_1                          |a0|a1|         |a1| 1|
 6: iadd                              |a0|a1|         |v1|
 7: ireturn                           |a0|a1|         |
 8: iload_1                           |a0|a1|         |a1|
 9: ifne          20                  |a0|a1|         |
12: iload_0                           |a0|a1|         |a0|
13: iconst_1                          |a0|a1|         |a0| 1|
14: isub                              |a0|a1|         |v2|
15: iconst_1                          |a0|a1|         |v2| 1|
16: invokestatic ackermann:(II)I      |a0|a1|         |v3|
19: ireturn                           |a0|a1|         |
20: iload_0                           |a0|a1|         |a0|
21: iconst_1                          |a0|a1|         |a0| 1|
22: isub                              |a0|a1|         |v4|
23: iload_0                           |a0|a1|         |v4|a0|
24: iload_1                           |a0|a1|         |v4|a0|a1|
25: iconst_1                          |a0|a1|         |v4|a0|a1| 1|
26: isub                              |a0|a1|         |v4|a0|v5|
27: invokestatic ackermann:(II)I      |a0|a1|         |v4|v6|
30: invokestatic ackermann:(II)I      |a0|a1|         |v7|
33: ireturn                           |a0|a1|         |
```

```java
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
  else return ackermann2(ackermann(m, n - 1));
}
```

The big issue with bytecode is that modifying the program is hard. Imagine if we want to replace the outer call to **ackermann** taking two arguments with a call to **ackermann2** only taking one argument.

```
BYTECODE                               LOCALS          STACK
                                       |a0|a1|         |
 0: iload_0                            |a0|a1|         |a0|
 1: ifne           8                   |a0|a1|         |
 4: iload_1                            |a0|a1|         |a1|
 5: iconst_1                           |a0|a1|         |a1| 1|
 6: iadd                               |a0|a1|         |v1|
 7: ireturn                            |a0|a1|         |
 8: iload_1                            |a0|a1|         |a1|
 9: ifne           20                  |a0|a1|         |
12: iload_0                            |a0|a1|         |a0|
13: iconst_1                           |a0|a1|         |a0| 1|
14: isub                               |a0|a1|         |v2|
15: iconst_1                           |a0|a1|         |v2| 1|
16: invokestatic ackermann:(II)I       |a0|a1|         |v3|
19: ireturn                            |a0|a1|         |
20: iload_0                            |a0|a1|         |a0|
21: iconst_1                           |a0|a1|         |a0| 1|
22: isub                               |a0|a1|         |v4|
23: iload_0                            |a0|a1|         |v4|a0|
24: iload_1                            |a0|a1|         |v4|a0|a1|
25: iconst_1                           |a0|a1|         |v4|a0|a1| 1|
26: isub                               |a0|a1|         |v4|a0|v5|
27: invokestatic ackermann:(II)I       |a0|a1|         |v4|v6|
30: invokestatic ackermann:(II)I       |a0|a1|         |v7|
30: invokestatic ackermann2:(I)I       |a0|a1|         |v7|
33: ireturn                            |a0|a1|         |
```

```java
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
  else return ackermann2(ackermann(m, n - 1));
}
```

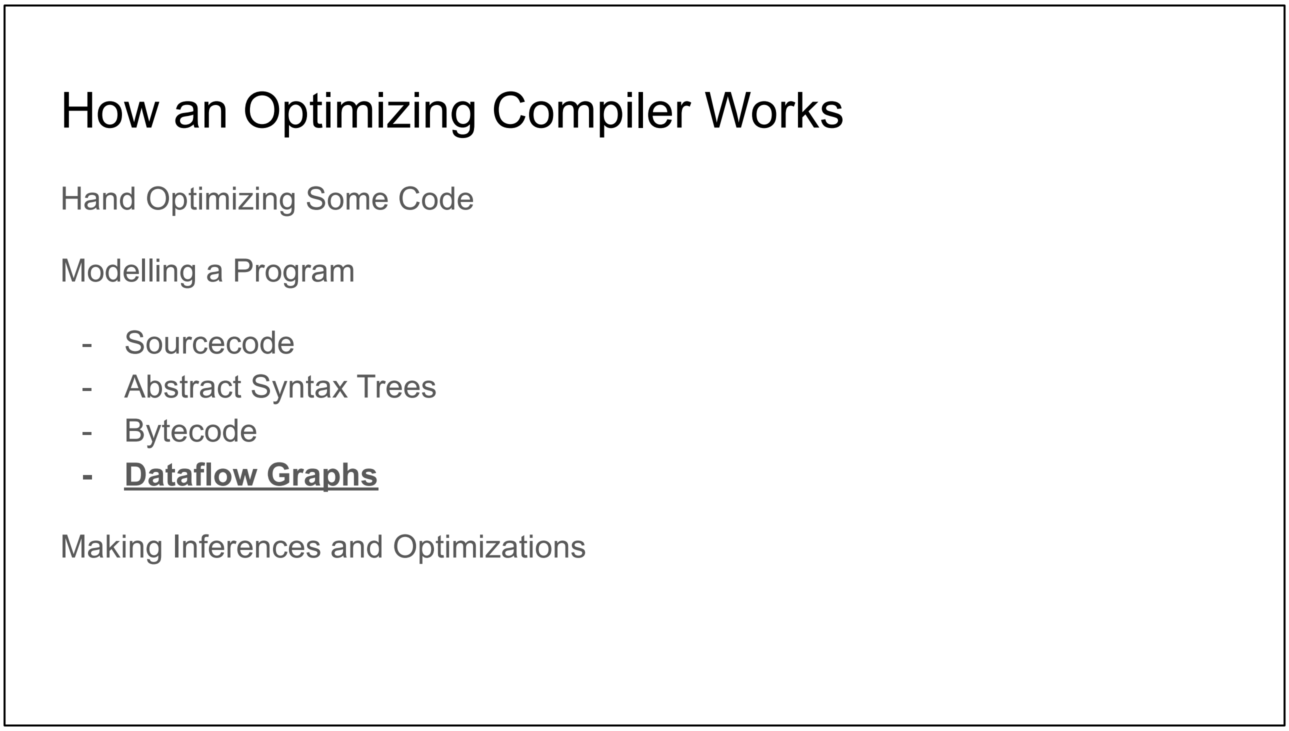
Not only do you have to replace the instruction itself.

```
BYTECODE                             LOCALS          STACK
                                     |a0|a1|         |
 0: iload_0                          |a0|a1|         |a0|
 1: ifne          8                  |a0|a1|         |
 4: iload_1                          |a0|a1|         |a1|
 5: iconst_1                         |a0|a1|         |a1| 1|
 6: iadd                             |a0|a1|         |v1|
 7: ireturn                          |a0|a1|         |
 8: iload_1                          |a0|a1|         |a1|
 9: ifne          20                 |a0|a1|         |
12: iload_0                          |a0|a1|         |a0|
13: iconst_1                         |a0|a1|         |a0| 1|
14: isub                             |a0|a1|         |v2|
15: iconst_1                         |a0|a1|         |v2| 1|
16: invokestatic ackermann:(II)I     |a0|a1|         |v3|
19: ireturn                          |a0|a1|         |
20: iload_0                          |a0|a1|         |a0|
21: iconst_1                         |a0|a1|         |a0| 1|
22: isub                             |a0|a1|         |v4|
23: iload_0                          |a0|a1|         |v4|a0|
24: iload_1                          |a0|a1|         |v4|a0|a1|
25: iconst_1                         |a0|a1|         |v4|a0|a1| 1|
26: isub                             |a0|a1|         |v4|a0|v5|
27: invokestatic ackermann:(II)I     |a0|a1|         |v4|v6|
30: invokestatic ackermann:(II)I     |a0|a1|         |v7|
30: invokestatic ackermann2:(I)I     |a0|a1|         |v7|
33: ireturn                          |a0|a1|         |
```

```java
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
  else return ackermann2(ackermann(m, n - 1));
}
```

Uou also need to trace the now-unnecessary first argument backwards along the STACK to find three more instructions that need to be removed.

With bytecode, the inputs to an instruction may come from other instructions scattered throughout the program, and modifying one instruction often requires a similar scattering of supporting changes. That makes it difficult to work with.



How an Optimizing Compiler Works

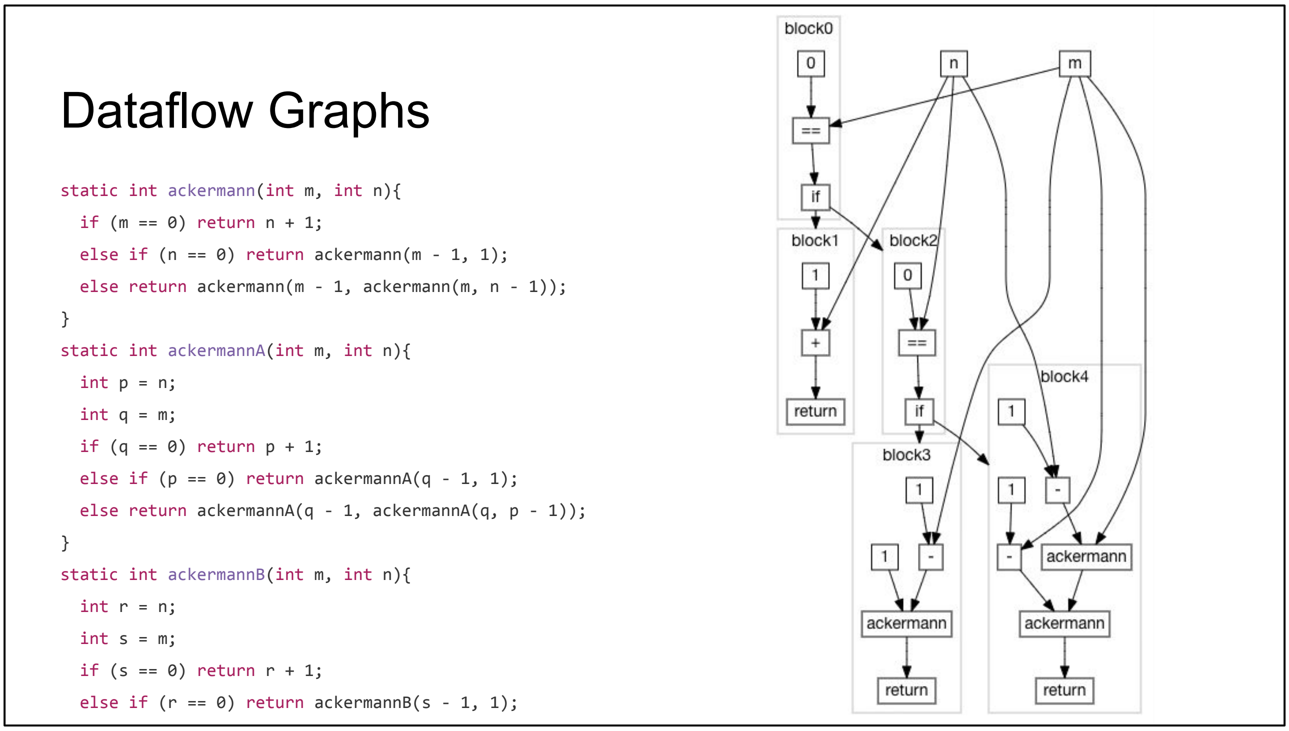
Hand Optimizing Some Code

Modelling a Program

- Sourcecode
- Abstract Syntax Trees
- Bytecode
- **<u>Dataflow Graphs</u>**

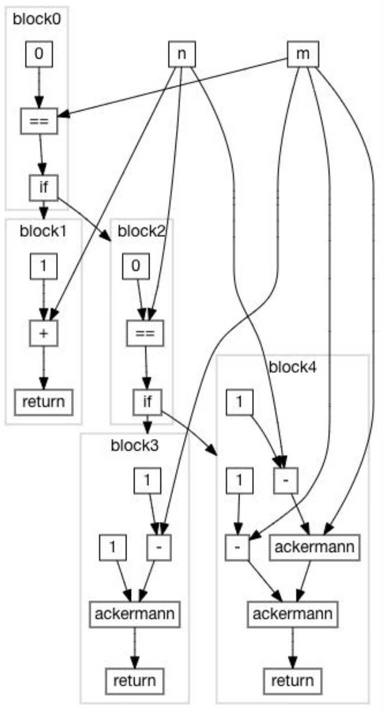
Making Inferences and Optimizations

The **Ident**/**Assign** pairs in our ASTs and the LOCALS and STACK of our bytecode both form a directed graph, with edges going from the instruction that computes a value, to the instructions that use it. Why not work with that graph directly?



Dataflow Graphs

