How an Optimizing Compiler Works

Rewriting code with simple data structures and algorithms

Li Haoyi, Scaladays 12 June 2019

Who Am I

Software Engineer at Databricks

Developer Tools

Lots of Scala internally

Lots of cool technology

Unified Analytics

Hiring in SF and Amsterdam!

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

How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

Making Inferences and Optimizations

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

Manual Optimizations: Baseline

```
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;
}
```

```
// 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); }
}
```

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); }
}
```

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); }
}
```

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); }
}
```

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); }
- }
```

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));
}
```

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));
}
```

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));
}
```

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));
}
```

Automated Optimizations

How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

- Sourcecode
- Abstract Syntax Trees
- Bytecode
- Dataflow Graphs

Making Inferences and Optimizations

How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

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

Making Inferences and Optimizations

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));
}
"""
```

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));           }}
"""
```

How an Optimizing Compiler Works

Hand Optimizing Some Code

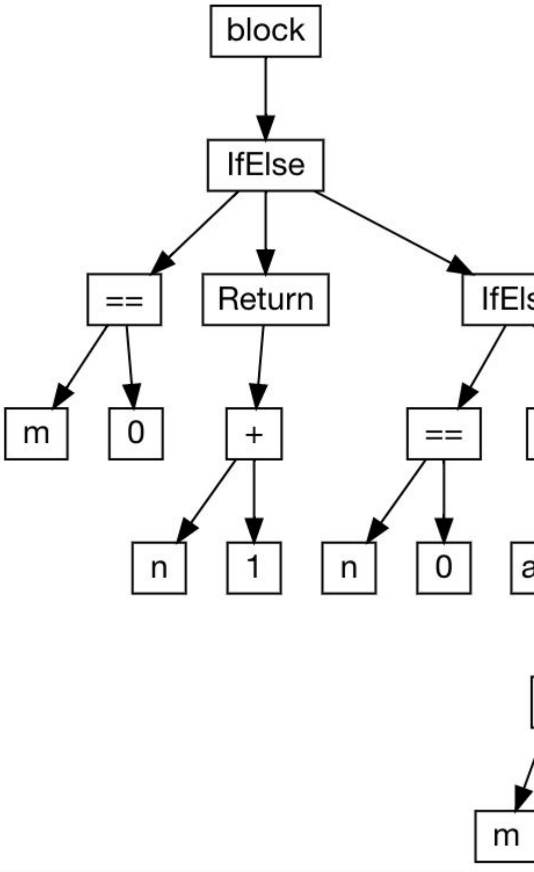
Modelling a Program

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

Making Inferences and Optimizations

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)))
            )
        )
    )
)
```



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));
}
```

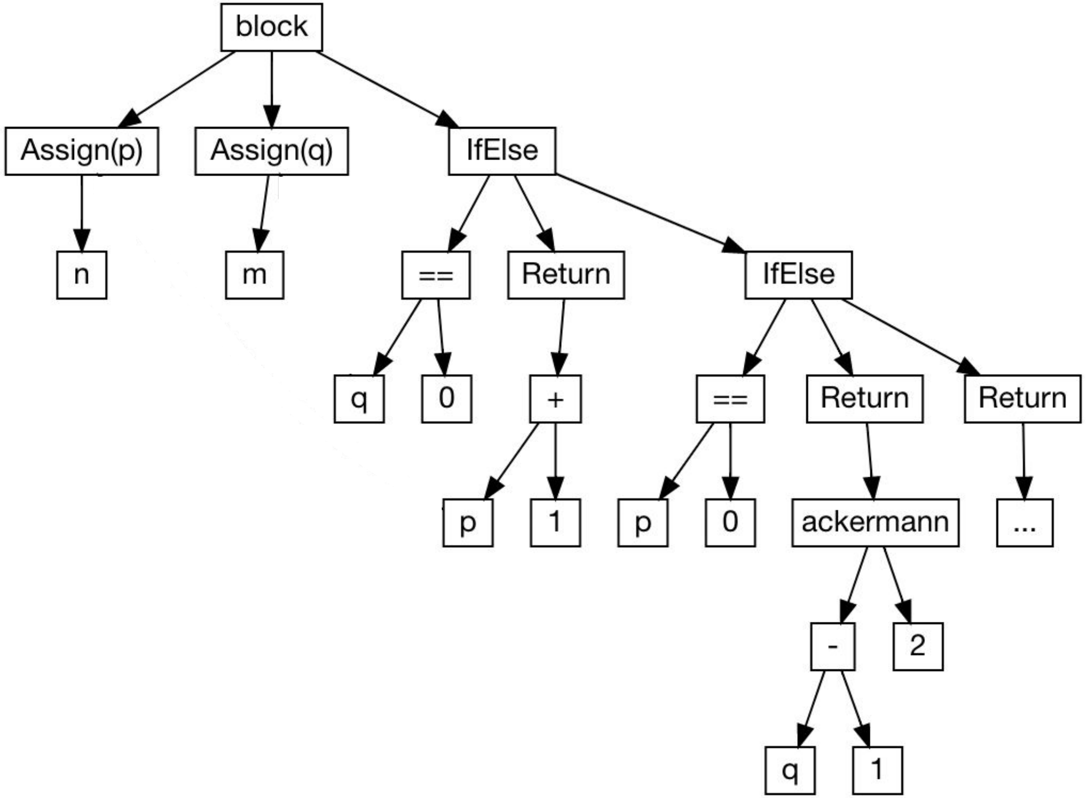
Abstract Syntax Trees

```
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)))
                )
            )
        )
```

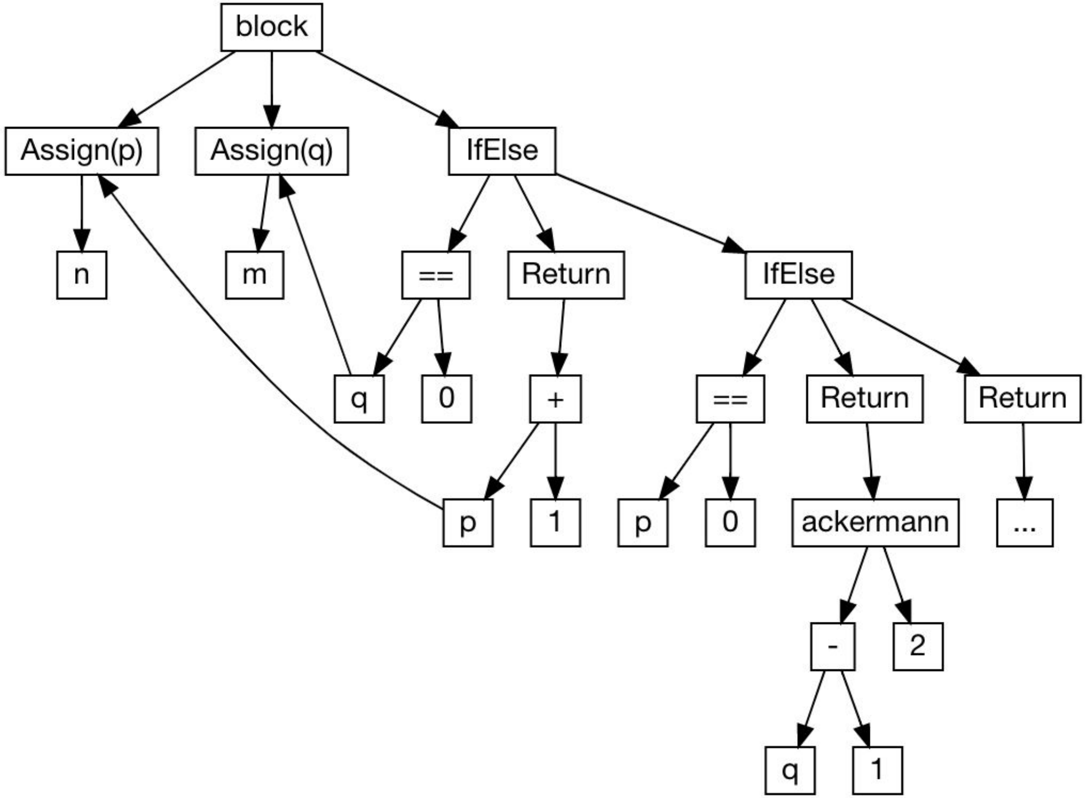
Abstract Syntax Trees

```
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)))
                )
            )
        )
```

Abstract Syntax Trees



Abstract Syntax Trees



How an Optimizing Compiler Works

Hand Optimizing Some Code

Modelling a Program

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

Making Inferences and Optimizations

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
```

```
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));
}
```

```
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));
}
```

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));
}
```

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|	|

```
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));
}
```

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));
}
```

How an Optimizing Compiler Works

Hand Optimizing Some Code

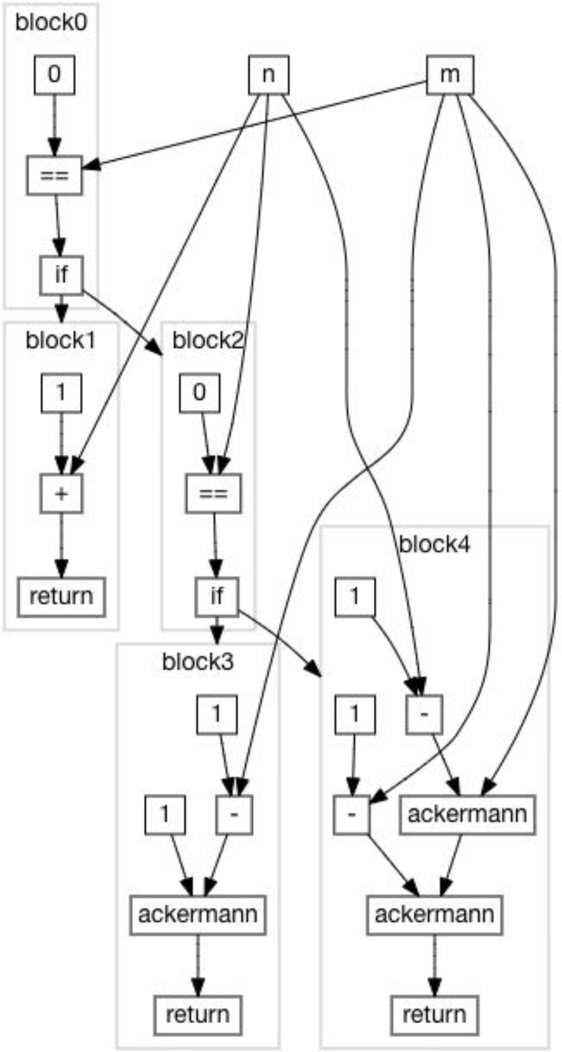
Modelling a Program

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

Making Inferences and Optimizations

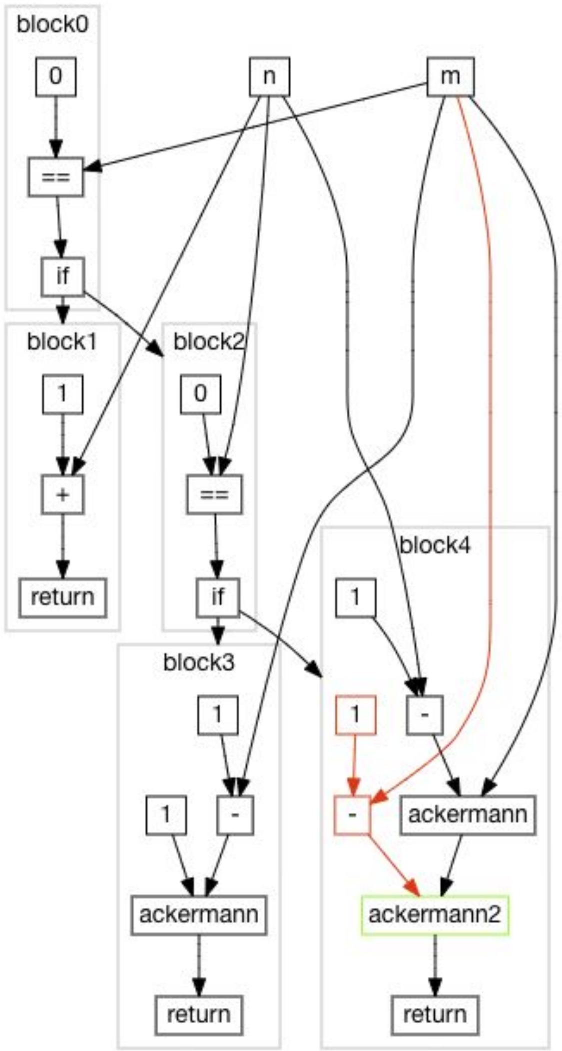
Dataflow Graphs

