Functional Programming

by Łukasz Stafiniak

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

Lecture 4: Functions.

Programming in untyped λ-calculus.

Introduction to Lambda Calculus **Henk Barendregt, Erik Barendsen**

Lecture Notes on the Lambda Calculus **Peter Selinger**

Review: a "computation by hand" example

Let's compute some larger, recursive program.
Recall that we use `fix` instead of `let rec` to simplify rules for recursion.
Also remember our syntactic conventions:
`fun x y -> e` stands for `fun x -> (fun y -> e)`, etc.

```
let rec fix f x = f (fix f) x
type int_list = Nil | Cons of int * int_list
```

Preparations.

We will evaluate (reduce) the following expression.

```
let length =
  fix (fun f l ->
    match l with
      | Nil -> 0
      | Cons (x, xs) -> 1 + f xs) in
length (Cons (1, (Cons (2, Nil))))
```

```
let length =
  fix (fun f l ->
    match l with
      | Nil -> 0
      | Cons (x, xs) -> 1 + f xs) in
length (Cons (1, (Cons (2, Nil))))
```

$$\texttt{let}\ x = v\ \texttt{in}\ a \rightsquigarrow a[x := v]$$

```
  fix (fun f l ->
    match l with
      | Nil -> 0
      | Cons (x, xs) -> 1 + f xs) (Cons (1, (Cons (2, Nil))))
```

$$\texttt{fix}^2\, v_1\, v_2 \rightsquigarrow v_1\, (\texttt{fix}^2\, v_1)\, v_2$$

$$\texttt{fix}^2\, v_1\, v_2 \;\rightsquigarrow\; v_1\, (\texttt{fix}^2\, v_1)\, v_2$$

```
(fun f l ->
  match l with
    | Nil -> 0
    | Cons (x, xs) -> 1 + f xs)
  (fix (fun f l ->
    match l with
      | Nil -> 0
      | Cons (x, xs) -> 1 + f xs))
  (Cons (1, (Cons (2, Nil))))
```

$$(\texttt{fun}\;\; x\,\texttt{->}\,a)\; v \;\rightsquigarrow\; a[x := v]$$

$$a_1\, a_2 \;\rightsquigarrow\; a_1'\, a_2$$

$$(\texttt{fun}\ x\texttt{->}a)\,v \ \rightsquigarrow\ a[x := v]$$

$$a_1\,a_2 \ \lightning\ a_1'\,a_2$$

```
(fun l ->
  match l with
    | Nil -> 0
    | Cons (x, xs) -> 1 + (fix (fun f l ->
      match l with
        | Nil -> 0
        | Cons (x, xs) -> 1 + f xs)) xs)
  (Cons (1, (Cons (2, Nil))))
```

$$(\texttt{fun}\ x\texttt{->}a)\,v \ \lightning\ a[x := v]$$

$$(\texttt{fun}\ x \texttt{->} a)\, v \rightsquigarrow a[x := v]$$

```
(match Cons (1, (Cons (2, Nil))) with
  | Nil -> 0
  | Cons (x, xs) -> 1 + (fix (fun f l ->
    match l with
      | Nil -> 0
      | Cons (x, xs) -> 1 + f xs)) xs)
```

$$\begin{array}{l}\texttt{match}\ C_1^n(v_1, ..., v_n)\ \texttt{with} \\ \quad C_2^n(p_1, ..., p_k)\texttt{->}a\ \texttt{|}\ \mathrm{pm} \rightsquigarrow \texttt{match}\ C_1^n(v_1, ..., v_n) \\ \hfill \texttt{with}\ \mathrm{pm}\end{array}$$

$$\texttt{match}\ C_1^n(v_1, \ldots, v_n)\ \texttt{with}$$
$$C_2^n(p_1, \ldots, p_k)\texttt{->}a\ \texttt{|}\ \text{pm} \rightsquigarrow \texttt{match}\ C_1^n(v_1, \ldots, v_n)\ \texttt{with}\ \text{pm}$$

```
(match Cons (1, (Cons (2, Nil))) with
  | Cons (x, xs) -> 1 + (fix (fun f l ->
    match l with
      | Nil -> 0
      | Cons (x, xs) -> 1 + f xs)) xs)
```

$$\texttt{match}\ C_1^n(v_1, \ldots, v_n)\ \texttt{with}$$
$$C_1^n(x_1, \ldots, x_n)\texttt{->}a\ \texttt{|}\ \ldots \rightsquigarrow a[x_1 := v_1; \ldots; x_n := v_n]$$