```
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
static int ackermannA(int m, int n){
  int p = n;
  int q = m;
  if (q == 0) return p + 1;
  else if (p == 0) return ackermannA(q - 1, 1);
  else return ackermannA(q - 1, ackermannA(q, p - 1));
}
static int ackermannB(int m, int n){
  int r = n;
  int s = m;
  if (s == 0) return r + 1;
  else if (r == 0) return ackermannB(s - 1, 1);
```

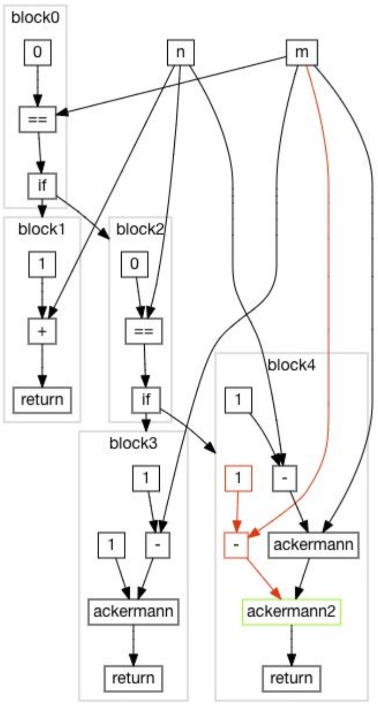


Dataflow graphs are like ASTs, but with the **Assign**/**Ident** pairs explicitly modelled as first class edges. A dataflow graph doesn't care how you move values in an out of local variables: only where values come from, and where they are used.

In this case, **ackermann** and both variants **ackermannA** and **ackermannB** all result in the same dataflow graph

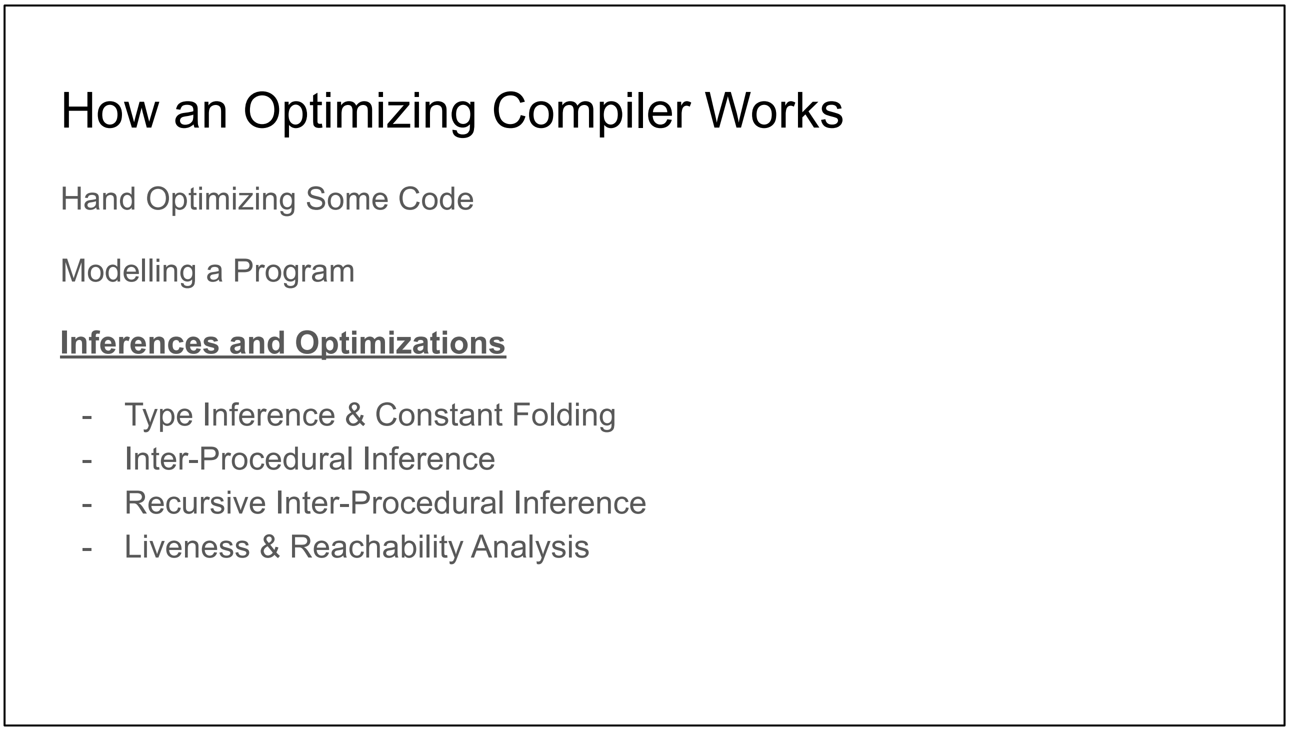
Dataflow Graphs

```
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
  else return ackermann2(ackermann(m, n - 1));
}
```



In a dataflow graph, the inputs to an instruction are simply the incoming edges. A modification like the **ackermann/ackermann2** change earlier is just replacing one node, removing one edge, and removing the nodes upstream of it. When analyzing an instruction, there are generally direct edges between that instruction and all other instructions you might care about, making dataflow graphs relatively easy to both analyze and modify.

This description of dataflow graphs is greatly simplified. We can go into more detail later if we have time



How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

Inferences and Optimizations

- Type Inference & Constant Folding
- Inter-Procedural Inference
- Recursive Inter-Procedural Inference
- Liveness & Reachability Analysis

Now that we have seen the different optimizations we want to do, and the different ways we can model programs, let’s finally move on to performing those inferences and optimizations automatically

How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

Inferences and Optimizations

- **Type Inference & Constant Folding**
- Inter-Procedural Inference
- Recursive Inter-Procedural Inference
- Liveness & Reachability Analysis

First, let’s look at type inference and constant folding

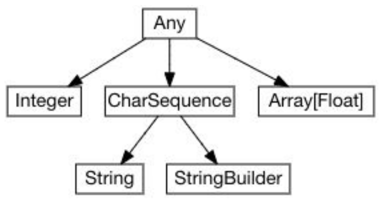
Type Inference & Constant Folding

What do we know about a value?

- Is it an Integer? String? Array[Float]? PrintLogger?
- Is it a CharSequence, which could be either a String or a StringBuilder?
- Is it Any, meaning we don't know anything about it?

Type inference involves inferring constraints about values within your program. Although the ASTs and bytecode may already be typed, we can often infer much more detail than is already present.

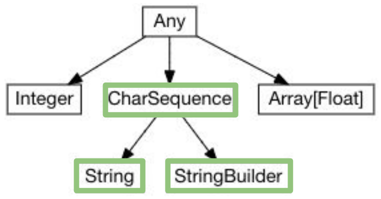
Type Lattices



Types are often modelled as a lattice, which tells us how much we know about a value. Can it be Anything? Or only an integer? Or a CharSequence? Or a specific CharSequence, like String or StringBuilder?

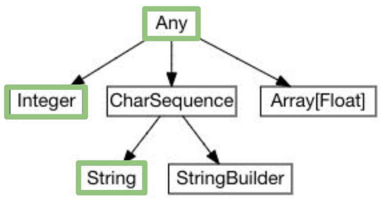
Given a value in your program, the higher up in the lattice its inferred type lives, the less we know about it. If we have a value which could be one of two types, we look for the least upper bound in the lattice and treat it as that.

Type Lattices



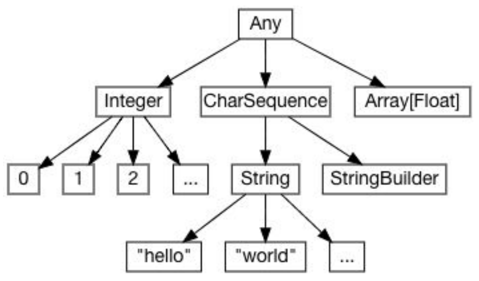
For example, a value that is either a **String** or **StringBuilder** is treated as a **CharSequence**

Type Lattices



While a value which could be one of **String** or **Integer** is treated as **Any**

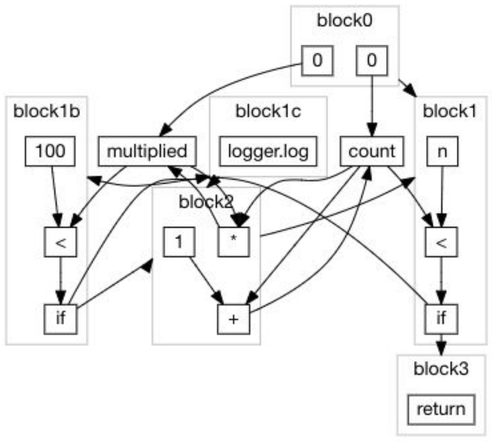
Type Lattices



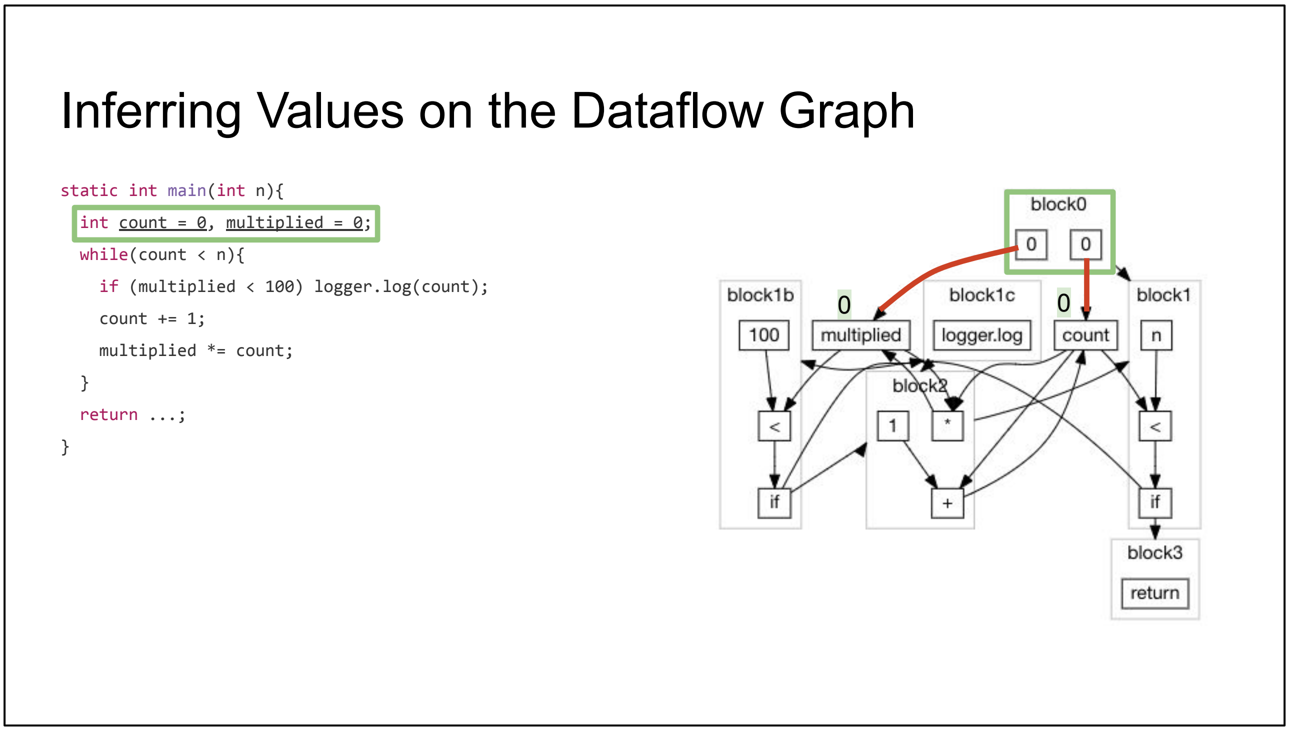
Sometimes we infer not only that something is an integer, but that it is the integer **0**. Or that not only is something a String, but the string **“hello”**. Extending the lattice to handle this straightforward: these are called “singleton types”, and can be treated the same as any other type in our lattice.

Inferring Values on the Dataflow Graph