```
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

```
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));
}
```



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

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

Type Inference & Constant Folding

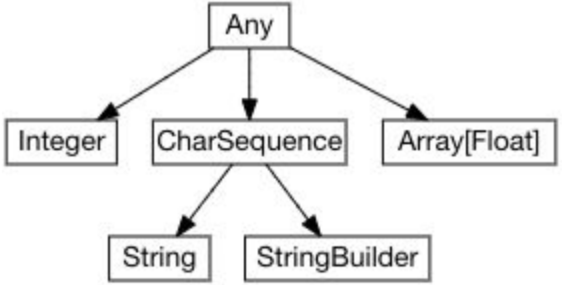
What do we know about a value?

- Is it an Integer? String? Array[Float]? PrintLogger?

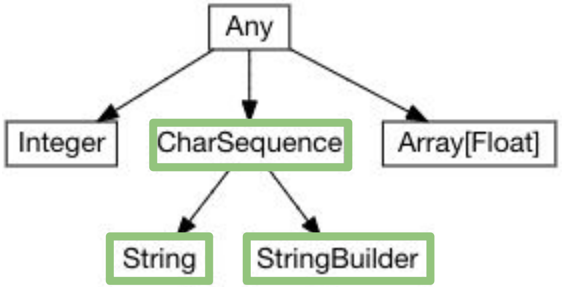
- Is it a CharSequence, which could be either a String or a StringBuilder?

- Is it Any, meaning we don’t know anything about it?

