S4.1: Expression Evaluation: Eager and Lazy Evaluation

CSci 2041: Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

Exercise #1:

Above we saw that evaluation can be seen as a sequence of rewrites of the expression.

Using this approach, evaluate

```
a (3 + 4) (3 / 0)
```

where

```
a x y = x + 3
```

using

1. call-by-name semantics, and then
2. call-by-value semantics

Exercise #2:

Using call-by-value evaluation and then lazy evaluation, write out the first 10 steps of the evaluation of the following:
`take 2 (makefrom 4 5)`

Use the following definitions (clearly not OCaml syntax):

```
take n [] = []
take 0 (x::xs) = []
take n (x::xs) = x::take (n-1) xs

makefrom 0 v = []
makefrom n v = v :: makefrom (n-1) (v+1)
```

Exercise #3:

Write a function named `cond` that has the same behavior as OCaml's if-then-else construct.

Exercise #4:

Evaluate the first dozen steps or so of the following:

```
sum (take 3 (squares_from 1))
```

where

```
sum [] = 0
sum x::xs -> x + sum xs

squares_from v = v*v :: squares_from (v+1)
```

Exercise #5:

Define an infinite list containing all natural numbers, starting at 0. Call this `allnats`. Recall our definition of squares:

```
 let allnats = let ns n = n :: ns (n+1) in ns 0
```

Then evaluate the first 10 steps of

```
drop 3 allnats
```

where

```
let drop n [] = []
let drop n x::xs = if n > 0
                   then drop (n-1) xs
                   else x::xs
```