```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```

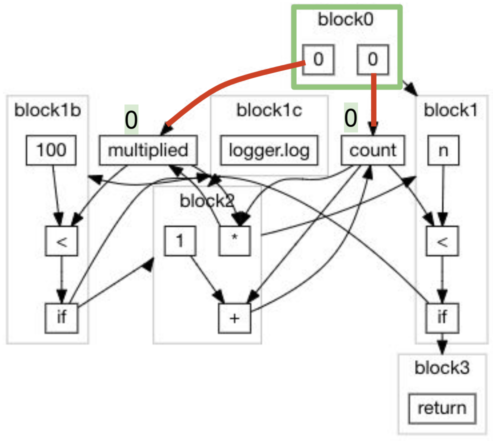


To see how type inference works, let's look at a simplified version of our **main** method, and its corresponding Dataflow Graph



Inferring Values on the Dataflow Graph

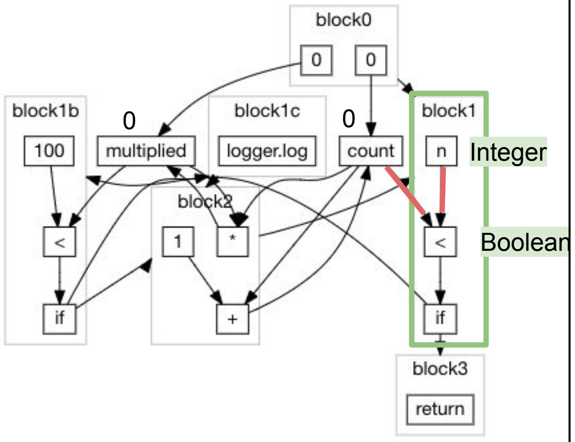
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



We start off at **block0**, and see that **0** is assigned to **multiplied** and **count**

Inferring Values on the Dataflow Graph

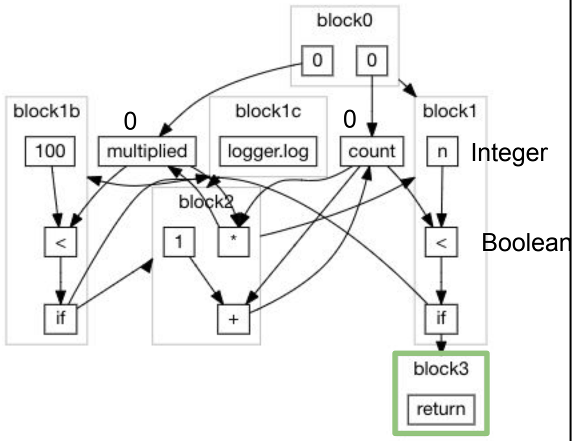
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



Next, we transition into **block1**, **n** is an unknown **Integer** input parameter, and **0 < Integer** is an unknown **Boolean**, so we need to consider both branches of the **if**

Inferring Values on the Dataflow Graph

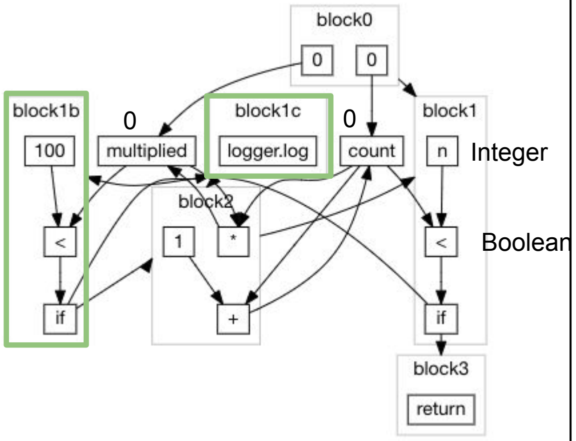
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



block3 is just a return, and we can ignore it for now

Inferring Values on the Dataflow Graph

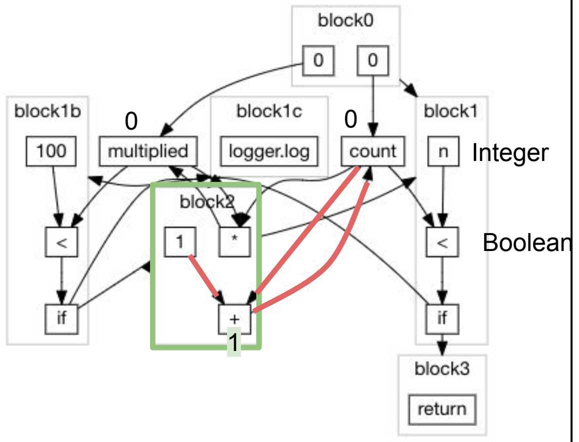
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



While on the other branch of the **if**, **block1b** and **block1c** are just doing some logging, and can also be mostly ignored.

Inferring Values on the Dataflow Graph

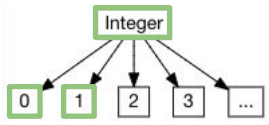
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```

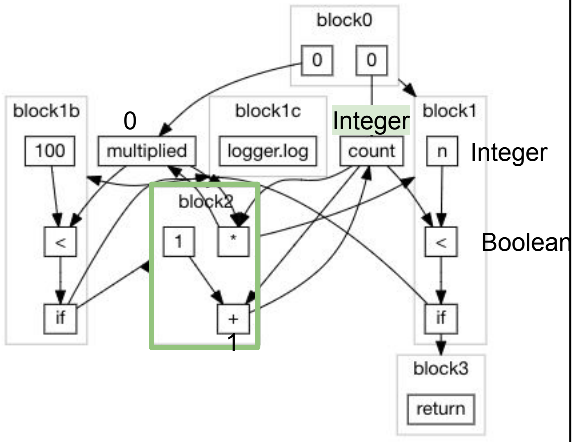


block2 is the next block that modifies **multiplied** and **count**. We see it adds **count** (previously **0**) and **1**, giving us **1**, and stores it back in **count**. We now know **count** is either **0** or **1**

Inferring Values on the Dataflow Graph

```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```

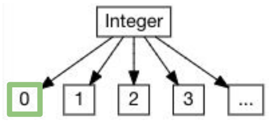


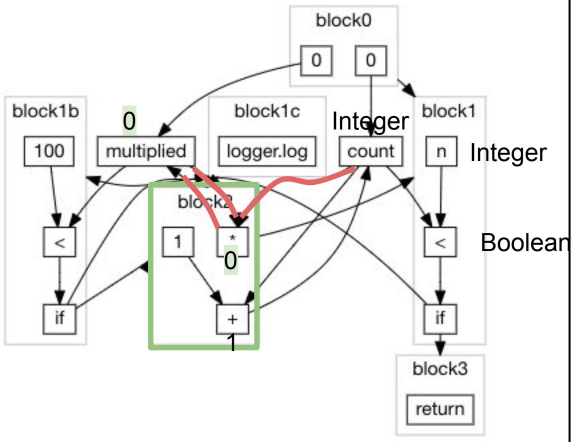


We can look at our type lattice to see that the least upper bound of both **0** and **1** is **Integer**, and assign that inference to **count**.

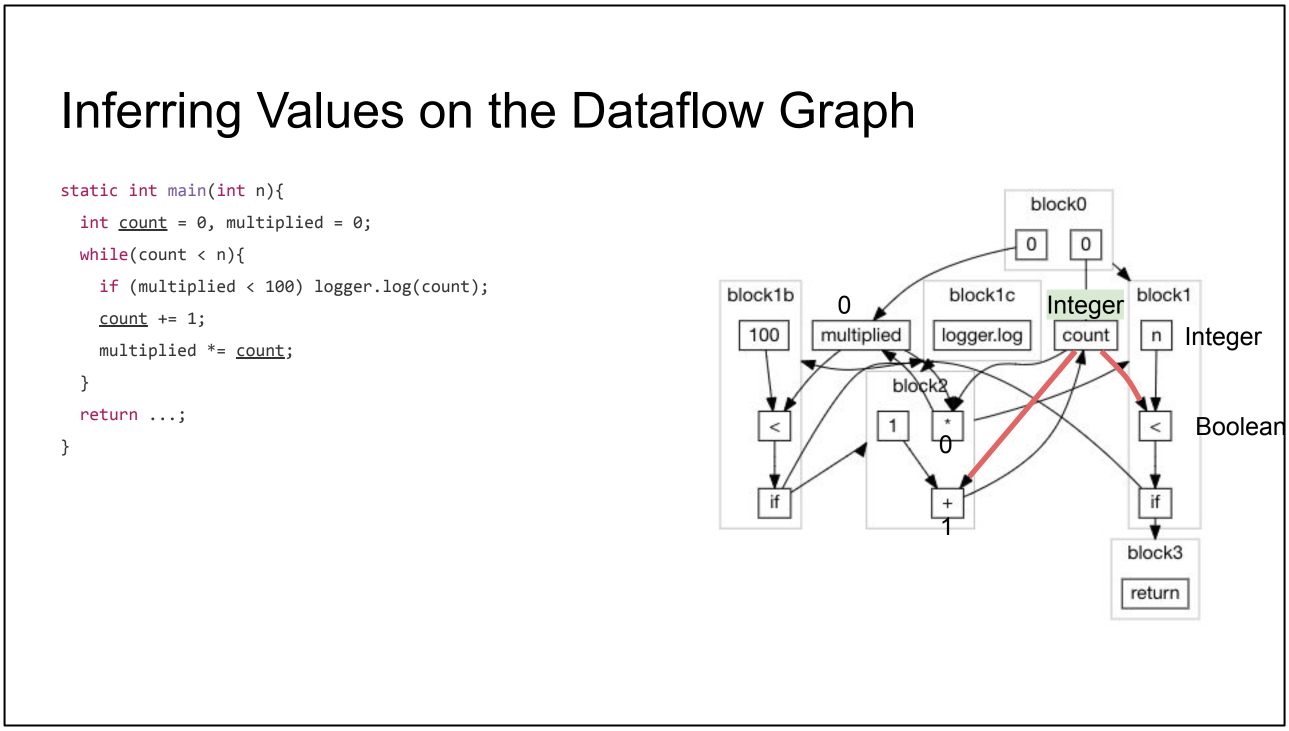
Inferring Values on the Dataflow Graph

```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



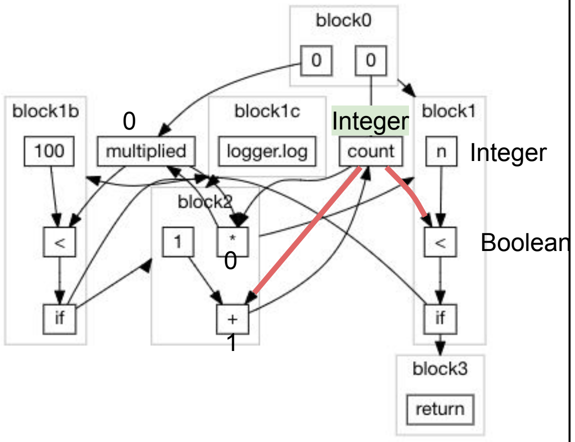


We also see it multiplying **multiplied** by **count**, but since **multiplied** is already **0**, it remains **0** even after multiplication.