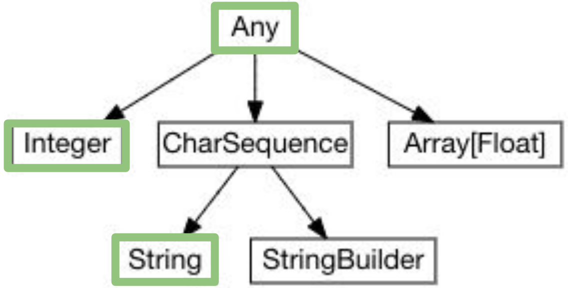
Type Lattices



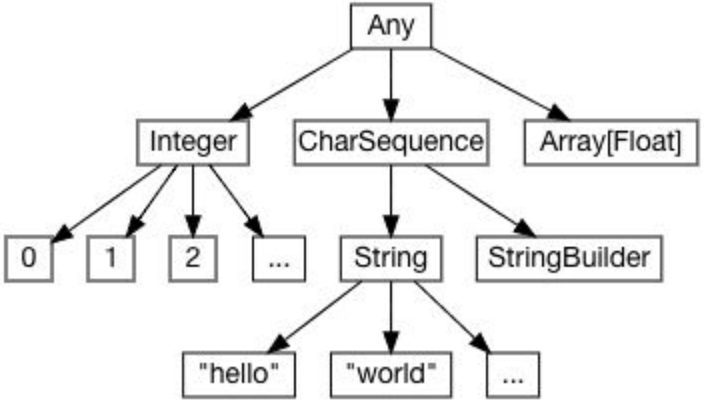
Type Lattices



Type Lattices

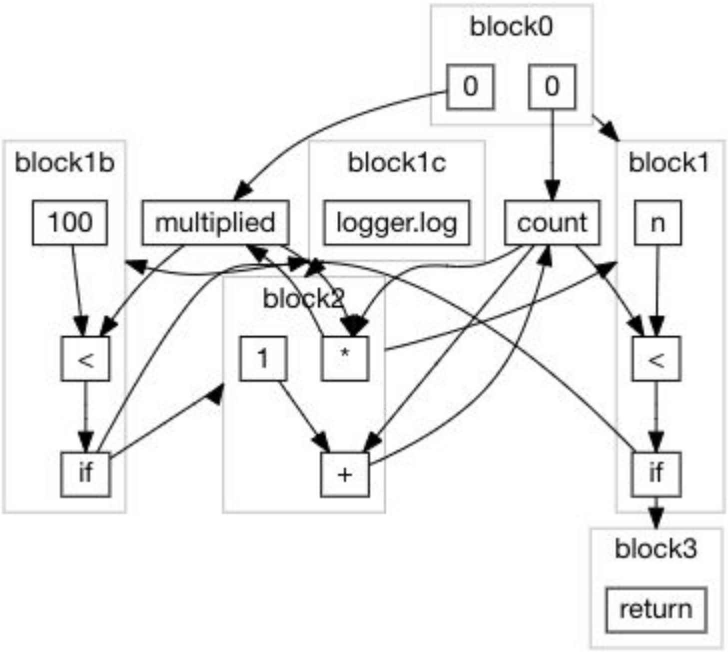


Type Lattices



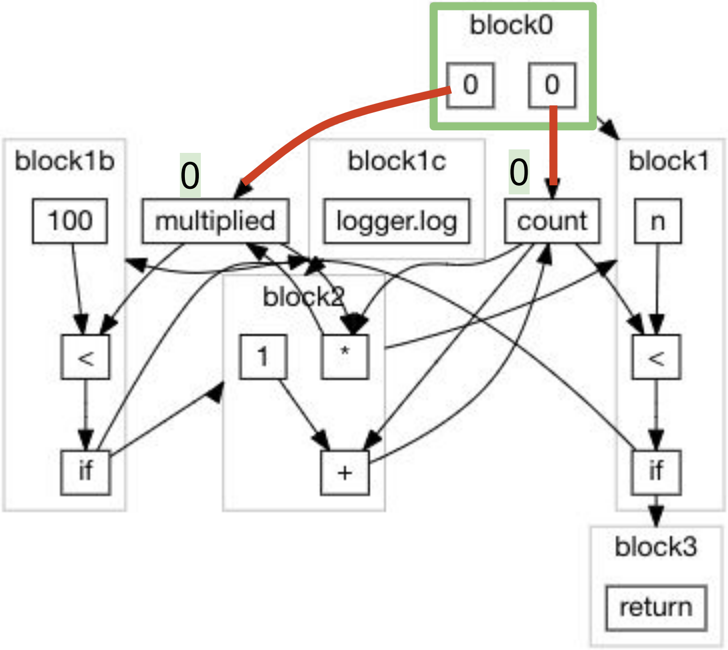
Inferring Values on the Dataflow Graph

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



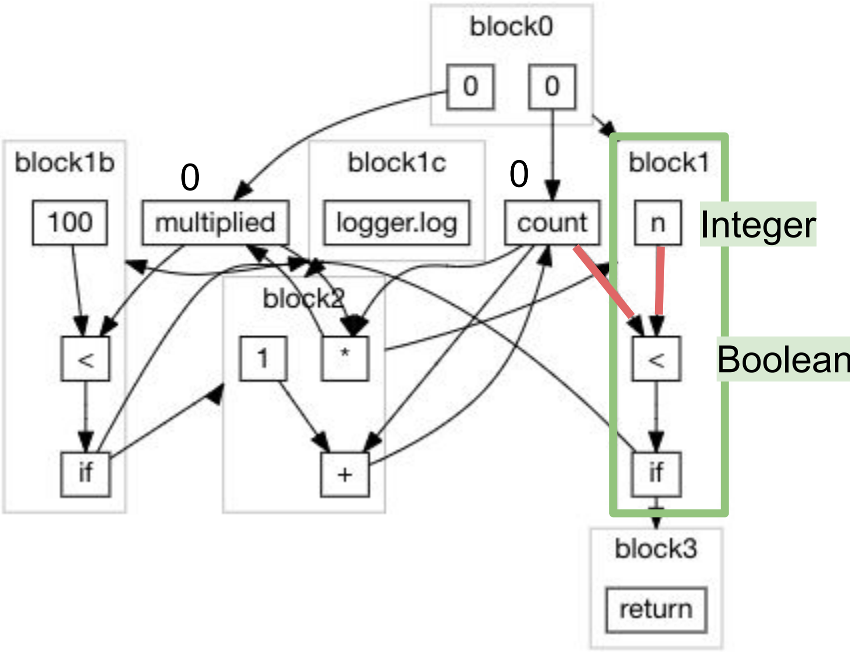
Inferring Values on the Dataflow Graph

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



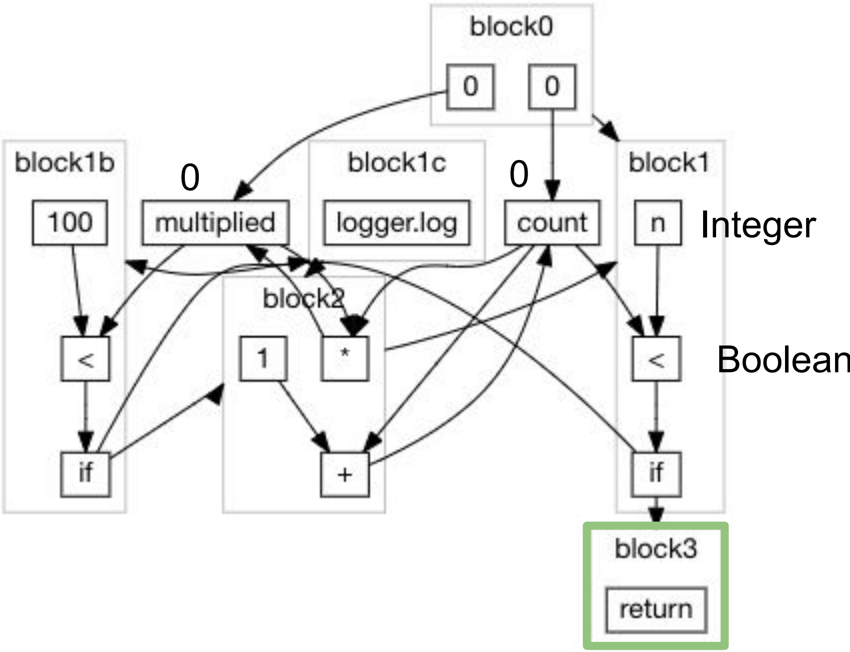
Inferring Values on the Dataflow Graph

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



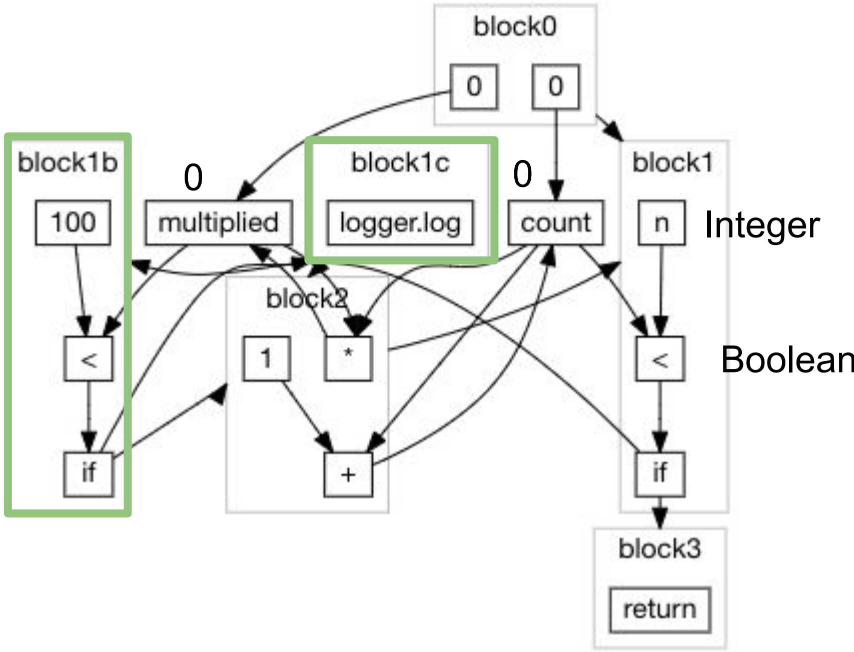
Inferring Values on the Dataflow Graph

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



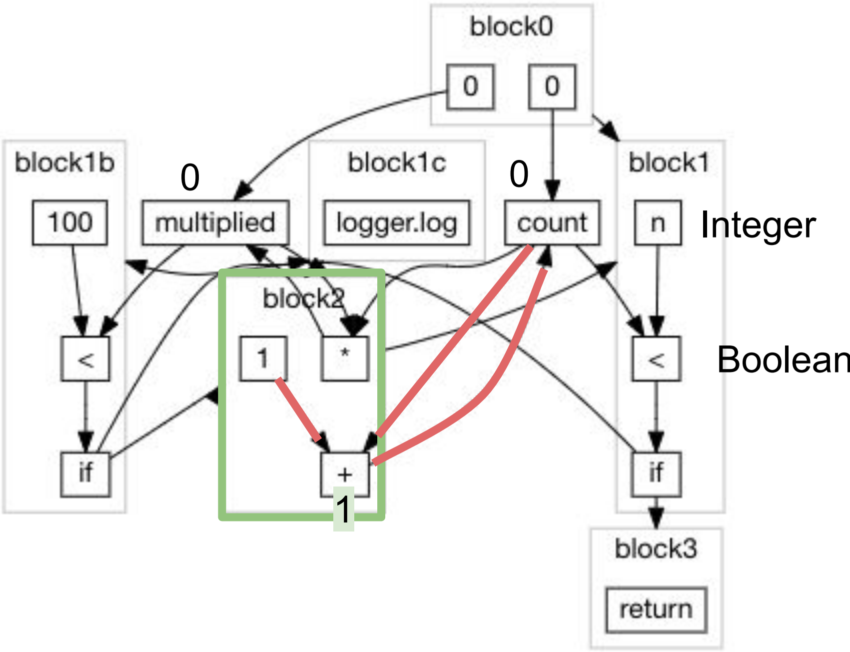
Inferring Values on the Dataflow Graph

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



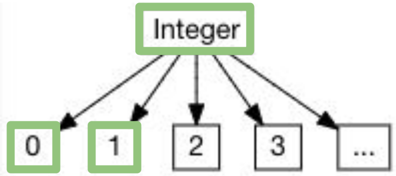
Inferring Values on the Dataflow Graph

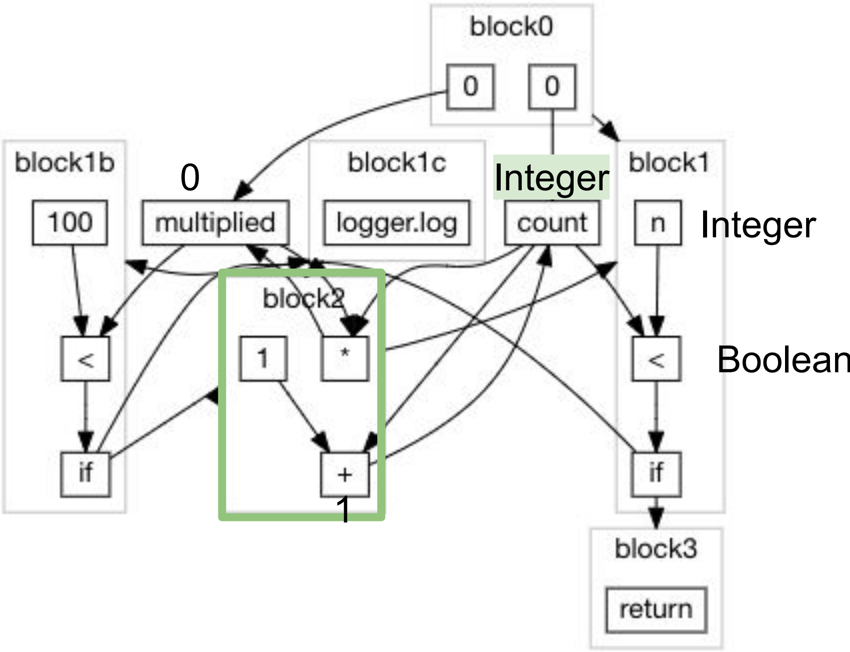
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```



Inferring Values on the Dataflow Graph

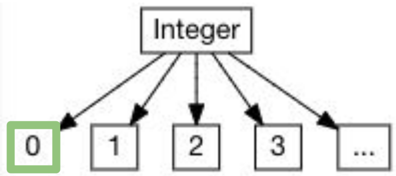
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```

