**Postfix, Eager, Translation of Boolean Operators**

Eager function argument = argument that is always evaluated in the function

In Java and many programming languages, all function arguments are eager

In [Haskell](http://en.wikipedia.org/wiki/Haskell), all function arguments are **lazy** by default - evaluated only if they are used in the given computation

In Scala, argument is lazy by default if declared with type T, but can be made lazy by declaring them by ⇒ .

For example:

[def](http://scala-lang.org) ifNotE(c : Boolean, s : Unit) : Unit = {

[if](http://scala-lang.org) (!c) {

s

} [else](http://scala-lang.org) { println("Does it pay off to be lazy?") }

}

[var](http://scala-lang.org) x = 0

IfNot(x==0, 3/x) // division by zero

How to prevent 3/x from being evaluated?

[def](http://scala-lang.org) ifNot(c : Boolean, s : => Unit) : Unit = {

[if](http://scala-lang.org) (!c) {

s

} [else](http://scala-lang.org) { println("Does it pay off to be lazy?") }

}

IfNotE(x==0, 3/x) // no problem

**Bitwise Operators vs Logical Connectives**

These are logical connectives, they operate on Booleans:

&& ||

These are bitwise operators:

& |

Example:

10110

& 11011

= 10010

10110

| 11011

= 11111

Note: when the arguments are 0,1 then the *truth* table is same for && and for &. Similarly, same for || as for |. However, laziness is not the same.

Truth tables:   
\begin{displaymath}
\begin{array}{c|cc}
\land & 0 & 1\\ \hline
    0 & 0 & 0\\
    1 & 0 & 1
\end{array}
\end{displaymath}

\begin{displaymath}
\begin{array}{c|cc}
\lor & 0 & 1\\ \hline
    0 & 0 & 1\\
    1 & 1 & 1
\end{array}
\end{displaymath}

**Compiling Bitwise Operations**

[[ e1 & e2 ]] =

[[ e1 ]]

[[ e2 ]]

iand

[[ e1 | e2 ]] =

[[ e1 ]]

[[ e2 ]]

ior

This indeed works for &, | (note: single sign, not && or ||)

Java code:

static boolean test(boolean x, boolean y) {

return (x & y);

}

static boolean test(boolean, boolean);

Code:

0: iload\_0

1: iload\_1

2: iand

3: ireturn

**Effect of Eager Evaluation**

**Example 1**

What does the following Java program do:

class Test {

static boolean bigFraction(int x, int y) {

return ((y==0) | (x/y > 100));

}

public static void main([String](http://www.google.com/search?hl=en&q=allinurl%3AString+java.sun.com&bntI=I%27m%20Feeling%20Lucky)[] args) {

boolean is = bigFraction(10,0);

}

}

Answer:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Test.bigFraction(Test.java:4)

at Test.main(Test.java:19)

**Example 2**

What does the following Java program do:

class Test {

static int iterate() {

int[] a = new int[10];

int i = 0;

int res = 0;

while ((i < a.length) & (a[i] >= 0)) {

i = i + 1;

res = res + 1;

}

return res;

}

public static void main([String](http://www.google.com/search?hl=en&q=allinurl%3AString+java.sun.com&bntI=I%27m%20Feeling%20Lucky)[] args) {

[System](http://www.google.com/search?hl=en&q=allinurl%3ASystem+java.sun.com&bntI=I%27m%20Feeling%20Lucky).out.println(iterate());

}

}

Answer:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10

at Test.iterate(Test.java:16)

at Test.main(Test.java:25)

**Lazy Versions of Logical Operators**

To enable code patterns similar to above, instead of bitwise operators

| &

we use operators

|| &&

The semantics of Java requires: when evaluating

* (p && q), if p is false, then **q is not evaluated**
* (p || q), if p is true, then **q is not evaluated**

This makes sense because of truth tables (expressions have not only truth values, but also side effects):

\begin{displaymath}
\begin{array}{c|cc}
\land & 0 & 1\\ \hline
    0 & 0 & 0\\
    1 & 0 & 1
\end{array}
\end{displaymath}

\begin{displaymath}
\begin{array}{c|cc}
\lor & 0 & 1\\ \hline
    0 & 0 & 1\\
    1 & 1 & 1
\end{array}
\end{displaymath}

Because we have explained the evaluation of && and || using ‘if’, we show first how to compile the conditional expression ‘?’, that is, expression

if (c) then c1 else c2

where c,c1,c2 are all of type Boolean.