$$\texttt{match}\ C_1^n(v_1, \ldots, v_n)\ \texttt{with}$$

$$C_1^n(x_1, \ldots, x_n)\texttt{->}a\ \texttt{|}\ \ldots \leadsto a[x_1 := v_1; \ldots; x_n := v_n]$$

```
1 + (fix (fun f l ->
     match l with
       | Nil -> 0
       | Cons (x, xs) -> 1 + f xs)) (Cons (2, Nil))
```

$$\texttt{fix}^2\, v_1\, v_2 \rightsquigarrow v_1\, (\texttt{fix}^2\, v_1)\, v_2$$

$$a_1\, a_2 \leadsto a_1\, a_2'$$

$$\texttt{fix}^2\, v_1\, v_2 \;\rightsquigarrow\; v_1\,(\texttt{fix}^2\, v_1)\, v_2$$

$$a_1\, a_2 \;\rightsquigarrow\; a_1\, a_2'$$

```
1 + (fun f l ->
      match l with
        | Nil -> 0
        | Cons (x, xs) -> 1 + f xs))
      (fix (fun f l ->
         match l with
           | Nil -> 0
           | Cons (x, xs) -> 1 + f xs)) (Cons (2, Nil))
```

$$(\texttt{fun}\ x\texttt{->}a)\, v \;\rightsquigarrow\; a[x := v]$$

$$a_1\, a_2 \;\rightsquigarrow\; a_1\, a_2'$$

$$(\texttt{fun}\ x\texttt{->}a)\,v \ \rightsquigarrow\ a[x:=v]$$

$$a_1\,a_2 \ \rightsquigarrow\ a_1\,a_2'$$

```
1 + (fun l ->
      match l with
        | Nil -> 0
        | Cons (x, xs) -> 1 + (fix (fun f l ->
            match l with
              | Nil -> 0
              | Cons (x, xs) -> 1 + f xs)) xs))
      (Cons (2, Nil))
```


$$(\texttt{fun}\ x\texttt{->}a)\,v \ \rightsquigarrow\ a[x:=v]$$

$$a_1\,a_2 \ \rightsquigarrow\ a_1\,a_2'$$

$$(\texttt{fun}\ \ x\texttt{->}a)\,v \ \rightsquigarrow\ a[x := v]$$

$$a_1\,a_2 \ \ \wr\!\!\downarrow\ \ a_1\,a_2'$$

```
1 + (match Cons (2, Nil) with
      | Nil -> 0
      | Cons (x, xs) -> 1 + (fix (fun f l ->
        match l with
          | Nil -> 0
          | Cons (x, xs) -> 1 + f xs)) xs))
```

$$\begin{aligned}&\texttt{match}\ C_1^n(v_1, \ldots, v_n)\ \texttt{with}\\ &\quad C_2^n(p_1, \ldots, p_k)\texttt{->}a\ \texttt{|}\ \mathrm{pm} \ \rightsquigarrow\ \texttt{match}\ C_1^n(v_1, \ldots, v_n)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{with}\ \mathrm{pm}\end{aligned}$$

$$a_1\,a_2 \ \ \wr\!\!\downarrow\ \ a_1\,a_2'$$

$$\begin{array}{l}\texttt{match}\ C_1^n(v_1,\ldots,v_n)\ \texttt{with}\\ \quad C_2^n(p_1,\ldots,p_k)\texttt{->}a\ \texttt{|}\ \mathrm{pm}\ \rightsquigarrow\ \texttt{match}\ C_1^n(v_1,\ldots,v_n)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{with}\ \mathrm{pm}\end{array}$$

$$a_1\, a_2 \ \rightsquigarrow\ a_1\, a_2'$$

```
1 + (match Cons (2, Nil) with
       | Cons (x, xs) -> 1 + (fix (fun f l ->
         match l with
           | Nil -> 0
           | Cons (x, xs) -> 1 + f xs)) xs)
```

$$\begin{array}{l}\texttt{match}\ C_1^n(v_1,\ldots,v_n)\ \texttt{with}\\ \quad C_1^n(x_1,\ldots,x_n)\texttt{->}a\ \texttt{|}\ \ldots\ \rightsquigarrow\ a[x_1:=v_1;\ldots;x_n:=v_n]\end{array}$$

$$\texttt{match}\ C_1^n(v_1, \ldots, v_n)\ \texttt{with}$$
$$C_1^n(x_1, \ldots, x_n)\texttt{->}a\ \texttt{|}\ \ldots \ \rightsquigarrow\ a[x_1 := v_1; \ldots; x_n := v_n]$$
$$a_1\, a_2 \ \rightsquigarrow\ a_1\, a_2'$$

```
1 + (1 + (fix (fun f l ->
            match l with
              | Nil -> 0
              | Cons (x, xs) -> 1 + f xs)) Nil)
```