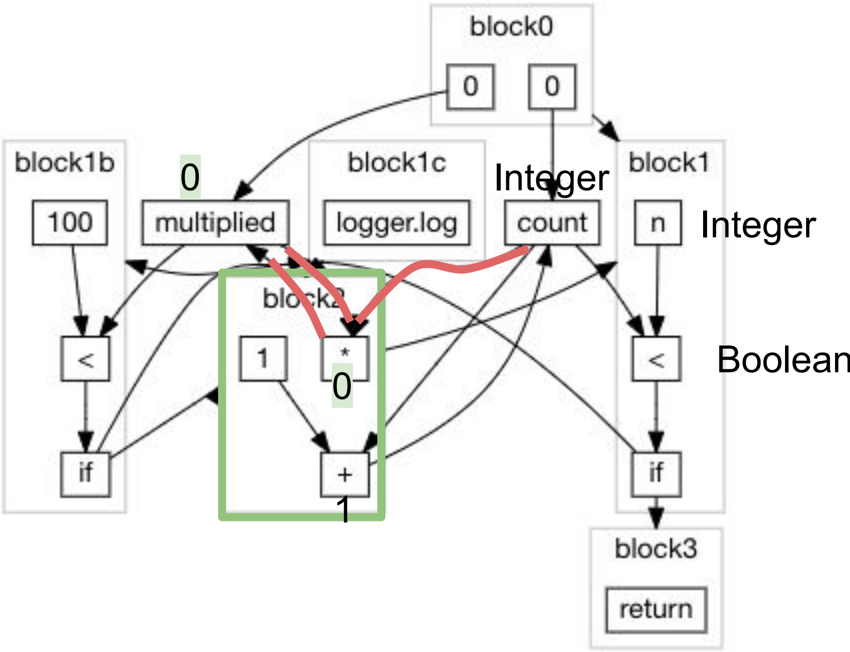




Inferring Values on the Dataflow Graph

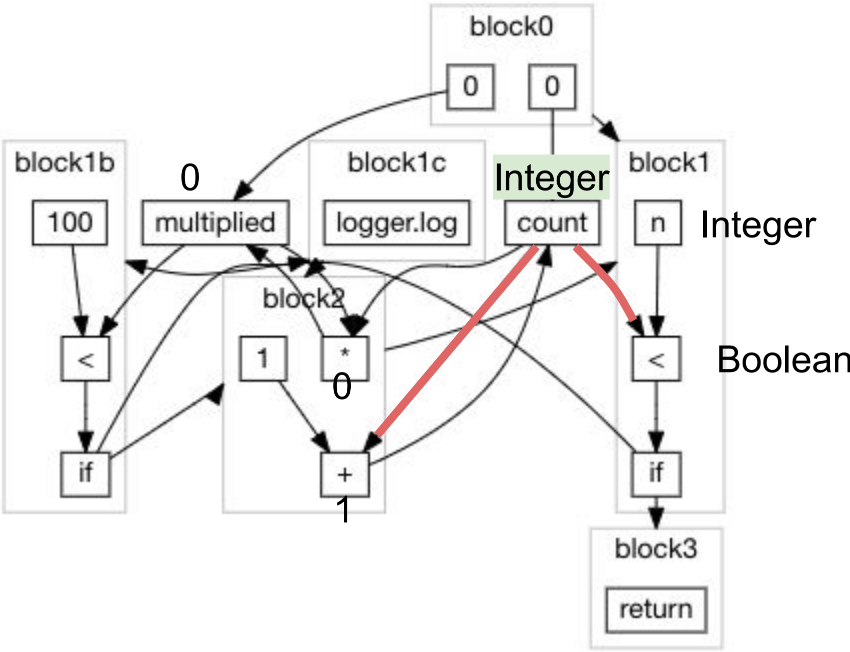
```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```





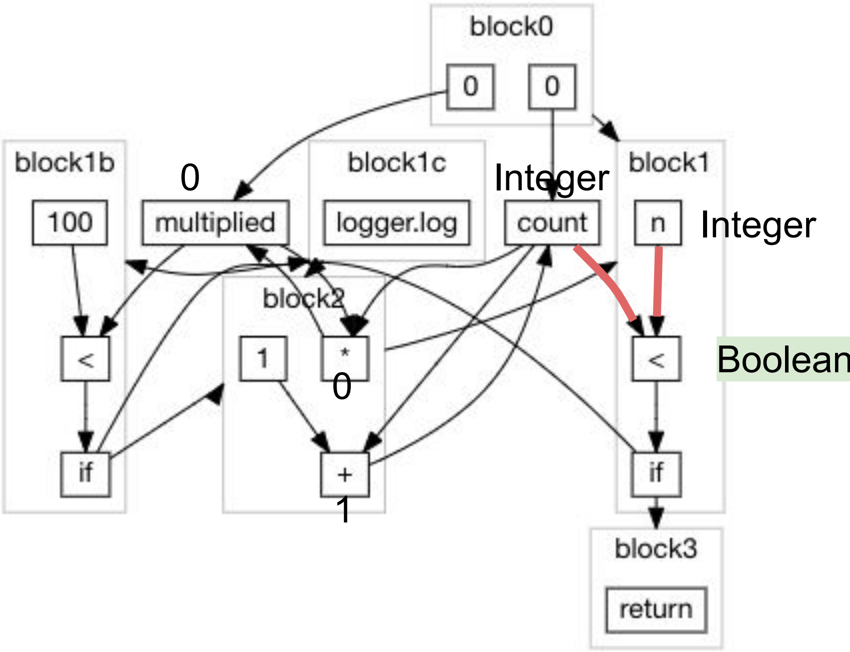
Inferring Values on the Dataflow Graph

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



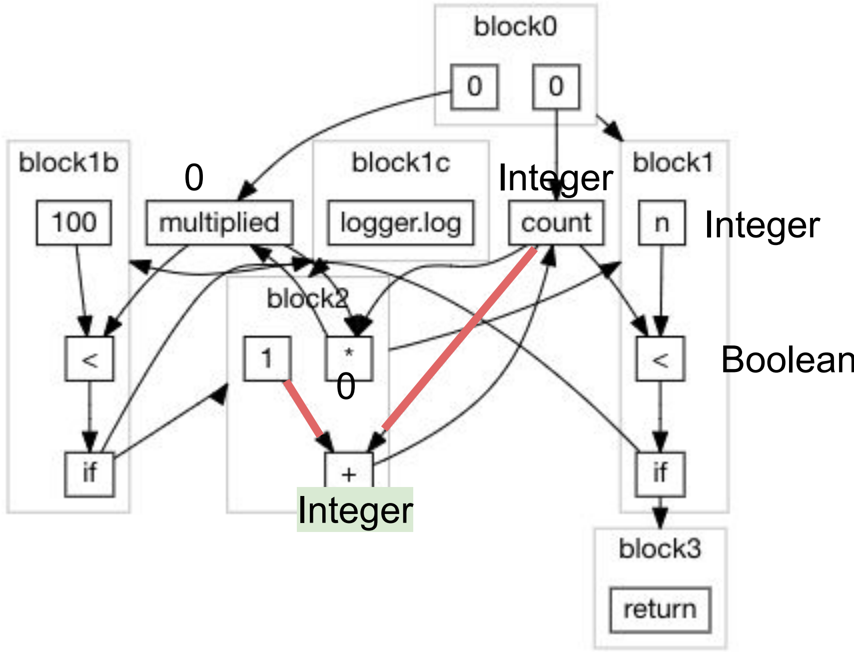
Inferring Values on the Dataflow Graph

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



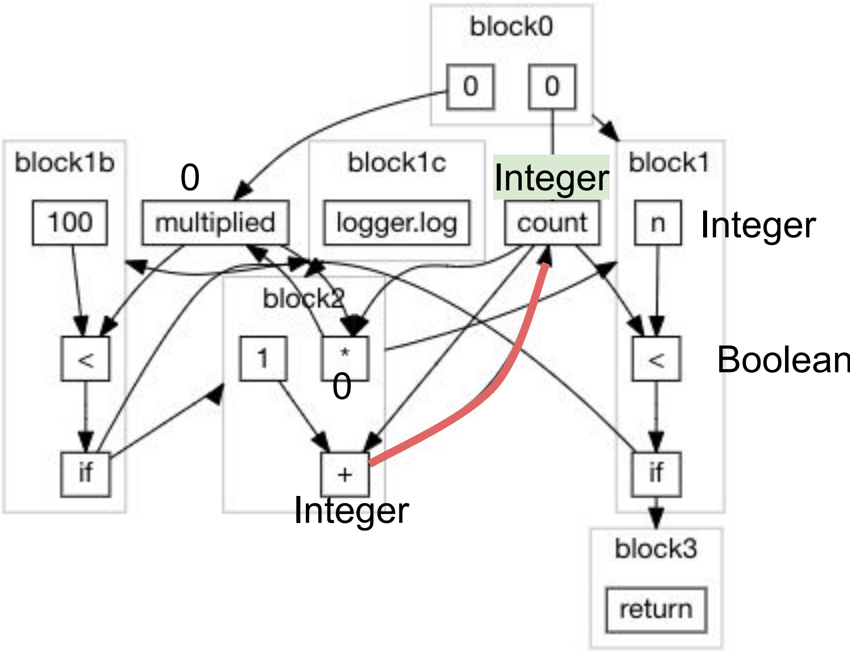
Inferring Values on the Dataflow Graph

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



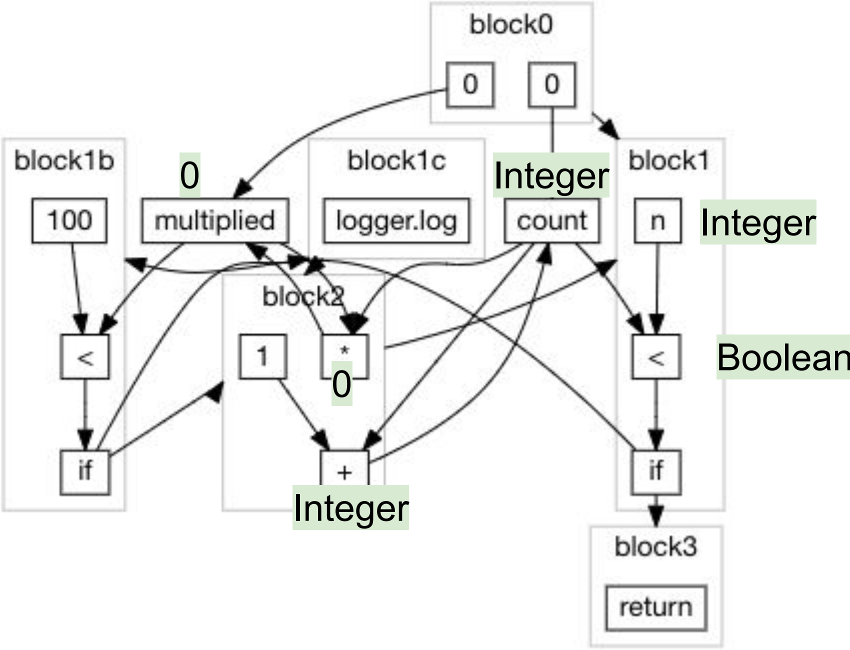
Inferring Values on the Dataflow Graph

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



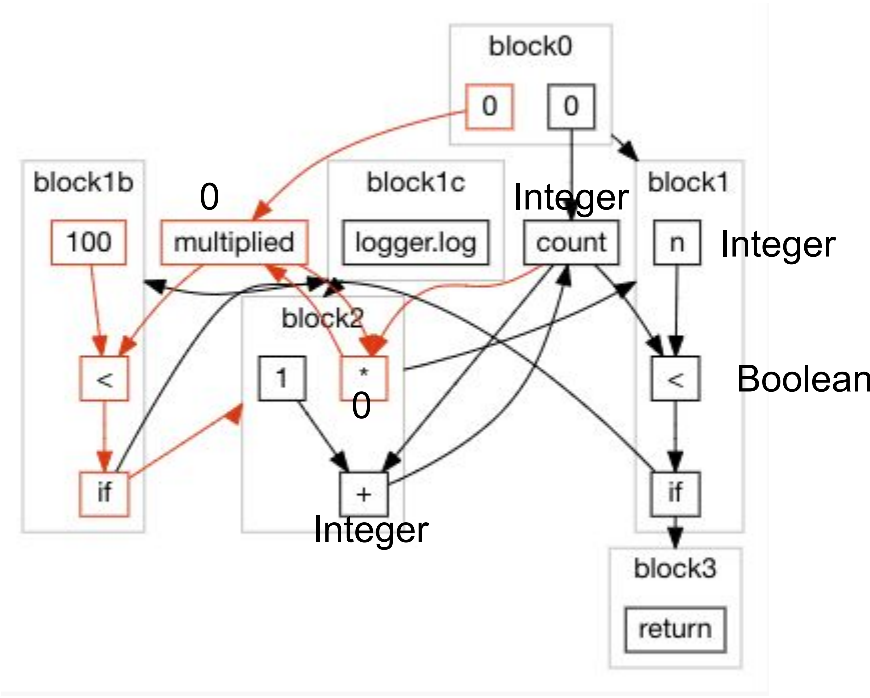
Inferring Values on the Dataflow Graph

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



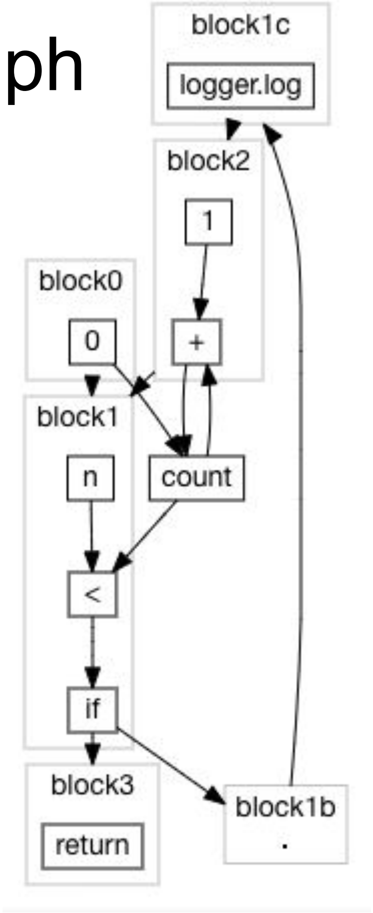
Inferring Values on the Dataflow Graph

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



Inferring Values on the Dataflow Graph