Inferring Values on the Dataflow Graph

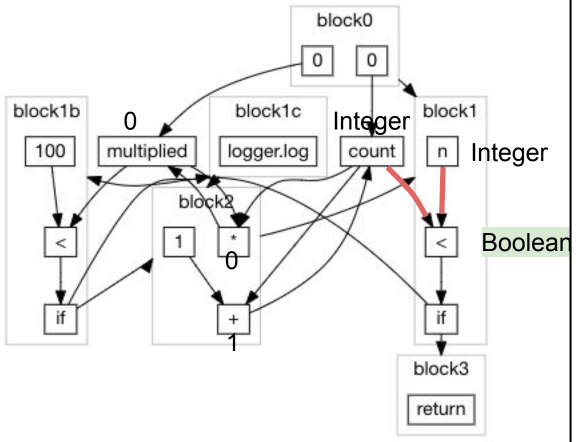
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



Since **count**'s inference has been updated from **1** to **Integer**, we have look at all the nodes downstream to see if they need updating

Inferring Values on the Dataflow Graph

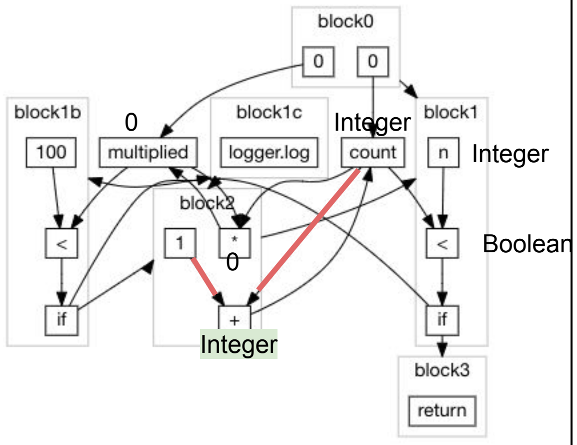
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



< taking two unknown **Integers** remains an unknown **Boolean**

Inferring Values on the Dataflow Graph

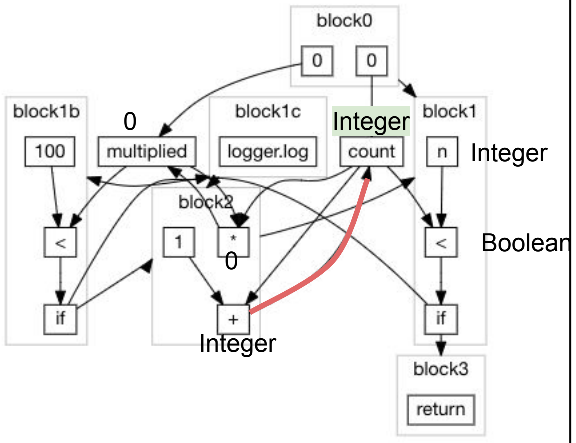
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



But **count + 1**, previously inferred as **1**, is now **Integer + 1** -> **Integer**

Inferring Values on the Dataflow Graph

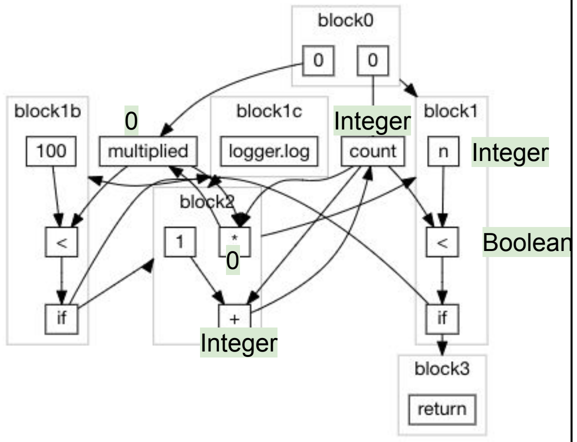
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



Since we the output of **+** gets stored back into **count**, we propagate the inference back there as well. **count** was previously an **Integer**, and remains an **Integer** now, so that propagation terminates.

Inferring Values on the Dataflow Graph

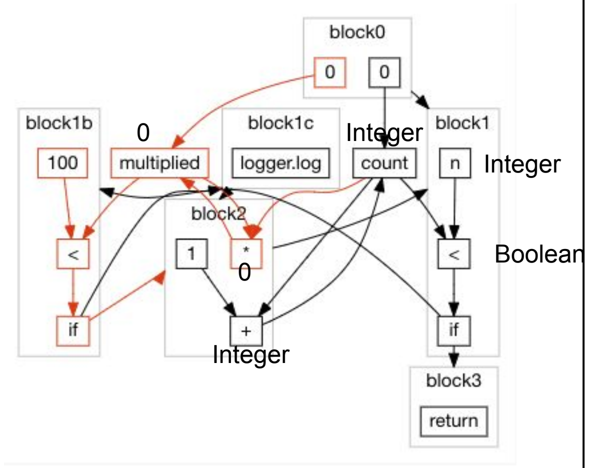
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



And we’re done: the dataflow inference has reach a fixed point, and further iteration will not change any of the inferred values.

Inferring Values on the Dataflow Graph

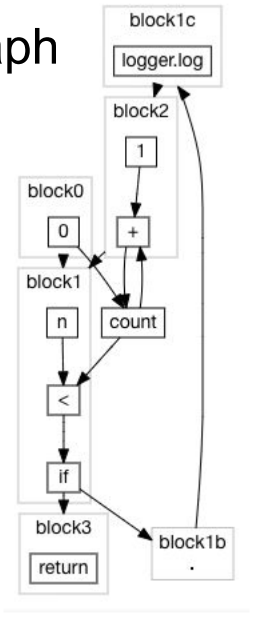
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



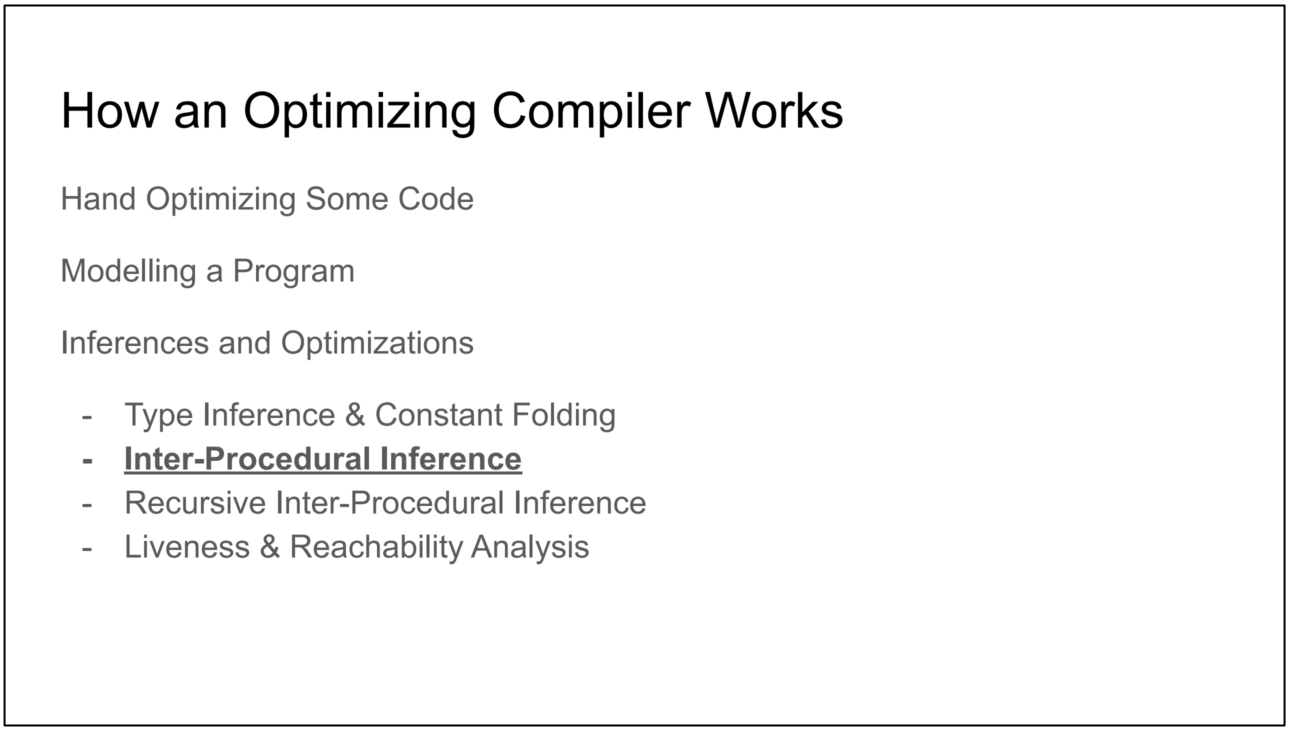
We can then make use of our inference to implify the program: since we know **multiplied** is always **0**, we can remove any nodes which affect **multiplied**, and replace **multiplied** with **0** in any node which uses it

Inferring Values on the Dataflow Graph

```
static int main(int n){
  int count = 0;
  while(count < n){
    logger.log(count);
    count += 1;
  }
  return ...;
}
```



That gives us the following simplified dataflow graph, which serializes into a simplified Java program. In the process of type inference, we have performed constant folding, which is really nothing more than normal type inference on singleton types.



How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

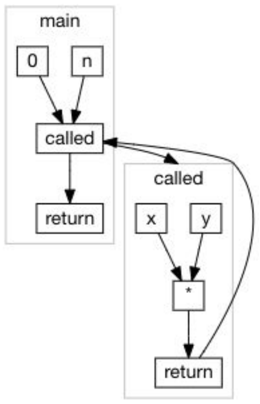
Inferences and Optimizations

- Type Inference & Constant Folding
- **Inter-Procedural Inference**
- Recursive Inter-Procedural Inference
- Liveness & Reachability Analysis

We've seen how inference works within a single function body. Now let's look at how it works between functions.