$$\texttt{fix}^2\, v_1\, v_2 \ \rightsquigarrow\ v_1\,(\texttt{fix}^2\, v_1)\, v_2$$
$$a_1\, a_2 \ \rightsquigarrow\ a_1\, a_2'$$
$$a_1\, a_2 \ \rightsquigarrow\ a_1\, a_2'$$

$$
\begin{array}{rcl}
\mathtt{fix}^2\, v_1\, v_2 & \rightsquigarrow & v_1\,(\mathtt{fix}^2\, v_1)\, v_2 \\
a_1\, a_2 & \wr & a_1\, a_2' \\
a_1\, a_2 & \wr & a_1\, a_2'
\end{array}
$$

```
1 + (1 + (fun f l ->
           match l with
             | Nil -> 0
             | Cons (x, xs) -> 1 + f xs) (fix (fun f l ->
               match l with
                 | Nil -> 0
                 | Cons (x, xs) -> 1 + f xs)) Nil)
```

$$
\begin{array}{rcl}
(\mathtt{fun}\ x\,\texttt{->}a)\, v & \rightsquigarrow & a[x := v] \\
a_1\, a_2 & \wr & a_1\, a_2' \\
a_1\, a_2 & \wr & a_1\, a_2'
\end{array}
$$

$$
\begin{array}{rcl}
(\texttt{fun}\ x\texttt{->}a)\,v & \rightsquigarrow & a[x := v] \\
a_1\,a_2 & \wr & a_1\,a_2' \\
a_1\,a_2 & \wr & a_1\,a_2'
\end{array}
$$

```
1 + (1 + (fun l ->
           match l with
             | Nil -> 0
             | Cons (x, xs) -> 1 + (fix (fun f l ->
               match l with
                 | Nil -> 0
                 | Cons (x, xs) -> 1 + f xs)) xs) Nil)
```

$$
\begin{array}{rcl}
(\texttt{fun}\ x\texttt{->}a)\,v & \rightsquigarrow & a[x := v] \\
a_1\,a_2 & \wr & a_1\,a_2' \\
a_1\,a_2 & \wr & a_1\,a_2'
\end{array}
$$

$$(\texttt{fun } x\texttt{->}a)\,v \;\rightsquigarrow\; a[x := v]$$

$$a_1\,a_2 \;\rightsquigarrow\; a_1\,a_2'$$

$$a_1\,a_2 \;\rightsquigarrow\; a_1\,a_2'$$

```
1 + (1 + (match Nil with
            | Nil -> 0
            | Cons (x, xs) -> 1 + (fix (fun f l ->
              match l with
                | Nil -> 0
                | Cons (x, xs) -> 1 + f xs)) xs))
```

$$\begin{array}{l} \texttt{match } C_1^n(v_1, ..., v_n) \texttt{ with} \\ \quad C_1^n(x_1, ..., x_n)\texttt{->}a \;\texttt{|}\; ... \;\rightsquigarrow\; a[x_1 := v_1; ...; x_n := v_n] \end{array}$$

$$a_1\,a_2 \;\rightsquigarrow\; a_1\,a_2'$$

$$a_1\,a_2 \;\rightsquigarrow\; a_1\,a_2'$$

$$\texttt{match } C_1^n(v_1, ..., v_n) \texttt{ with}$$
$$C_1^n(x_1, ..., x_n)\texttt{->}a \texttt{ | } ... \rightsquigarrow a[x_1 := v_1; ...; x_n := v_n]$$
$$a_1\, a_2 \;\; \wr\!\!\downarrow \;\; a_1\, a_2'$$
$$a_1\, a_2 \;\; \wr\!\!\downarrow \;\; a_1\, a_2'$$

```
1 + (1 + 0)
```

$$f^n\, v_1 \ldots v_n \rightsquigarrow f(v_1, ..., v_n)$$
$$a_1\, a_2 \;\; \wr\!\!\downarrow \;\; a_1\, a_2'$$

```
1 + 1
```

$$f^n\, v_1 \ldots v_n \;\; \wr\!\!\downarrow \;\; f(v_1, ..., v_n)$$

```
2
```

Language and rules of the untyped λ-calculus

- First, let's forget about types.
- Next, let's introduce a shortcut:
  - We write $\lambda x.a$ for `fun` x`->`a, $\lambda xy.a$ for `fun` x y`->`a, etc.
- Let's forget about all other constructions, only `fun` and variables.
- The real λ-calculus has a more general reduction:

$$(\texttt{fun } x\texttt{->}a_1)\, a_2 \rightsquigarrow a_1[x := a_2]$$

(called *β-reduction*) and uses *bound variable renaming* (called *α-conversion*), or some other trick, to avoid *variable capture*. But let's not overcomplicate things.