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



How an Optimizing Compiler Works

Hand Optimizing Some Code

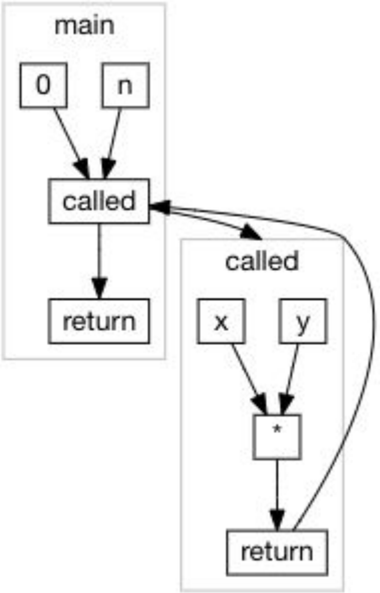
Modelling a Program

Inferences and Optimizations

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

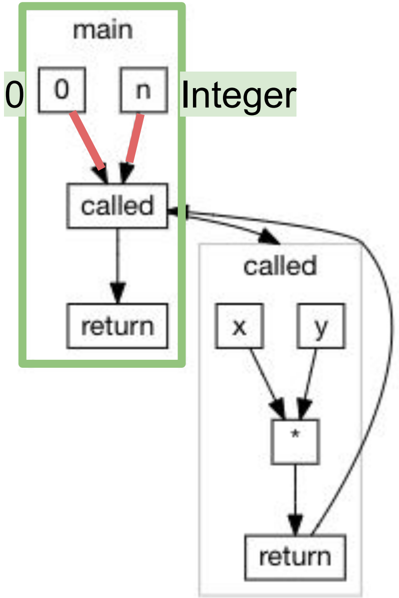
Inter-Procedural Inference

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

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



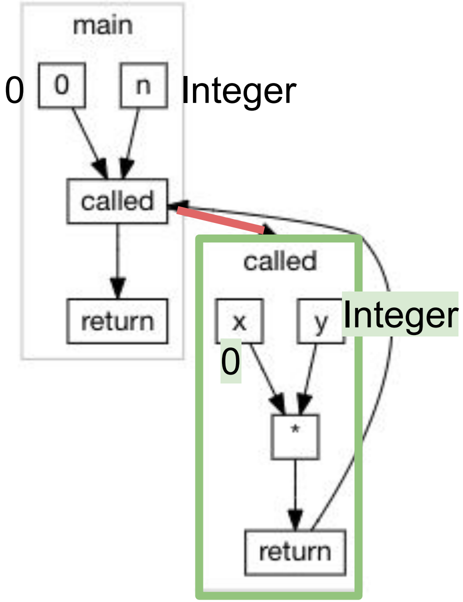
Inter-Procedural Inference

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

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



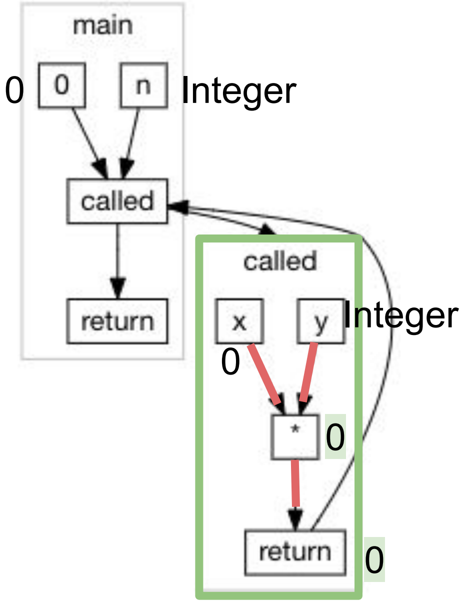
Inter-Procedural Inference

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

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



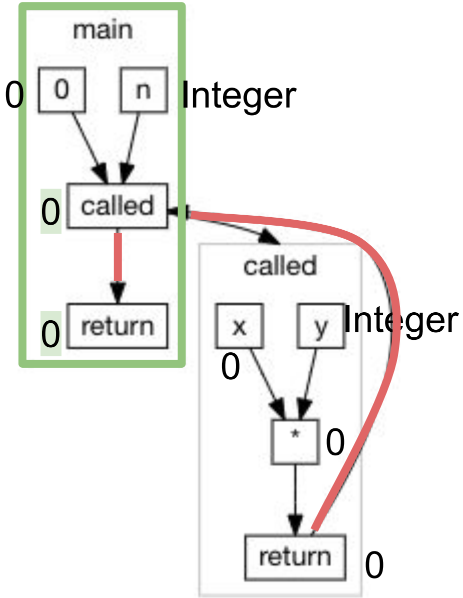
Inter-Procedural Inference

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

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



Inter-Procedural Inference

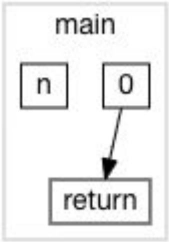
```
static int main(int n){
    return called(0, n);
}