Inter-Procedural Inference

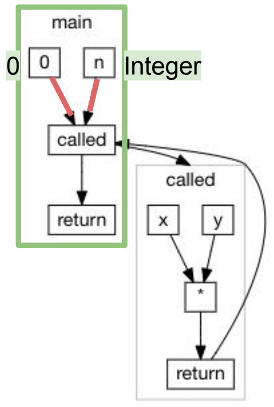
```
static int main(int n){
    return called(0, n);
}

static int called(int x, int y){
    return x * y;
}
```



Consider this case with two simple functions, one calling the other

Inter-Procedural Inference

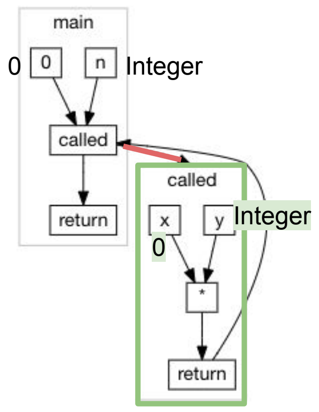
```
static int main(int n){
    return called(0, n);
}

static int called(int x, int y){
    return x * y;
}
```



We start off in the **main** function, which takes an argument **n** as an unknown **Integer**, and passes it with **0** to the function **called**. To know what type **called**'s return value is, we need to first analyze the body of **called**

Inter-Procedural Inference

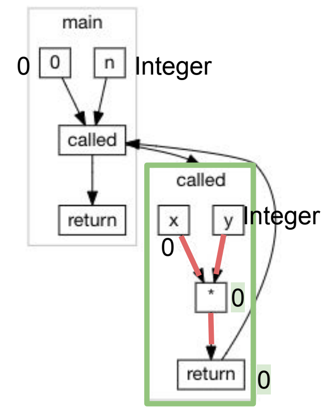
```
static int main(int n){
    return called(0, n);
}

static int called(int x, int y){
    return x * y;
}
```



Inside **called**, we see that **x** is bound to **0**, and **y** is bound to **Integer**.

Inter-Procedural Inference

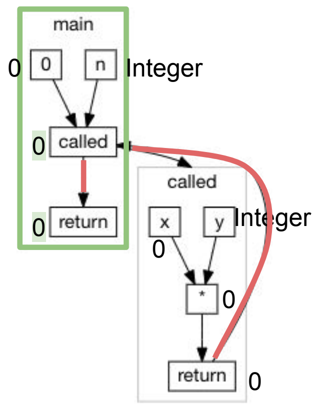
```
static int main(int n){
    return called(0, n);
}

static int called(int x, int y){
    return x * y;
}
```



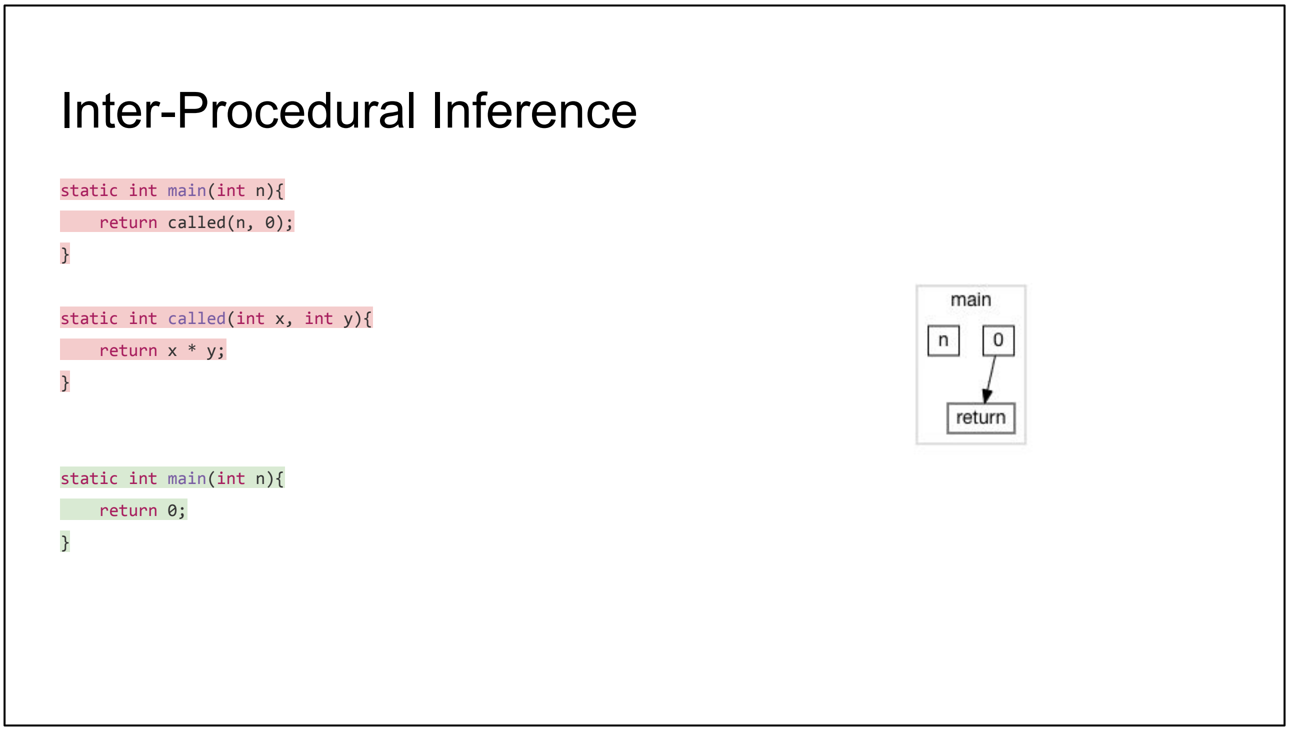
Thus ***** can be inferred to be **0**, which gets returned

Inter-Procedural Inference

```
static int main(int n){
    return called(0, n);
}

static int called(int x, int y){
    return x * y;
}
```

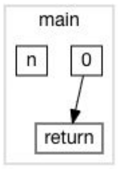


And so **main** can be inferred to return **0**

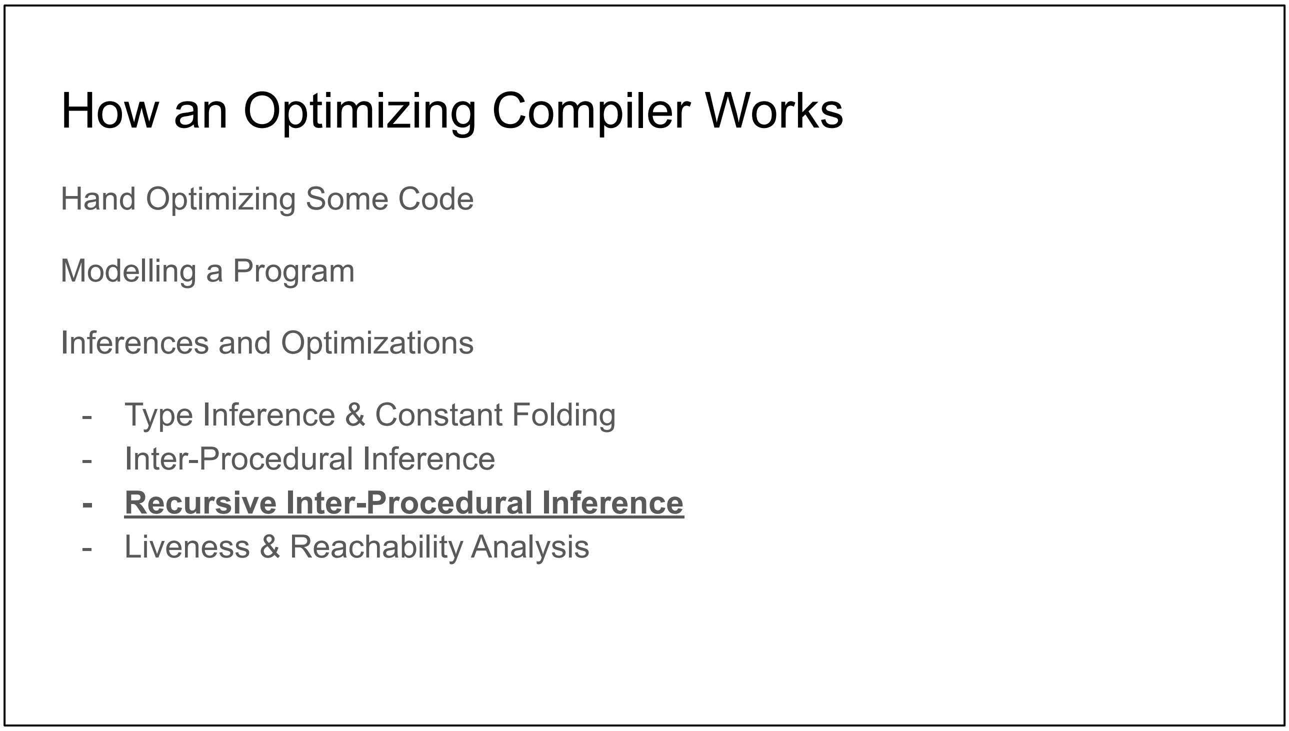


Inter-Procedural Inference

```
static int main(int n){
    return called(n, 0);
}

static int called(int x, int y){
    return x * y;
}


static int main(int n){
    return 0;
}
```



From there, we can simplify our dataflow graph to make **main** return **0** directly, without all this rigmarole. This simplified dataflow graph serializes to the corresponding simplified program



How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

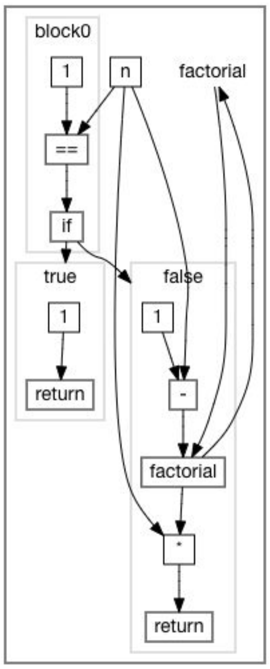
Inferences and Optimizations

- Type Inference & Constant Folding
- Inter-Procedural Inference
- **<u>Recursive Inter-Procedural Inference</u>**
- Liveness & Reachability Analysis

Next, let us look at how we handle inference of recursive functions

Recursive Inter-Procedural Inference

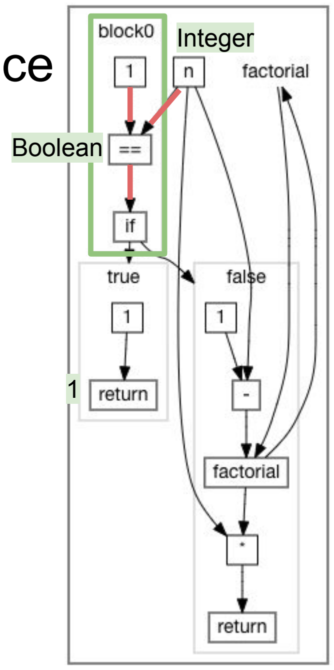
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```