  - We will look into the β-reduction rule in the **laziness** lecture.
  - Why is β-reduction more general than the rule we use?

Booleans

- Alonzo Church introduced λ-calculus to encode logic.
- There are multiple ways to encode various sorts of data in λ-calculus. Not all of them make sense in a typed setting, i.e. the straightforward encode/decode functions do not type-check for them.
- Define `c_true`$=\lambda xy.x$ and `c_false`$=\lambda xy.y$.
- Define `c_and`$=\lambda xy.xy$ `c_false`. Check that it works!
  - I.e. that `c_and c_true c_true` = `c_true`, otherwise `c_and a b` = `c_false`.

```
let c_true = fun x y -> x
let c_false = fun x y -> y
let c_and = fun x y -> x y c_false
let encode_bool b = if b then c_true else c_false
let decode_bool c = c true false
```

"True" is projection on the first argument.
And "false" on the second argument.
If one is false, then return false.
Test the functions in the toplevel.

- Define `c_or` and `c_not` yourself!

If-then-else and pairs

- We will just use the OCaml syntax from now.

```
let if_then_else = fun b -> b
```

Booleans select the argument!

Remember to play with the functions in the toplevel.

```
let c_pair m n = fun x -> x m n
let c_first = fun p -> p c_true
let c_second = fun p -> p c_false
```

We couple things
by passing them together.
Check that it works!

```
let encode_pair enc_fst enc_snd (a, b) =
  c_pair (enc_fst a) (enc_snd b)
let decode_pair de_fst de_snd c = c (fun x y -> de_fst x, de_snd y)
let decode_bool_pair c = decode_pair decode_bool decode_bool c
```

- We can define larger tuples in the same manner:

```
let c_triple l m n = fun x -> x l m n
```

Pair-encoded natural numbers

- Our first encoding of natural numbers is as the depth of nested pairs whose rightmost leaf is $\lambda x.x$ and whose left elements are `c_false`.

```
let pn0 = fun x -> x
let pn_succ n = c_pair c_false n
```

Start with the identity function.
Stack another pair.

```
let pn_pred = fun x -> x c_false
let pn_is_zero = fun x -> x c_true
```

[Explain these functions.]

We program in untyped lambda calculus as an exercise, and we need encoding / decoding to verify our exercises, so using "magic" for encoding / decoding is "fair game".

```
let rec encode_pnat n =
  if n <= 0 then Obj.magic pn0
  else pn_succ (Obj.magic (encode_pnat (n-1)))
let rec decode_pnat pn =
  if decode_bool (pn_is_zero pn) then 0
  else 1 + decode_pnat (pn_pred (Obj.magic pn))
```

We use `Obj.magic` to forget types.
Disregarding types, these functions are straightforward!

Church numerals (natural numbers in Ch. enc.)

- Do you remember our function `power f n`? We will use its variant for a different representation of numbers:

```
let cn0 = fun f x -> x
let cn1 = fun f x -> f x
let cn2 = fun f x -> f (f x)
let cn3 = fun f x -> f (f (f x))
```

The same as `c_false`.

Behaves like identity.

- This is the original Alonzo Church encoding.

```
let cn_succ = fun n f x -> f (n f x)
```

- Define addition, multiplication, comparing to zero, and the predecesor function “-1” for Church numerals.
- Turns out even Alozno Church couldn't define predecesor right away! But try to make some progress before you turn to the next slide.
  - His student Stephen Kleene found it.

```
let rec encode_cnat n f =
  if n <= 0 then (fun x -> x) else f -| encode_cnat (n-1) f
let decode_cnat n = n ((+) 1) 0
let cn7 f x = encode_cnat 7 f x
let cn13 f x = encode_cnat 13 f x
```

We need to η-*expand* these definitions for type-system reasons. (Because OCaml allows *side-effects*.)

```
let cn_add = fun n m f x -> n f (m f x)
```

Put `n` of `f` in front.

```
let cn_mult = fun n m f -> n (m f)
```

Repeat `n` times putting `m` of `f` in front.

```
let cn_prev n =
  fun f x ->
    n
      (fun g v -> v (g f))
      (fun z->x)
      (fun z->z)
```

This is the "Church numeral signature". The only thing we have is an `n`-step loop. We need sth that operates on `f`. We need to ignore the innermost step. We've build a "machine" not results – start the machine.