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



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;
}
```



How an Optimizing Compiler Works

Hand Optimizing Some Code

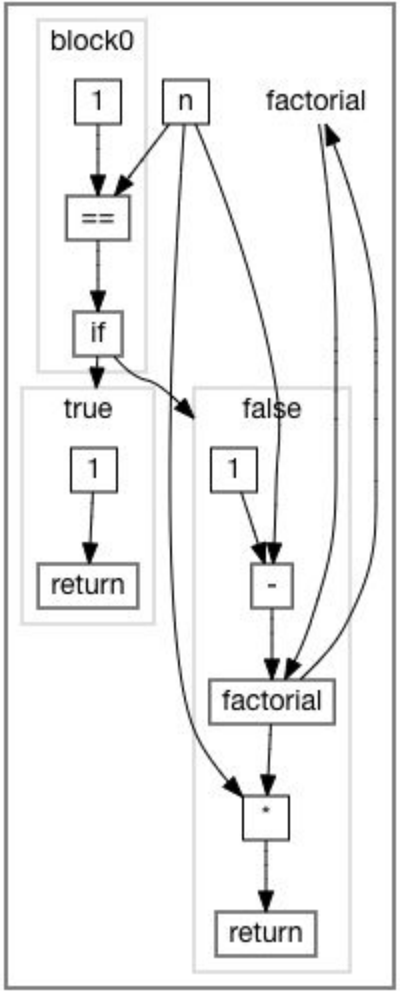
Modelling a Program

Inferences and Optimizations

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

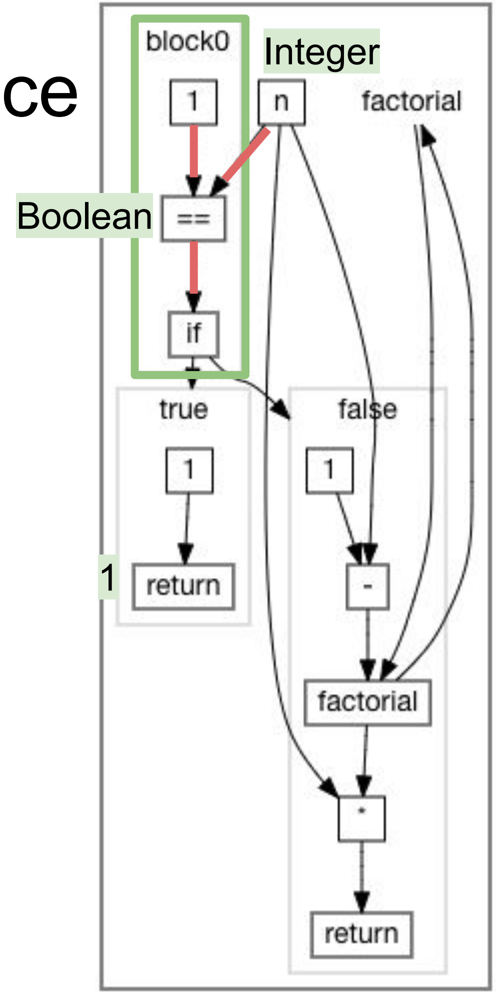
Recursive Inter-Procedural Inference

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



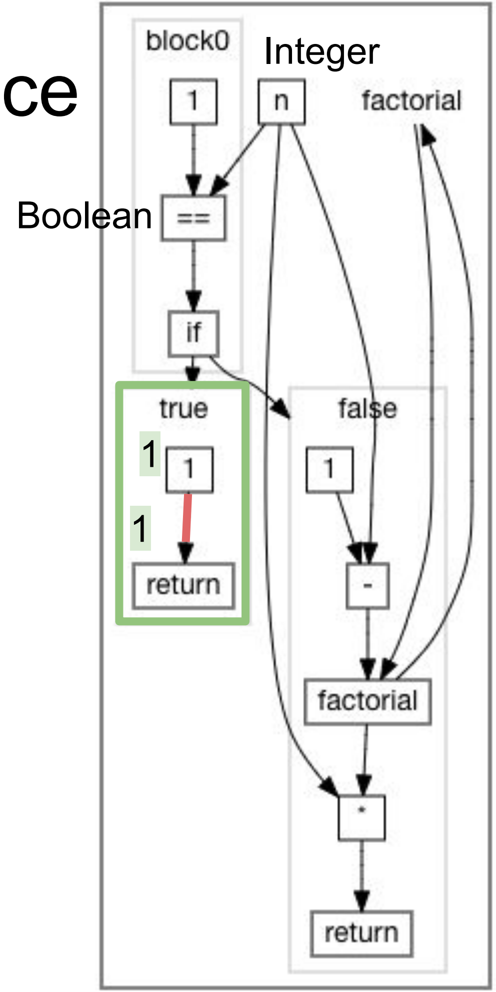
Recursive Inter-Procedural Inference

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



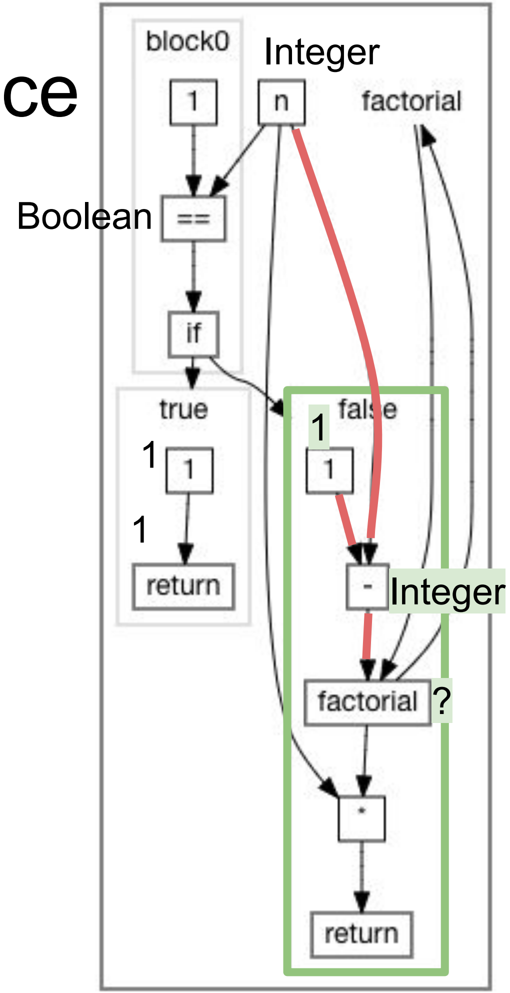
Recursive Inter-Procedural Inference

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



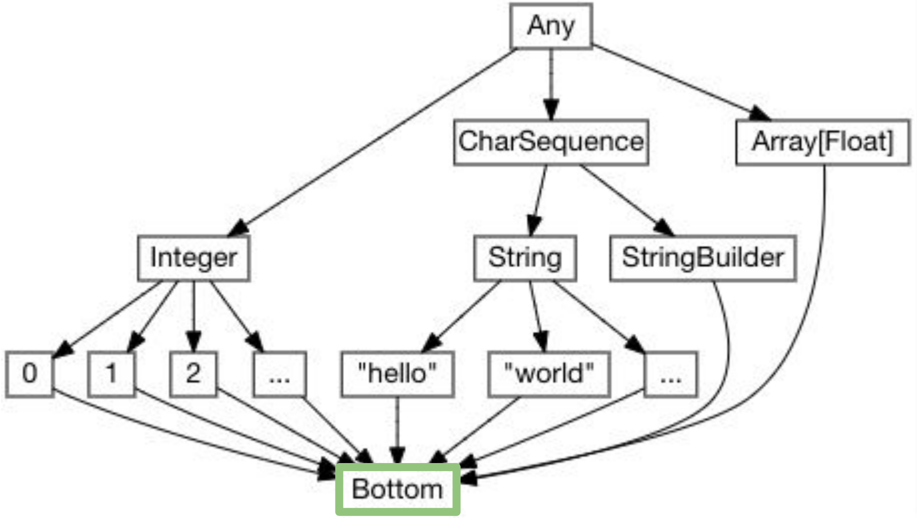
Recursive Inter-Procedural Inference

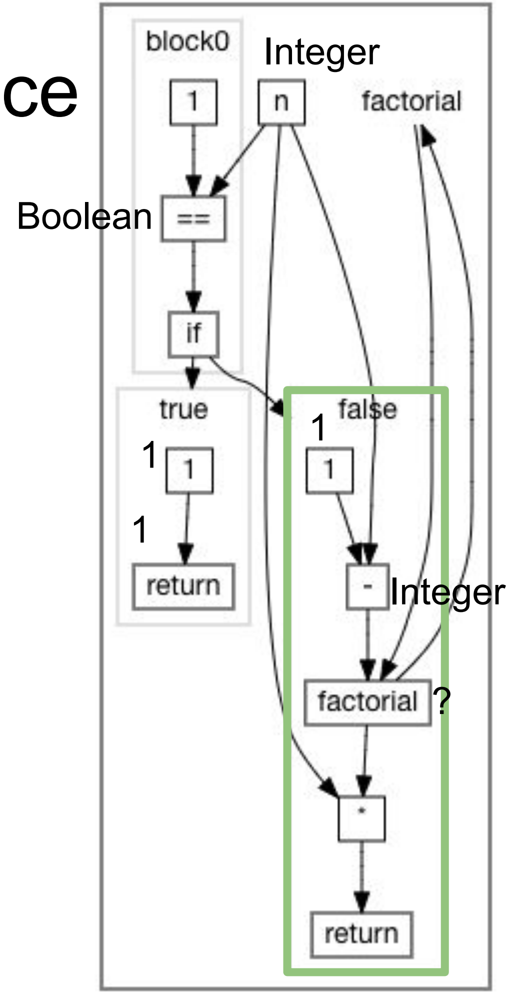
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```



Recursive Inter-Procedural Inference