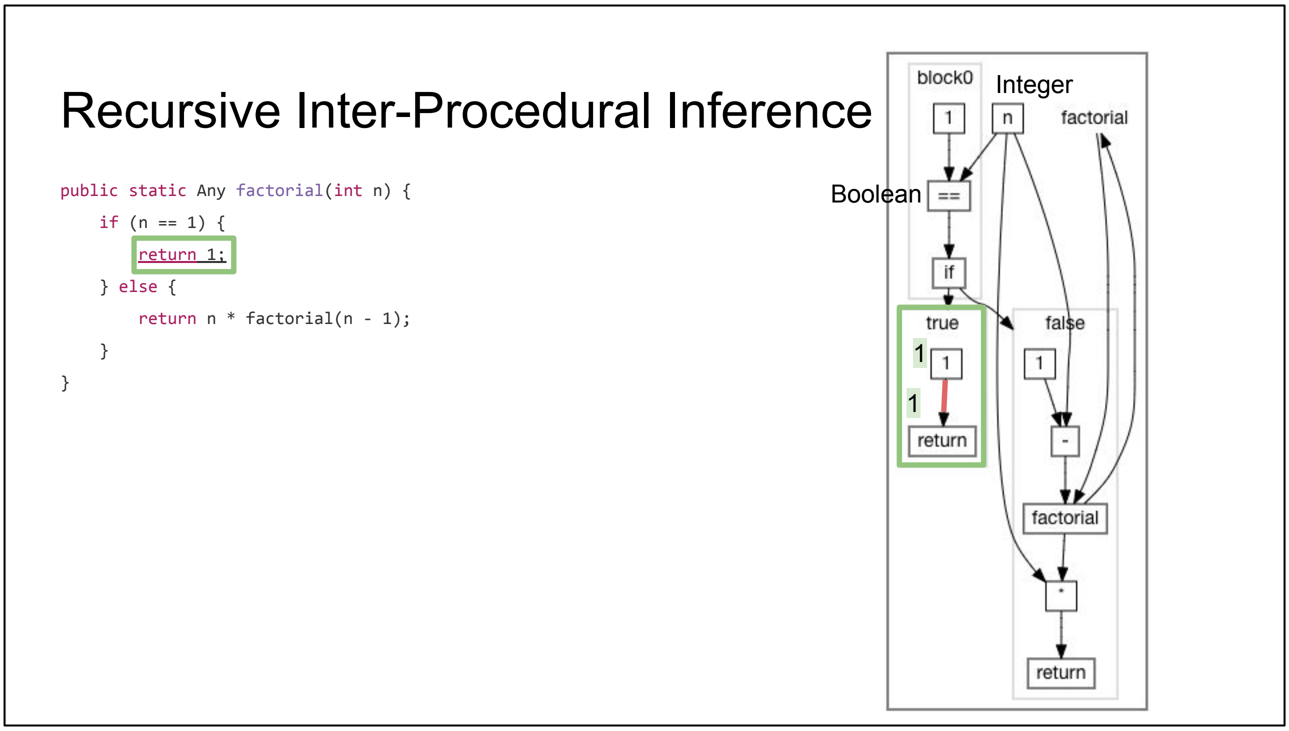
Consider this **factorial** function written in pseudo-java: it takes an **int n**, but it does not have its return type annotated (labeled **Any**). It is obvious to us that this function returns an integer, but how can our algorithm infer that automatically?

Recursive Inter-Procedural Inference

```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

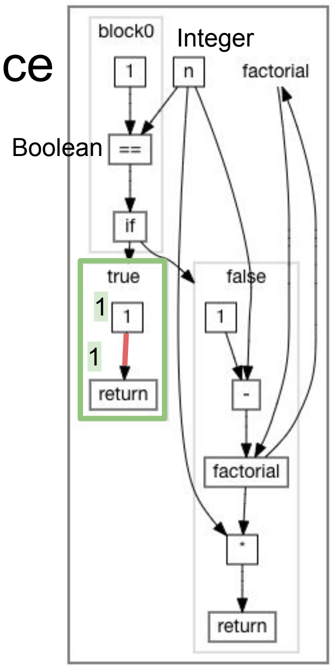


Following our existing strategy, we start at **block0. n** is an input **Integer**, **==** is an unknown **Boolean**, so we must consider both branches



Recursive Inter-Procedural Inference

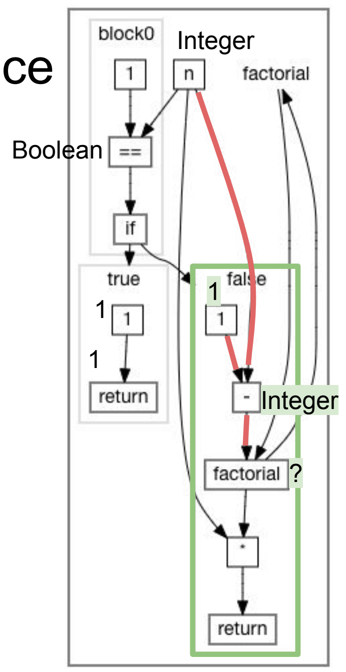
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```



The **true** block is simple: we return the constant **1**

Recursive Inter-Procedural Inference

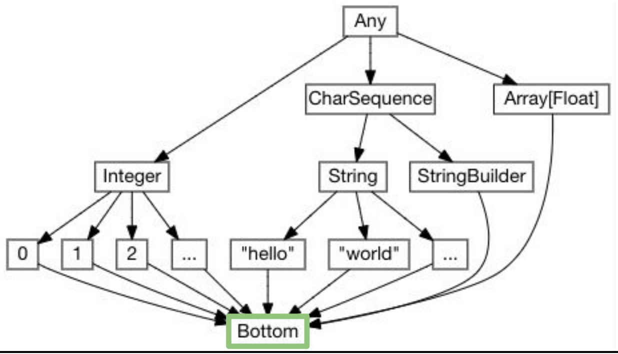
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

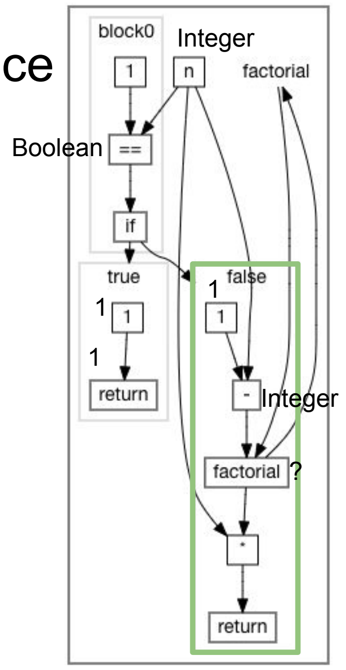


The false block, we know **n - 1** is **Integer - 1** which is just any **Integer**, but how do we infer the type of **factorial**? We can't recurse into it, because if we did we would just recurse infinitely.

Recursive Inter-Procedural Inference

```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

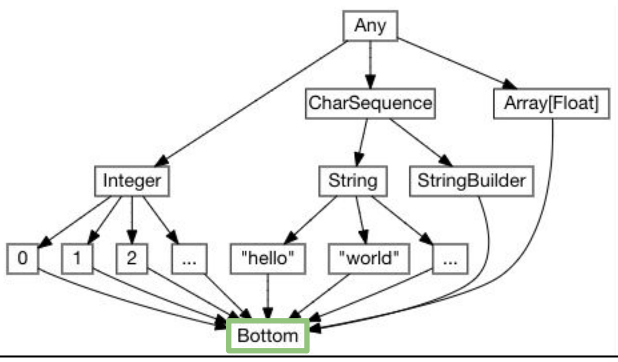


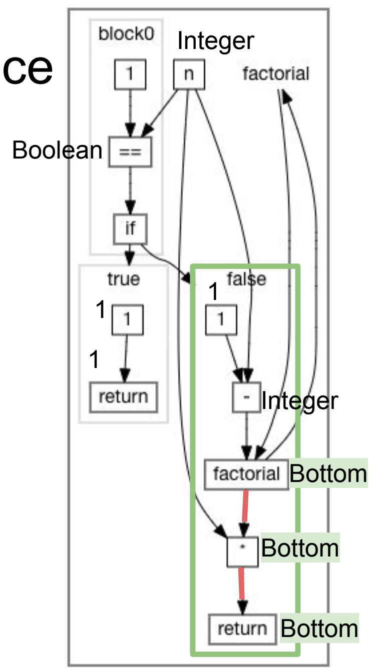


The solution to this is to extend our type lattice with a new type **Bottom**, which is a placeholder below every other type.

Recursive Inter-Procedural Inference

```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

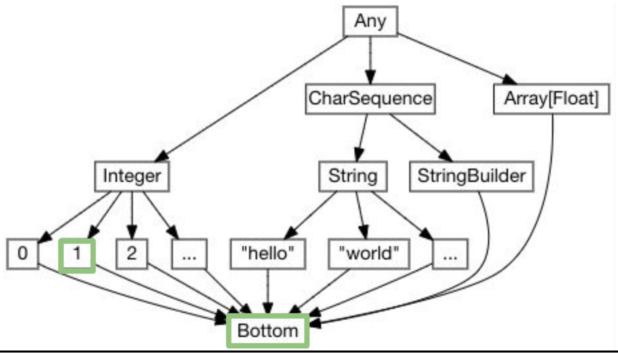


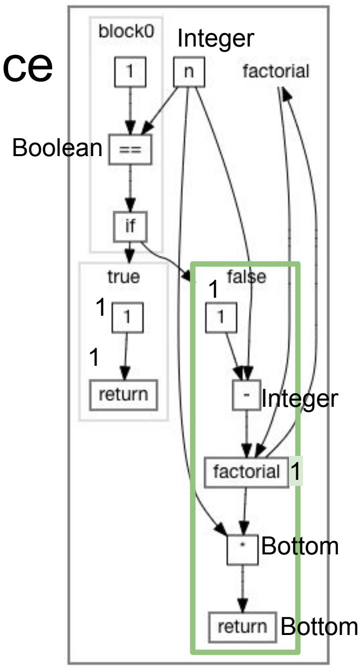


We tentatively infer the recursive call to **factorial** as **Bottom**, and everything downstream - here the ***** and **return** nodes - as **Bottom** as well.

Recursive Inter-Procedural Inference

```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```



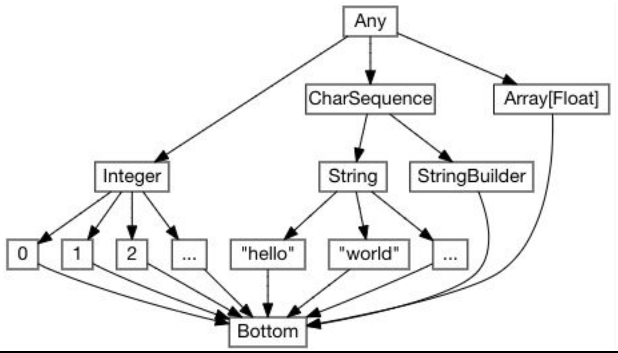


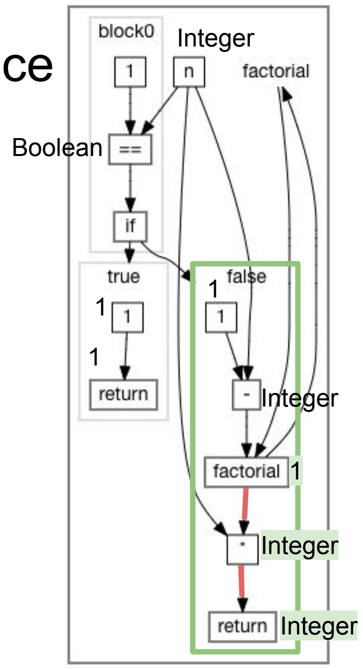
From this, we can take the least upper bound of the two returns: **1** and **Bottom**, and infer that the function as a whole must return **1**

Now, that we have inferred the return type of **factorial** using our **Bottom** placeholders, we must go back and plug **1** into the inference of our recursive **factorial** call.

