# Programmeringssprog: Assignment 2

Rasmus Dalsgaard (201605295)

April 2017

## Exercise 3

Consider the following definitions and describe the total of distinct functions implemented by them when applied to Boolean values:

### Lambda expression 1

```
(define not-and
  (lambda (b1 b2)
    (not (and b1 b2))))
```

Considering the lambda expression of applying *negation* to an *and*'ed set of Boolean arguments, we would expect the output to be $\neg(b_1 \wedge b_2)$, reminiscent of the binary NAND operation:

| b1 | b2 | out |
|---|---|---|
| #t | #t | #f |
| #t | #f | #t |
| #f | #t | #t |
| #f | #f | #t |

### Lambda expression 2

```
(define not-or
  (lambda (b1 b2)
    (not (or b1 b2))))
```

Considering the lambda expression of applying *negation* to an *or*'d set of Boolean arguments, we would expect the output to be $\neg(b_1 \vee b_2) = (b_1 \downarrow b_2)$, reminiscent of the binary NOR operation:

### Lambda expression 3

```
(define and-not-not
  (lambda (b1 b2)
    (and (not b1) (not b2))))
```

| b1 | b2 | out |
|---|---|---|
| #t | #t | #f |
| #t | #f | #f |
| #f | #t | #f |
| #f | #f | #t |

Considering the lambda expression of applying *and* to an already *negated* set of *and*'d Boolean arguments, we would expect the exact output table of previous NOR expression, merely rewritten as $\neg b_1 \wedge \neg b_2$:

| b1 | b2 | out |
|---|---|---|
| #t | #t | #f |
| #t | #f | #f |
| #f | #t | #f |
| #f | #f | #t |

**Lambda expression 4**

```
(define or-not-not
  (lambda (b1 b2)
    (or (not b1) (not b2))))
```

Considering the lambda expression of applying *or* to an already *negated* set of *and*'d Boolean arguments, we would expect the resulting table to math the very first table of Lambda expression 1, such to comply with the rewritten $\neg b_1 \vee \neg b_2$:

| b1 | b2 | out |
|---|---|---|
| #t | #t | #f |
| #t | #f | #t |
| #f | #t | #t |
| #f | #f | #t |

# Exercise 4

Consider the equality of the following definitions:

```
(define square-of-a-sum
  (lambda (x y)
    (expt (+ x y) 2)))
```

This lambda expression is defined as taking two integers, $x$ and $y$, adding them together by $x + y$ and then using this result in $(exptresult2)$, which is raising the result of $x + y$ to the power of 2, yielding the Calculus formula $(x + y)^2$.

```
(define something-else
  (lambda (x y)
    (+ (expt x 2) (* 2 x y) (expt y 2))))
```

This lambda expression is defined as taking two integers, $x$ and $y$, firstly raising $x$ to the power of 2, secondly multiplying $2 \cdot x \cdot t$, thirdly raising $y$ to the power of 2, and then, lastly, summing up these three subcalcuations, yielding the Calculus formula $x^2 + y^2 + 2xy$, which is the expanded version of $(x+y)^2$.

The first expression being $(x+y)^2$ and the second expression being $x^2 + y^2 + 2xy = (x+y)^2$, we may conclude that these two expressions differ in definition and way of progress, but returns the exact same output if given identical $x$ and $y$ values respectively, regardless of negative or positive integers, due to the fact that $(x+y)^2 = (x+y)^2$.

# Exercise 5

Considering the predefined procedure *iota* which, when applied to non-negative integers, returns a list of 0 to $n-1$, which function does the following procedure compute when given a non-negative integer?

```
(define length-iota
  (lambda (n)
    (length (iota n))))
```

The procedure named $length - iota$ invokes the lambda expression that takes $n$ as input, then computes $(iota\ n)$ which returns the list of $0, 1, ..., n-1, n$. Then *length* procedure is invoked with the computed list as input, returning the length of the list, quite similar to Haskell's $[0..n]$ way of producing lists and its *length* function that counts the length of a list, for example $length[0..n]$.

# Exercise 6

Continued from the previous exercise, Exercise 5, consider the predefined procedure *reverse* and define a compliant procedure *atoi*, that, when applied to a non-negative integer $n$, returns a list from $n-1$ to 0.

Applying the *reverse* function to any list $l$, returns $l$ in reverse, that is $l = (l_1, l_2, .., l_{n-1}, l_n)$, then $reverse(l) = (l_n, l_{n-1}, .., l_2, l_1)$. Combining this function with *iota* which produces $l = (0, 1, 2, .., l_{n-1})$; applying *reverse* to *iota* will suffice.

```
(define length-atoi
  (lambda (n)
    (length (reverse (iota n)))))
```

Tested, will output:

```
> (length-atoi 3)
3
> (length-atoi 7)
7
> (length-atoi -1)
Exception in iota: -1 is not a nonnegative fixnum
```

## Exercise 7

Considering the following procedure,

```
(define identity-at-a-given-point?
  (lambda (p x)
    (equal? (p x) x)))
```

argue whether or not the given list of lambda expressions implements the identity function:

```
(lambda (x) (+ x))
(lambda (x) (- x))
(lambda (x) (* x))
```

As covered earlier in this week's exercises, $+n$, $-n$ and $*n$ will return $n$, so then $x$ applied to $+x$ is merely $x$ applied to $x$ which is $\#t$.

```
(lambda (cs) (string-append cs))
```

An attempt to identify a list of characters and a list of characters as equal containers should result in a *true* statement. Let's test:

```
> (identity-at-a-given-point? (lambda (cs) (string-append cs)) "foo")
#t
```

```
(lambda (xs) (reverse xs))
```

This is a finicky little one. Checking if a list $xs$ is equal to its *reverse* $xs$ will only return $\#t$ if the lists are palindromes; that is, if they're the same read forwards and backwards. This behaviour will only happen if list $l$ is empty or contains a single digit from *iota*, otherwise $(1, 2) \neq (2, 1)$. Of course, any sufficiently clever list hand-crafted will pass this test every time, for example $(1, 1, 1, 2, 1, 1, 1)$. To sum up this rambling, calling this on *iota* lists of length 0 and 1 will return $\#t$ and $\#t$, but $\forall n > 1 | iota\ n \rightarrow \#f$.