# Datatypes in Lambda Calculus

Mathias Vorreiter Pedersen

May 1, 2015

## 1 Booleans

### 1.1 Definition

```
T ≡ λ t f . t
F ≡ λ t f . f
```

### 1.2 Not

```
not ≡ λ b . λ t f . b f t
not ≡ λ b . b (λ t f . f) (λ t f . t)
not ≡ λ b . b F T
¬a ⇝ not a
```

### 1.3 And

```
and ≡ λ b₁ b₂ . b₁ b₂ F
a & b ⇝ and a b
```

### 1.4 Or

```
or ≡ λ b₁ b₂ . b₁ T b₂
a | b ⇝ and a b
```

# 2 Numbers

## 2.1 Definition

```
nat ::= 0
      | Succ nat
```

```
0 ≡ λ f_s v_z . v_z
succ ≡ λ n . λ f_s v_z . f_s (n f_s v_z)
```

Church numerals

```
0 ≡ λ s z . z
1 ≡ λ s z . s z
2 ≡ λ s z . s (s z)
...
```

## 2.2 Is zero?

```
is_zero? ≡ λ n . n (λ α . F) T
```

## 2.3 Plus

```
plus ≡ λ m n . m succ n
a + b ⇝ plus a b
```

## 2.4 Times

```
times ≡ λ m n . m (plus n) 0
a * b ⇝ times a b
```

# 3 Pairs

## 3.1 Definition

```
pair ::= P * *
```

```
mk_pair ≡ λ a b . λ f_p . f_p a b
<a, b> ⇝ mk_pair a b
π_1 ≡ λ p . p (λ a b . a)
π_2 ≡ λ p . p (λ a b . b)
```

## 3.2 Swap

```
swap ≡ λ p . <π_2 p, π_1 p>
swap ≡ λ p . mk_pair (π_2 p) (π_1 p)
```

## 3.3 Sliding Window

$$\texttt{<a, b>} \xmapsto{f} \texttt{<b, a + b>}$$

```
fib ≡ λ n . π_1 (n f <0, 1>)
```

$$\texttt{<a, b>} \xmapsto{f} \texttt{<succ a, a * b>}$$

```
fac ≡ λ n . π_2 (n f <1, 1>)
```

$$\texttt{<a, b>} \xmapsto{f} \texttt{<b, a>}$$

```
is_even? ≡ λ n . π_1 (n f <T, F>)
```

# 4 Lists

## 4.1 Definition

```
list ::= Nil
       | Cons * list
```

```
mk_cons ≡ λ hd tl . λ f_c v_n . f_c hd (tl f_c v_n)
hd :: tl ⇝ mk_cons hd tl
nil ≡ λ f_c v_n . v_n
```

## 4.2 Sum

```
sum_list ≡ λ xs . xs plus 0
```

## 4.3 Append

```
append ≡ λ xs ys . xs mk_cons ys
```

# 5 Trees

## 5.1 Definition

```
tree ::= Leaf *
       | Node tree tree
```

```
mk_node ≡ λ t_1 t_2 . λ f_n f_l . f_n (t_1 f_n f_l) (t_2 f_n f_l)
mk_leaf ≡ λ n . λ f_n f_l . f_l n
```

## 5.2 Sum

```
sum_tree ≡ λ t . t plus id
```

# 6 Exercises

1. Define the binary xor function, which returns T exactly when one of its arguments is T.

2. Prove De Morgan's laws by verifying that for all boolean terms b1, b2 it holds that:

```
not (and b1 b2) = or (not b1) (not b2)
not (or b1 b2) = and (not b1) (not b2)
```

3. We extend boolean logic with another value called *unknown*. For obvious reasons this is called three-valued logic. Create an ML-like datatype describing this type, and derive lambda calculus terms for the constructors of this new datatype (e.g T, F and U).

4. The operations not, and and or for three-valued logic can be described by the following truth tables:

| A | ¬A |
|---|---|
| F | T |
| U | U |
| T | F |

| A | B | A ∧ B |
|---|---|---|
| F | F | F |
| F | U | F |
| F | T | F |
| U | F | F |
| U | U | U |
| U | T | U |
| T | F | F |
| T | U | U |
| T | T | T |

| A | B | A ∨ B |
|---|---|---|
| F | F | F |
| F | U | U |
| F | T | T |
| U | F | U |
| U | U | U |
| U | T | T |
| T | F | T |
| T | U | T |
| T | T | T |

   Implement these functions as lambda calculus terms. Optionally, prove that the De Morgan identities are still satisfied.

5. Implement a function power, which computes $n^m$ when given natural numbers $n$ and $m$ as church numerals.

6. Implement the function $\uparrow\uparrow$, which computes

$$\underbrace{n^{n^{\cdot^{\cdot^{\cdot^{n}}}}}}_{m \text{ times}}.$$

This notation is due to the great mathematician and computer scientist Donald Knuth, who also wrote $n^m$ as $n \uparrow m$, because of the difficulties of using the $n^m$-notation to describe continued exponentiation.

7. Implement a function that returns $\lfloor \frac{n}{2} \rfloor$ when given a natural number $n$ encoded as a church numeral.

Hint: Combine what we have learned from pairs, booleans and sliding window.

8. Implement a function car which returns the first element of a given list. You can assume the list is not empty.

Similarly implement a function cdr, which drops the first element of the list.

9. Implement a function list→pair which converts a list xs into nested pairs representing the same list. You can assume that $|\mathsf{xs}| \geq 2$.

10. Implement a function flatten, which returns a list of all the elements in a tree.

11. Implement a function mirror, which flips the nodes in a tree. Example:

```
mirror (mk_node
          (mk_node
            (mk_leaf 0)
            (mk_leaf 1))
          (mk_leaf 2)) =
(mk_node
  (mk_leaf 2)
  (mk_node
    (mk_leaf 1)
    (mk_leaf 0)))
```

12. The Option datatype is a well-known concept from functional programming:

```
Option ::= None
         | Some *
```

    Implement this datatype as lambda terms, along with predicates is_none? (resp. is_some?), which return T if its argument has been constructed using None (resp. Some).

13. Implement a function maybe, that when given a default argument d, a function f and an option value o, returns d if is_none? o = T. Alternatively when o = Some v the function should return f v.

14. Implement a function lift, that when given a function f: A → B, returns a function g: Option A → Option B.

    More concretely, None $\overset{g}{\mapsto}$ None and Some v $\overset{g}{\mapsto}$ Some (f v).