Short-Circuiting

By Blain Judkins and Kylee Friederichs

What is short-circuiting?

Short circuiting occurs when given a boolean expression, the outcome of the operation being performed can be determined by the value of the first sub-expression without having to consider the value of the second sub-expression. When this happens, the compiler is able/allowed to essentially “skip” evaluating the second sub-expression and produce the proper boolean result. Occurrences of short circuiting appear in boolean expressions that use the ‘||” or ‘&&’ operators that represent logical “or” and logical “and,” respectively.

Why use short-circuiting?

In general, we say that it is advantageous to use short-circuiting in programming and/or evaluation because it can be helpful in mitigating or avoiding computationally expensive tasks.

As a simple example to demonstrate what we mean is that if you have a very long line of code like

```
if (true || false || false || false || false || false || false || true || false || false)
```

It may take a compiler a long time to look at every single boolean statement and operator, but we know that when evaluating a boolean expression with a logical “or” operator, if any part of the expression is true, then the whole expression evaluates to true. By looking at this particular example, if we can implement short-circuiting, then the program knows that by the very first boolean expression is true and can immediately evaluate the whole expression to true without having to consider the other expressions.

How short-circuiting can be tricky

Short-circuiting can be tricky and confusing in that it only works if things are implemented in a certain way; we call this “certain way” the “evaluation order” in which sub-expressions within an overall expression must be evaluated in a particular order to be properly functional. Evaluation order typically defines the order in which sub-expressions should be evaluated to avoid errors or bugs in the code, as some sub-expressions may be dependant on the value of a previously evaluated sub-expression; in the case of short-circuiting, we can use evaluation order to determine whether or not an expression is capable of short-circuting.

Example 1

```
def isOddNum(num: Int): Boolean = {
    if (num % 0 == 0) { false }
    else { true }
}
def foo() {
    if (isOddNum(24) && print(“I should not print or else short-circuit didn’t happen”) {
        print(“Hello”)
    }
    else {
        print(“World”)
    }
}
// Output: “World”
```

Example 2

SearchDivide1:

$$\frac{e2 \rightarrow e2'}{e1 \text{ divided by } e2'}$$

SearchDivide2:

$$\frac{e1 \rightarrow e1'}{e1' \text{ divided by } n2}$$

DoDivide:

$$\frac{n' = n1 \text{ divided by } n2}{n1 \text{ divided by } n2 \rightarrow n'}$$

This set of inference rules allows for short-circuiting by explicitly stating the evaluation order the code must follow when trying to implement a function that will calculate division of a number (n1) divided by another number (n2). In math, we know that we cannot divide by the number 0–it's the most important rule to keep in mind when dividing numbers. If our second number, n2, is 0, which is evaluating with SearchDivide1, we can tell the code to short-circuit before evaluating our expression with SearchDivide2 because there is no point in evaluating that sub-expression if we already know that our expression is already undefined, as we would be dividing by 0.