Recursive Inter-Procedural Inference

```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

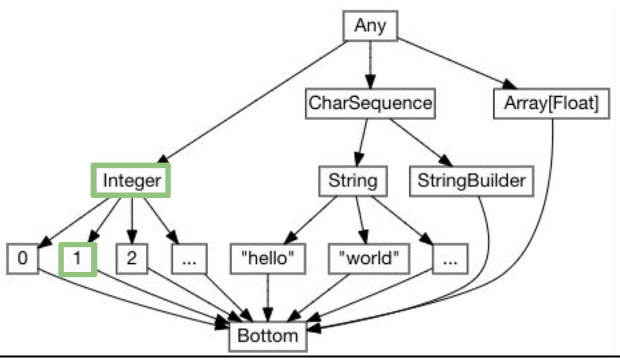


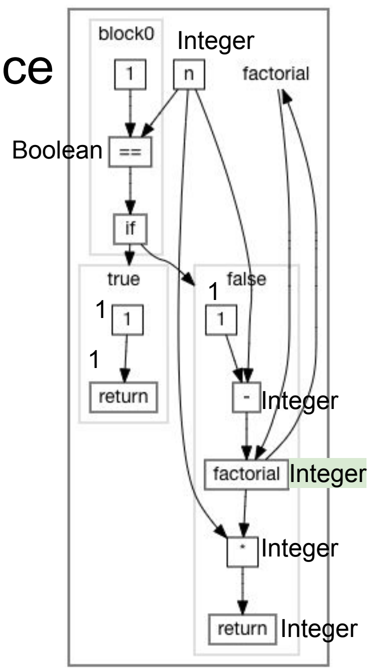


Since **factorial** returns **1**, **Integer * 1** is a **Integer**, and so the **false** block return can be inferred to return **Integer**.

Recursive Inter-Procedural Inference

```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```



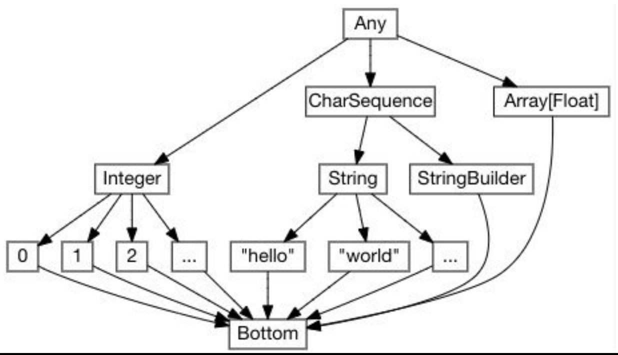


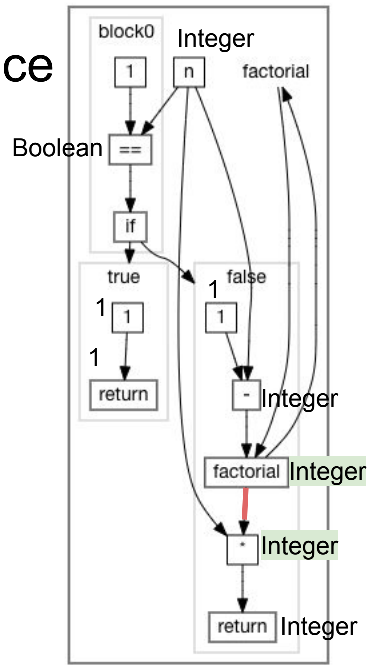
Now, we can again take the least upper bound of the two returns - **1** and **Integer** - and find that **factorial** must return an **Integer**

Again we plug that back into the recursive **factorial** call

Recursive Inter-Procedural Inference

```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

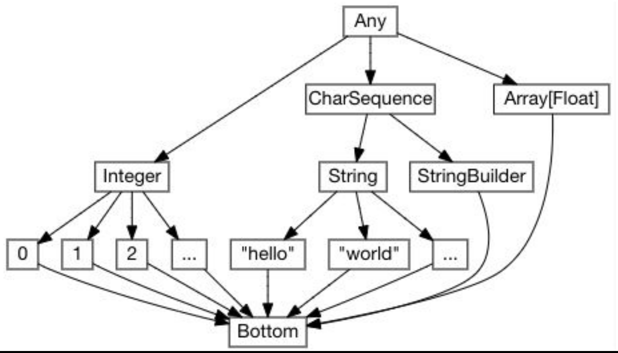


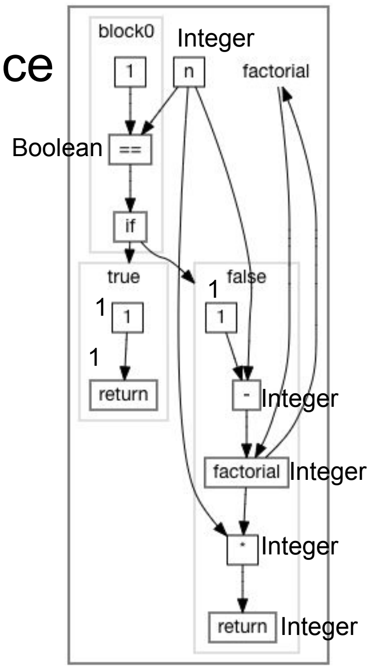


Giving us **factorial -> Integer** and *** -> Integer**. Since the inferred type of ***** did not change, the inference has reached a fixed point and stops.

Recursive Inter-Procedural Inference

```
public static Any factorial(int n) {
public static Integer factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```





Thus, we can infer that **factorial** returns not just **Any**, but an **Integer**!

You may have noticed that in two of cases we have walked through so far - analyzing the original **main** and analyzing **factorial** - are both iterative. We had to make multiple inference passes around either a loop or a recursive function call before our inference stabilizes.

In general, programs without cycles - loops or recursion - can always be analyzed in one pass, whereas given a program with cycles you have a choice: do one pass and settle for a less-precise inference, or use iteration to most precisely infer all their properties. Your number of passes is bounded by the height of your lattice.

How an Optimizing Compiler Works

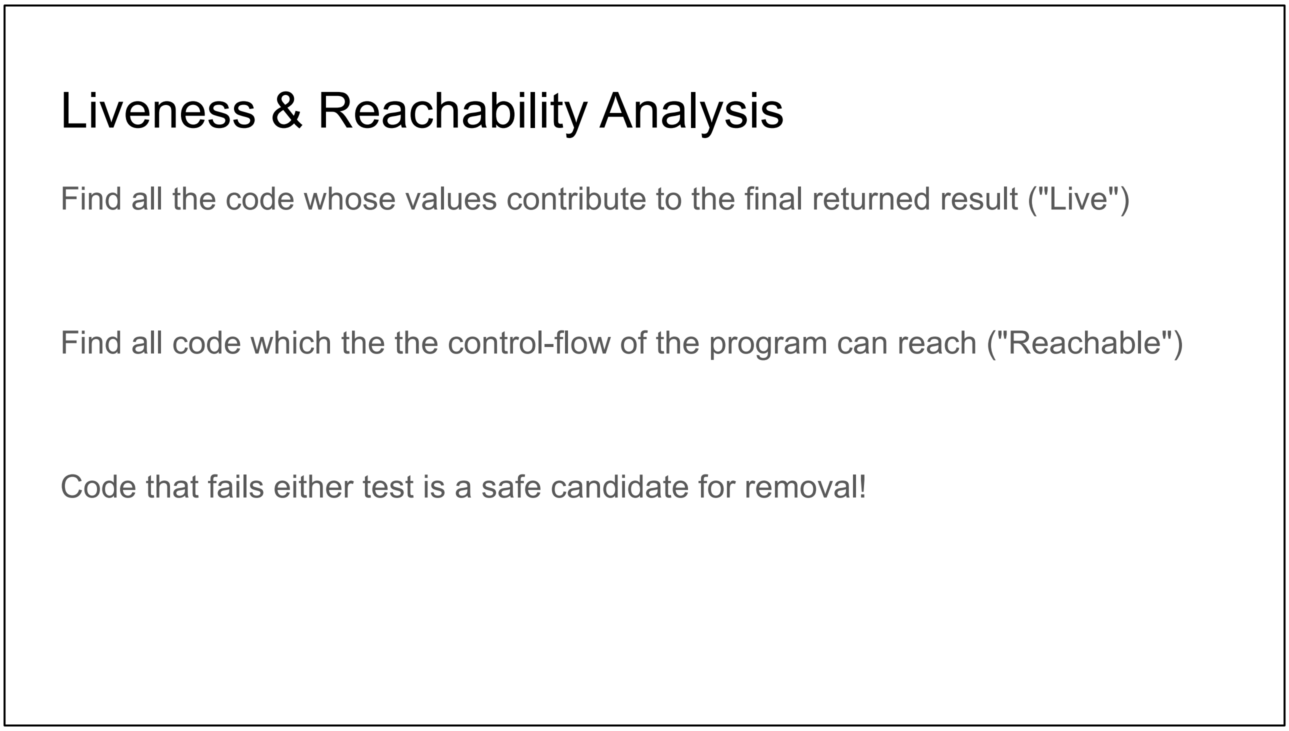
Hand Optimizing Some Code

Modelling a Program

Inferences and Optimizations

- Type Inference & Constant Folding
- Inter-Procedural Inference
- Recursive Inter-Procedural Inference
- **Liveness & Reachability Analysis**

The last thing I will go into is liveness and reachability analysis



Liveness & Reachability Analysis

Find all the code whose values contribute to the final returned result ("Live")

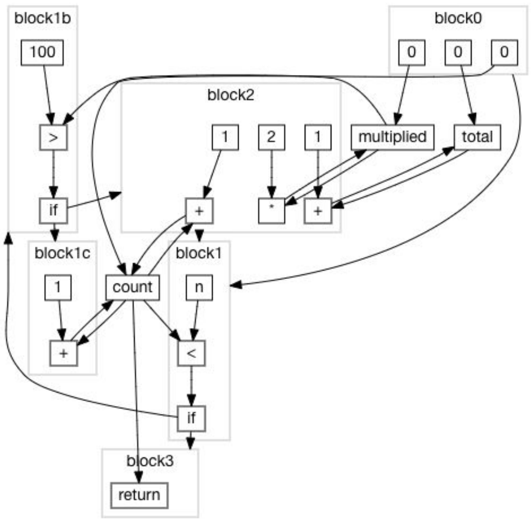
Find all code which the the control-flow of the program can reach ("Reachable")

Code that fails either test is a safe candidate for removal!

The goal of these two analyses is to remove code that either never runs, or runs but doesn’t affect the program output

Strawman Program

```
static int main(int n){
  int count = 0, total = 0, multiplied = 0;
  while(count < n){
    if (multiplied > 100) count += 1;
    count += 1;
    multiplied *= 2;
    total += 1;
  }
  return count;
}
```

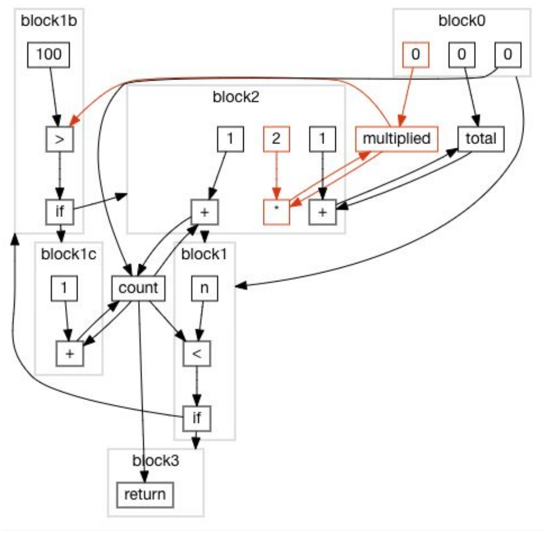


Here's another simplified version of **main**. We can see that **total** is not used, even though it gets initialized and updated. On the other hand, **count** is used, but it has a **count += 1** update site in the **if** block that never runs.