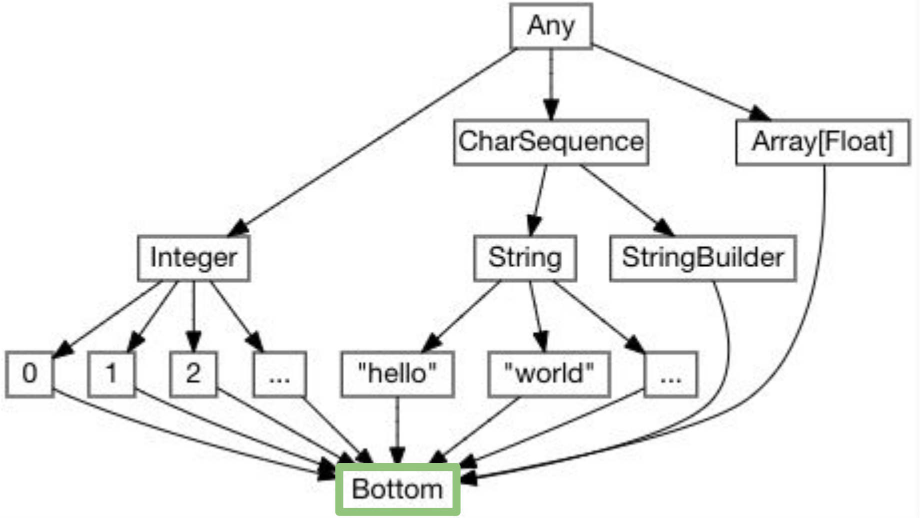
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

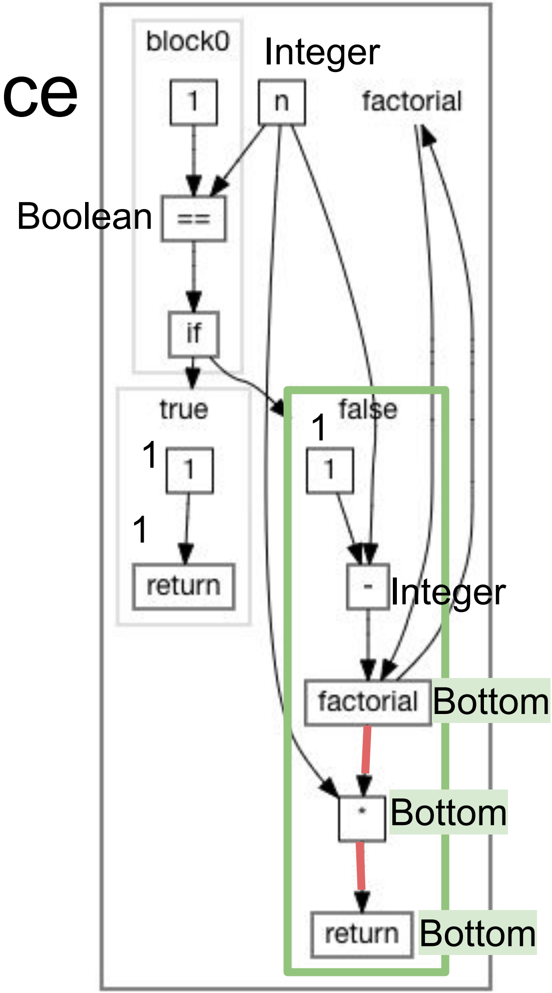




Recursive Inter-Procedural Inference

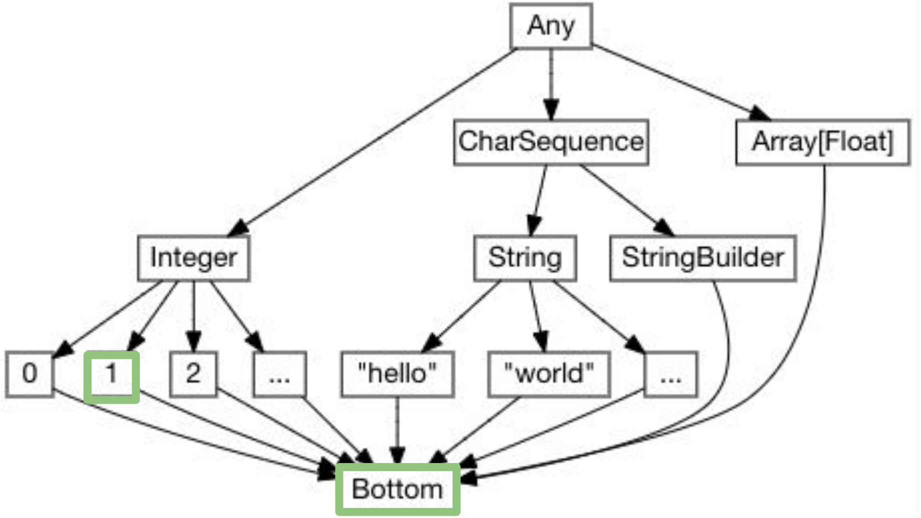
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

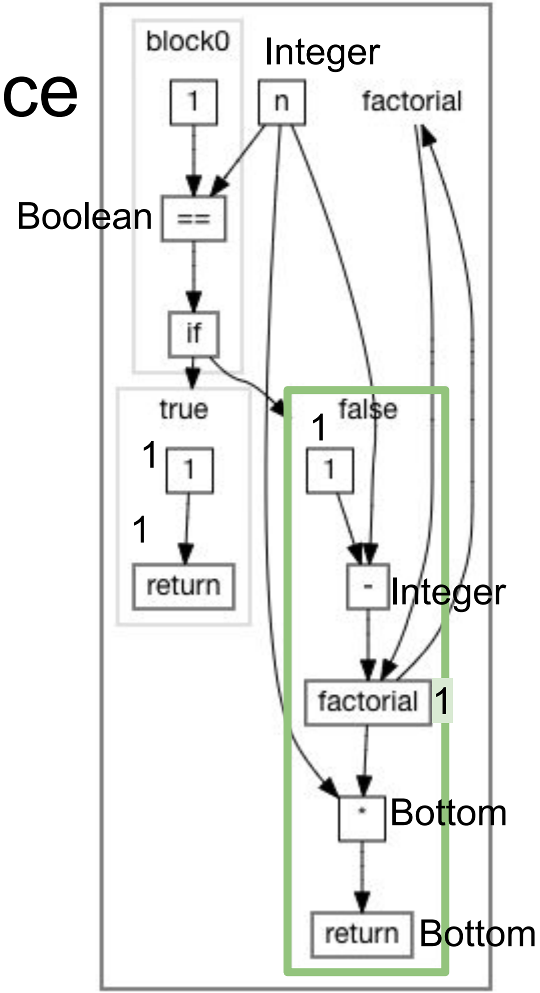




Recursive Inter-Procedural Inference

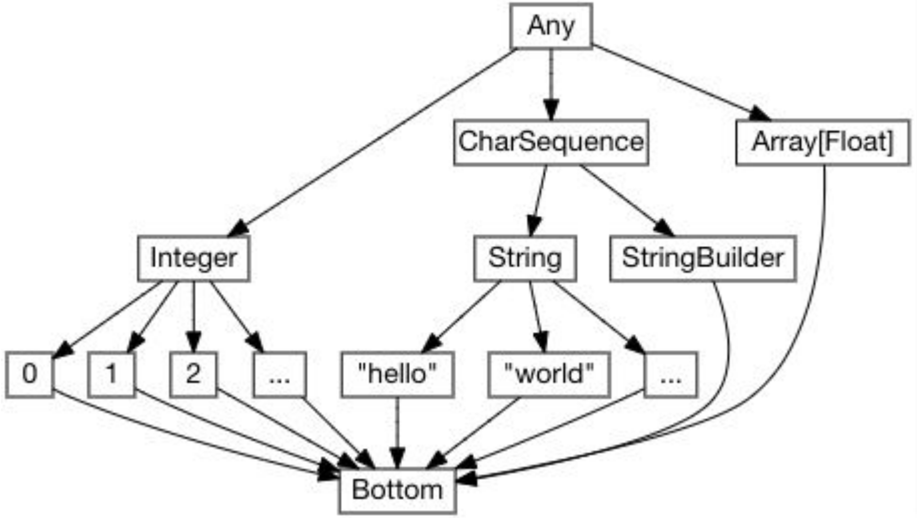
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

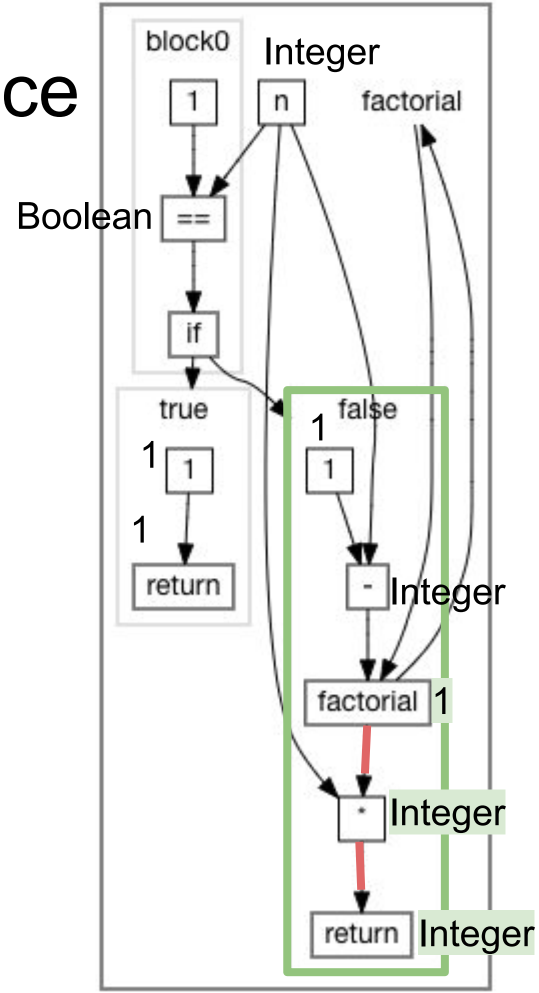




Recursive Inter-Procedural Inference

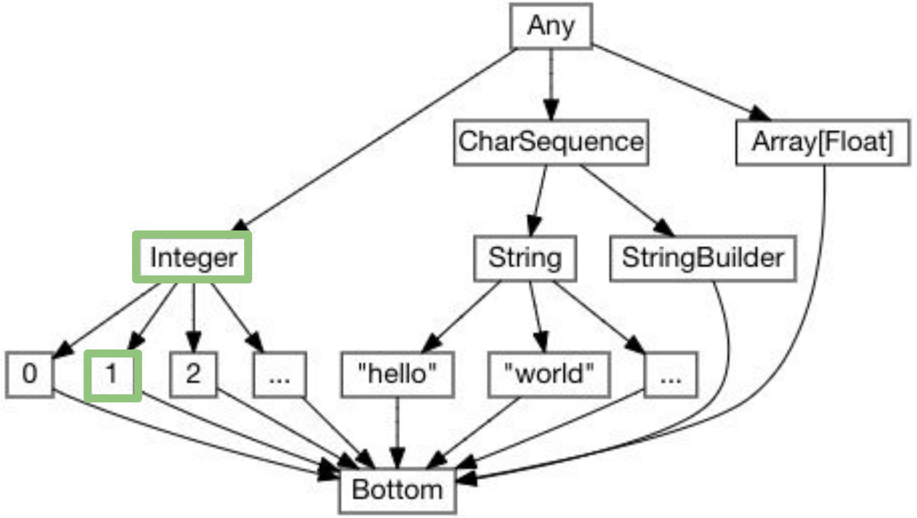
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

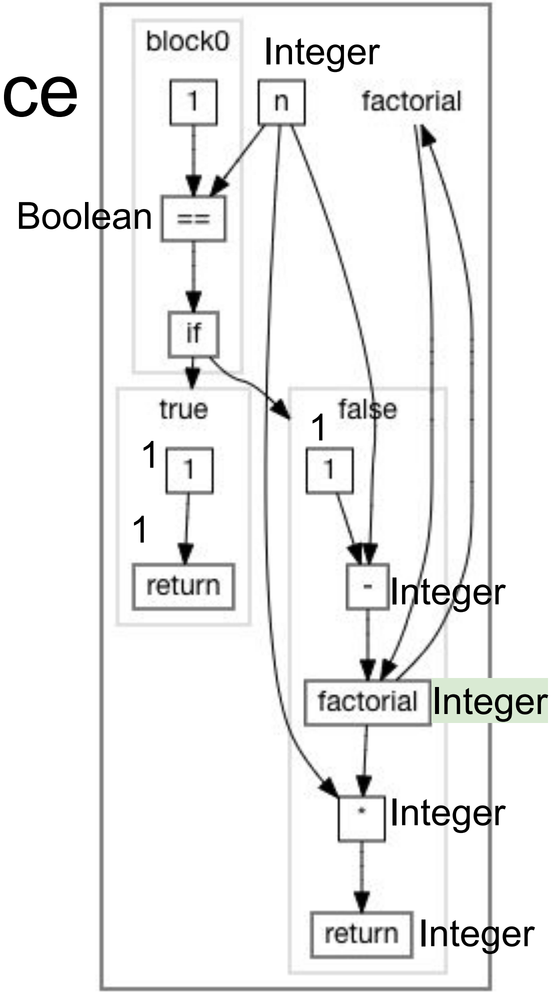




Recursive Inter-Procedural Inference

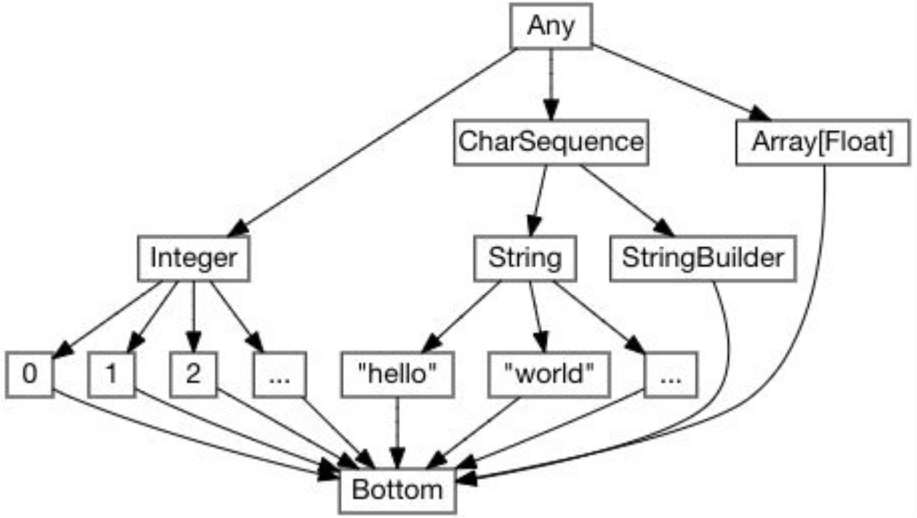
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

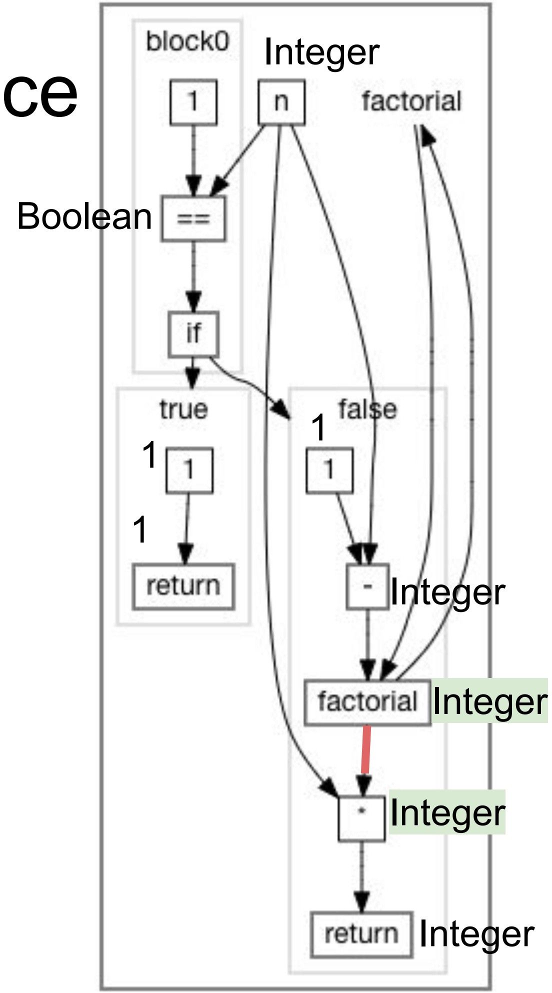




Recursive Inter-Procedural Inference

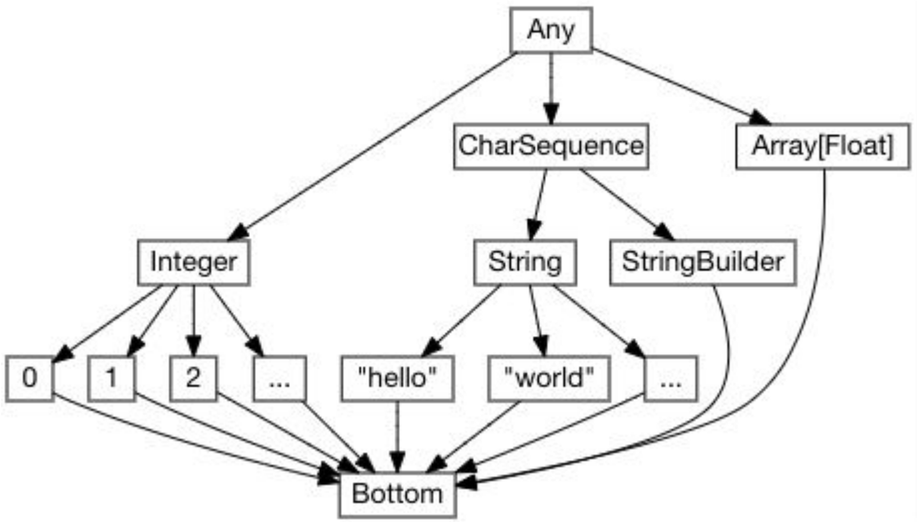
```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