Thus **total** is "not live", while the **count += 1** is "not reachable": both can be eliminated.

Type Inference & Constant Folding

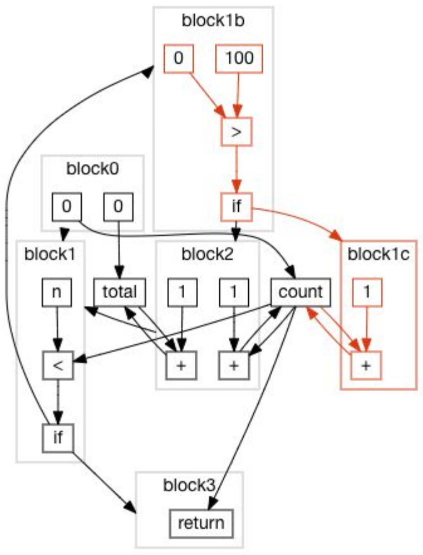
```
static int main(int n){
  int count = 0, total = 0, multiplied = 0;
  while(count < n){
    if (multiplied > 100) count += 1;
    count += 1;
    multiplied *= 2;
    total += 1;
  }
  return count;
}
```



To derive these facts, we start off with the same inference algorithm we saw earlier. This tells us **multiplied** is always **0**, and lets us simplify the dataflow graph

Branch Elimination & Reachability Analysis

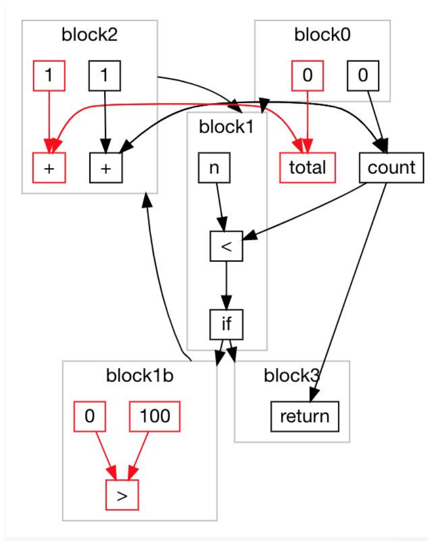
```
static int main(int n){
  int count = 0, total = 0;
  while(count < n){
    if (0 > 100) count += 1;
    count += 1;
    total += 1;
  }
  return count;
}
```



From here, we can see that **0 > 100** is always **false**. That means we can eliminate the **if** branch at the end of **block1b**, as well as removing the entire **block1c** in the process.

Liveness Optimizations

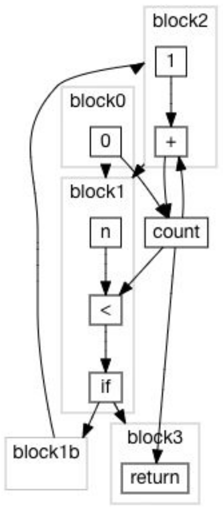
```
static int main(int n){
  int count = 0, total = 0;
  while(count < n){
    0 > 100;
    count += 1;
    total += 1;
  }
  return count;
}
```



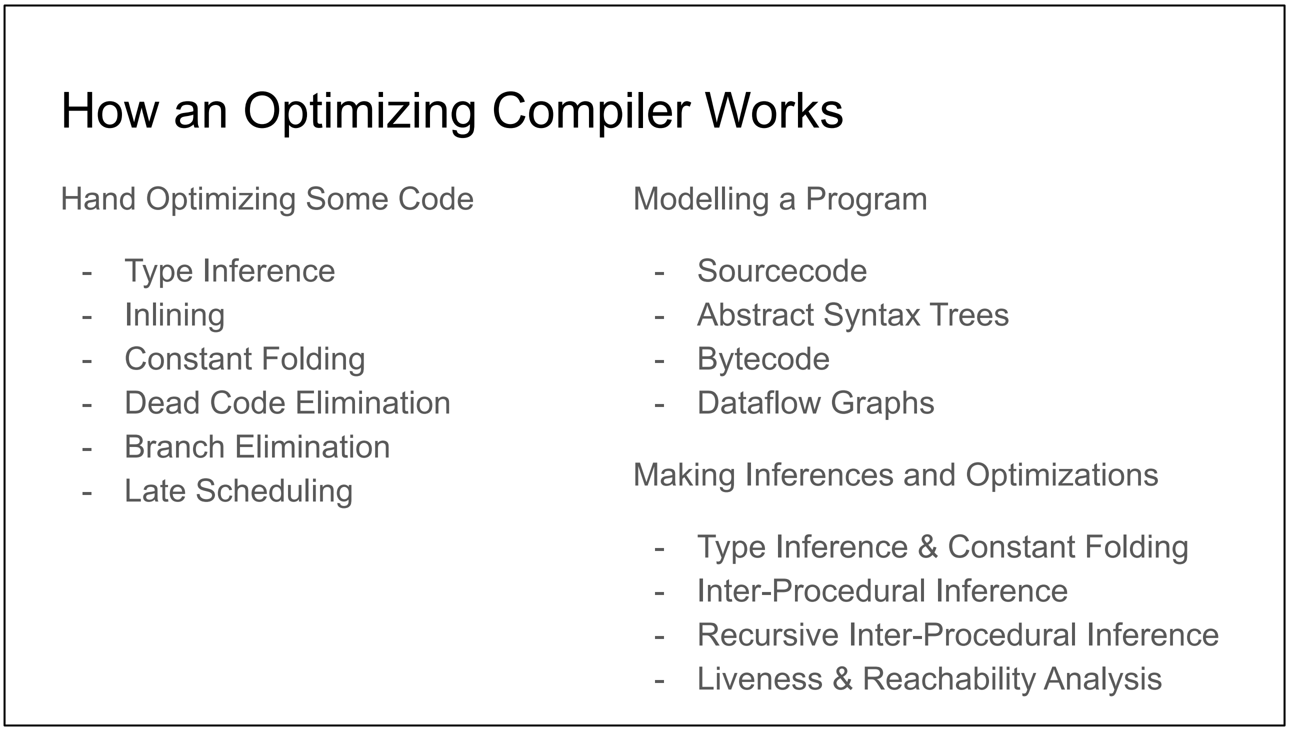
Lastly, we can do a simple breadth-first-traversal of the program graph starting from the **return** node and going upstream. This finds that **total** and **>** are not used in the final value being returned. We thus eliminate them as well

Final Output Code

```
static int main(int n){
  int count = 0;
  while(count < n){
    count += 1;
  }
  return count;
}
```



And so we end up with this simplified dataflow graph, and its corresponding simplified Java program



How an Optimizing Compiler Works

Hand Optimizing Some Code

- Type Inference
- Inlining
- Constant Folding
- Dead Code Elimination
- Branch Elimination
- Late Scheduling

Modelling a Program

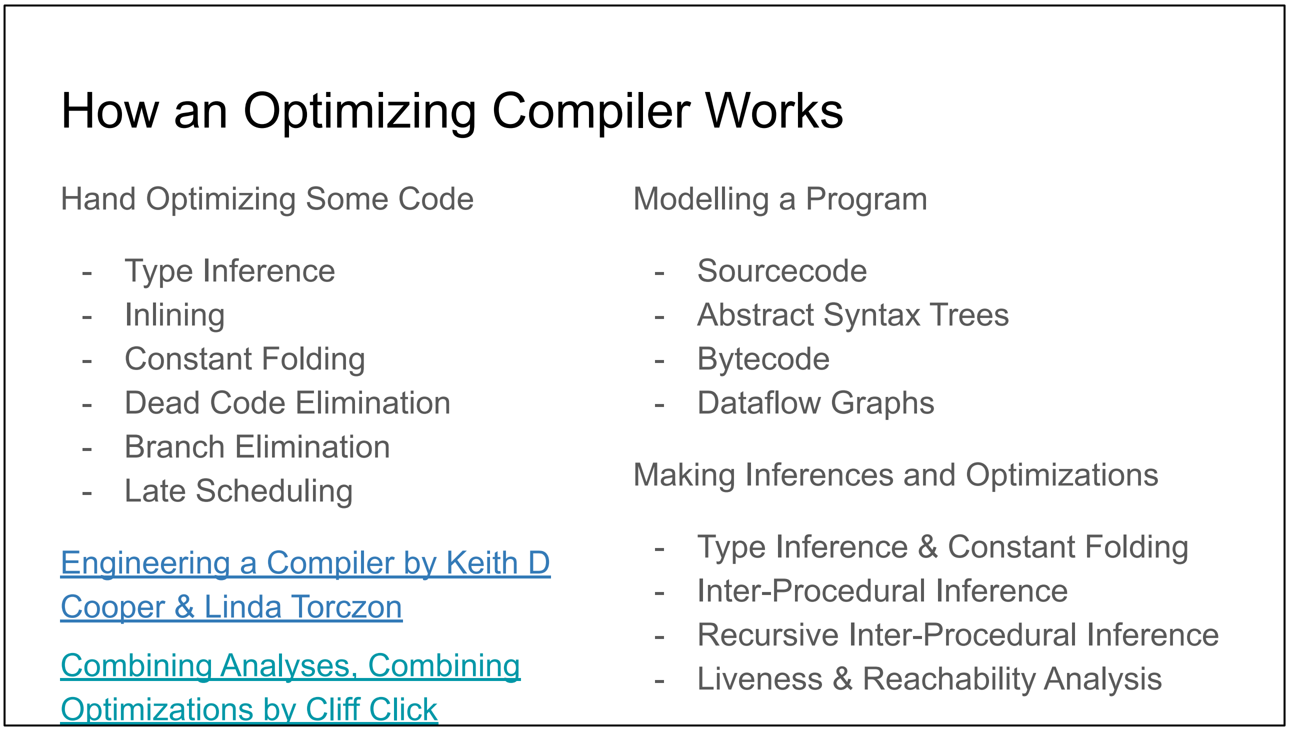
- Sourcecode
- Abstract Syntax Trees
- Bytecode
- Dataflow Graphs

Making Inferences and Optimizations

- Type Inference & Constant Folding
- Inter-Procedural Inference
- Recursive Inter-Procedural Inference
- Liveness & Reachability Analysis

To wrap up, we have seen some optimizations that we can perform manually, compared different ways an optimizing compiler can model a program in memory, and saw how to use algorithms on a graph to optimize our code: even in the presence of loops, function calls, or recursion. We saw how simple traversals over simple data structures is enough to perform optimizations that typically require a human programmer to do.

The examples here were of Java sourcecode and bytecode, but the techniques generalize to work with most programming languages.



How an Optimizing Compiler Works

Hand Optimizing Some Code

- Type Inference
- Inlining
- Constant Folding
- Dead Code Elimination
- Branch Elimination
- Late Scheduling

Engineering a Compiler by Keith D Cooper & Linda Torczon

Combining Analyses, Combining Optimizations by Cliff Click

Modelling a Program

- Sourcecode
- Abstract Syntax Trees
- Bytecode
- Dataflow Graphs

Making Inferences and Optimizations

- Type Inference & Constant Folding
- Inter-Procedural Inference
- Recursive Inter-Procedural Inference
- Liveness & Reachability Analysis

If you want to dig deeper in this topic, this book and paper are a great jumping off points into the field of optimizing compilation

The approach I have presented is just one of the many different ways of architecting an optimizing compiler, but is an approach used in real production systems such as the Java C2 HotSpot JIT. Hopefully this talk has helped give an intuition for the mechanisms through which optimizing compilers work their magic.