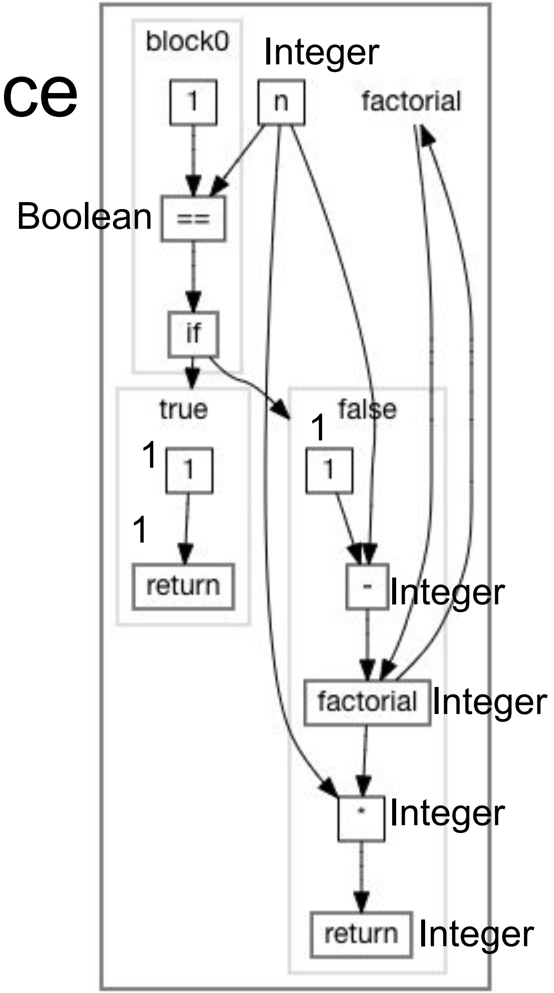




Recursive Inter-Procedural Inference

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





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**

Liveness & Reachability Analysis

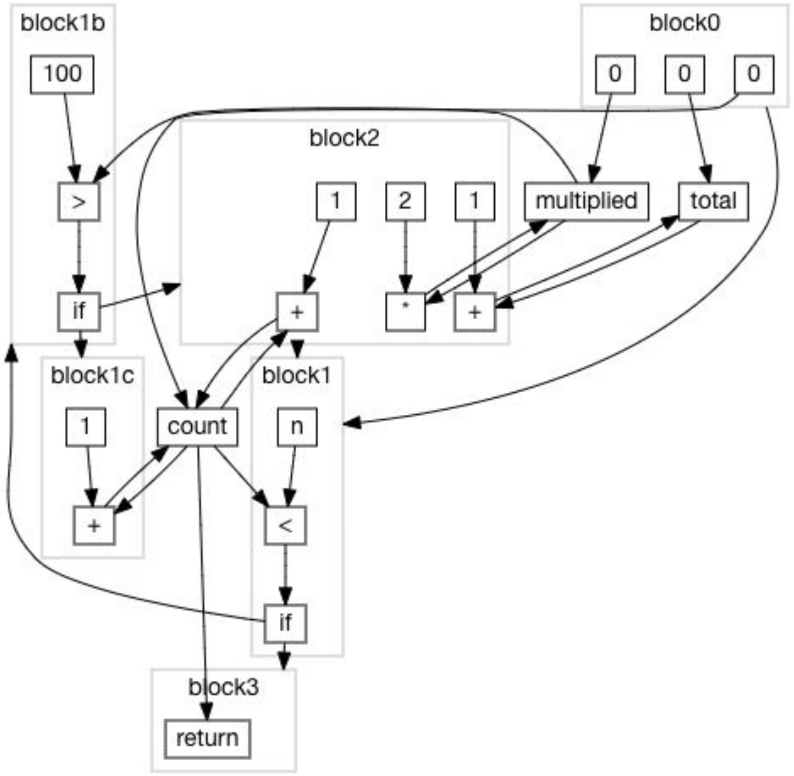
Find all the code whose values contribute to the final returned result ("Live")

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!

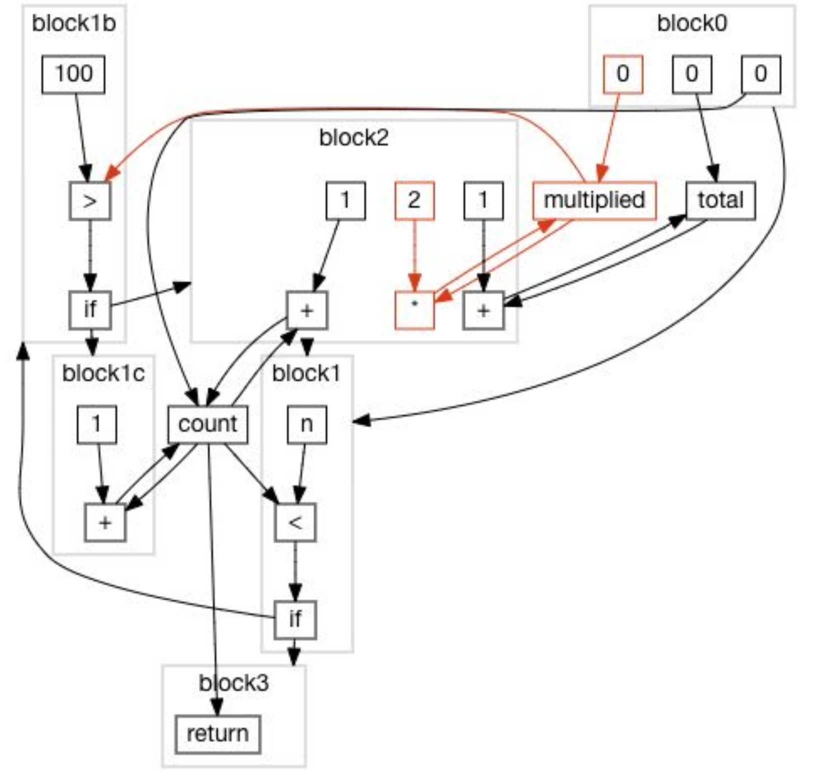
Strawman Program

```
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;
}
```



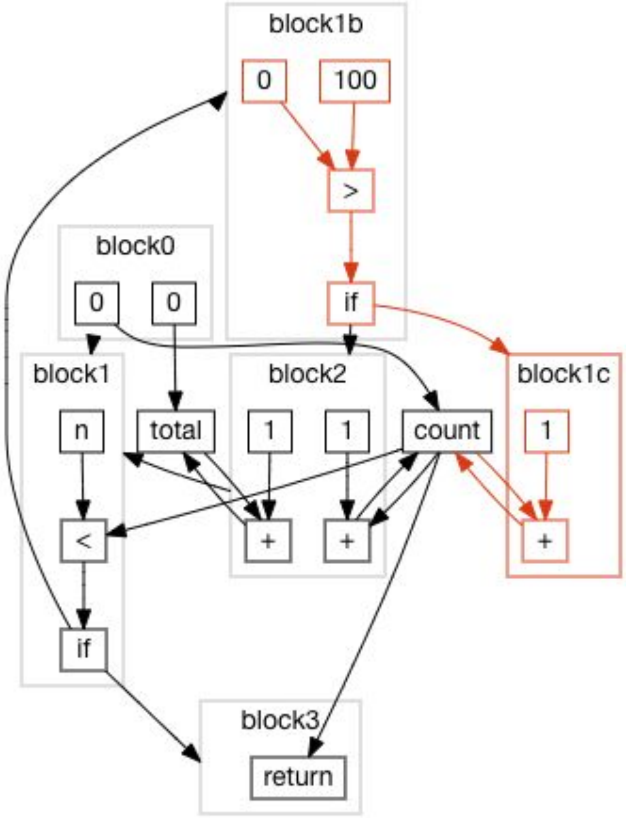
Type Inference & Constant Folding

```
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;
}
```



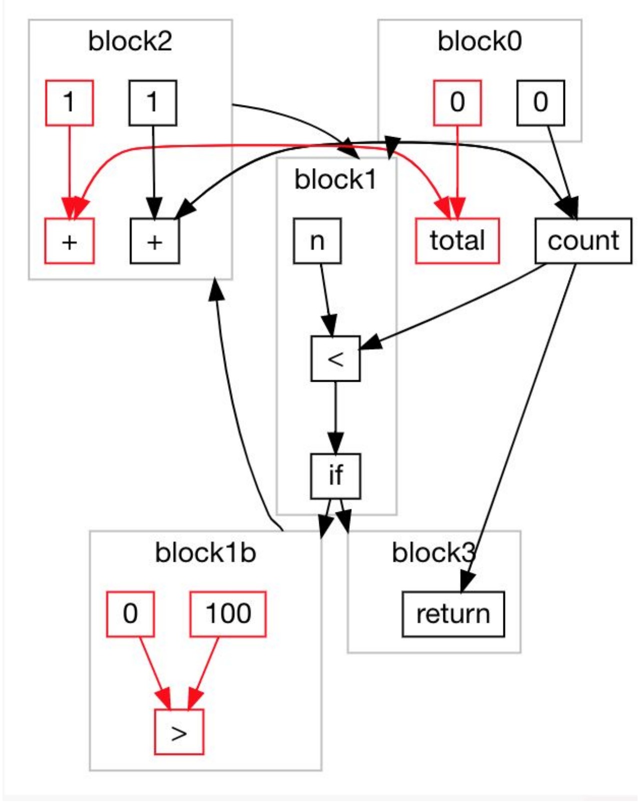
Branch Elimination & Reachability Analysis

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



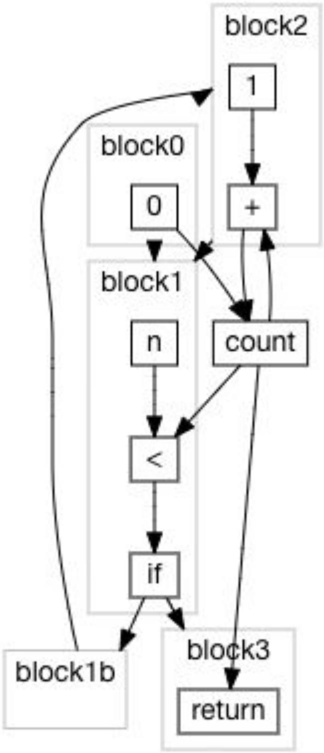
Liveness Optimizations

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



Final Output Code

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



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

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