`cn_is_zero` left as an exercise.

```
decode_cnat (cn_prev cn3)

(cn_prev cn3) ((+) 1) 0

(fun f x ->
    cn3
      (fun g v -> v (g f))
      (fun z->x)
      (fun z->z)) ((+) 1) 0

((fun f x -> f (f (f x)))
      (fun g v -> v (g ((+) 1)))
      (fun z->0)
      (fun z->z))
```

```
((fun g v -> v (g ((+) 1)))
  ((fun g v -> v (g ((+) 1)))
    ((fun g v -> v (g ((+) 1)))
      (fun z->0))))
  (fun z->z))
```

$\rightsquigarrow$

```
((fun z->z)
  (((fun g v -> v (g ((+) 1)))
    ((fun g v -> v (g ((+) 1)))
      (fun z->0)))) ((+) 1)))
```

$\rightsquigarrow$

```
(fun g v -> v (g ((+) 1)))
  ((fun g v -> v (g ((+) 1)))
    (fun z->0)) ((+) 1)
```

$\rightsquigarrow$

```
((+) 1) ((fun g v -> v (g ((+) 1)))
          (fun z->0) ((+) 1))
```

$\rightsquigarrow$

```
((+) 1) (((+) 1) ((fun z->0) ((+) 1)))
```

$\rightsquigarrow$

```
((+) 1) (((+) 1) (0))
```

$\rightsquigarrow$

```
((+) 1) 1
```

$\rightsquigarrow$

```
2
```

Recursion: Fixpoint Combinator

- Turing's fixpoint combinator: $\Theta = (\lambda xy.y\,(xxy))\,(\lambda xy.y\,(xxy))$

$$
\begin{array}{rcl}
N & = & \Theta\, F \\
 & = & (\lambda xy.y\,(xxy))\,(\lambda xy.y\,(xxy))\,F \\
 & =_{\to\to} & F\,((\lambda xy.y\,(xxy))\,(\lambda xy.y\,(xxy))\,F) \\
 & = & F\,(\Theta\, F) = FN
\end{array}
$$

- Curry's fixpoint combinator: $\boldsymbol{Y} = \lambda f.(\lambda x.f\,(xx))\,(\lambda x.f\,(xx))$

$$
\begin{array}{rcl}
N & = & \boldsymbol{Y}F \\
 & = & (\lambda f.(\lambda x.f\,(xx))\,(\lambda x.f\,(xx)))\,F \\
 & =_{\to} & (\lambda x.F\,(xx))\,(\lambda x.F\,(xx)) \\
 & =_{\to} & F\,((\lambda x.F\,(xx))\,(\lambda x.F\,(xx))) \\
 & =_{\leftarrow} & F\,((\lambda f.(\lambda x.f\,(xx))\,(\lambda x.f\,(xx)))\,F) \\
 & = & F\,(\boldsymbol{Y}F) = FN
\end{array}
$$

- Call-by-value *fix*point combinator: $\lambda f'.(\lambda fx.f'\,(ff)\,x)\,(\lambda fx.f'\,(ff)\,x)$

$$
\begin{array}{rcl}
N & = & \mathrm{fix}\,F \\
 & = & (\lambda f'.(\lambda fx.f'\,(ff)\,x)\,(\lambda fx.f'\,(ff)\,x))\,F \\
 & =_{\rightarrow} & (\lambda fx.F\,(ff)\,x)\,(\lambda fx.F\,(ff)\,x) \\
 & =_{\rightarrow} & \lambda x.F\,((\lambda fx.F\,(ff)\,x)\,(\lambda fx.F\,(ff)\,x))\,x \\
 & =_{\leftarrow} & \lambda x.F\,((\lambda f'.(\lambda fx.f'\,(ff)\,x)\,(\lambda fx.f'\,(ff)\,x))\,F)\,x \\
 & = & \lambda x.F\,(\mathrm{fix}\,F)\,x = \lambda x.FNx \\
 & =_{\eta} & FN
\end{array}
$$

- The λ-terms we have seen above are **fixpoint combinators** – means inside λ-calculus to perform recursion.
- What is the problem with the first two combinators?

$$
\begin{array}{rcl}
\Theta F & \rightsquigarrow\rightsquigarrow & F\,((\lambda xy.y\,(xxy))\,(\lambda xy.y\,(xxy))\,F) \\
 & \rightsquigarrow\rightsquigarrow & F\,(F\,((\lambda xy.y\,(xxy))\,(\lambda xy.y\,(xxy))\,F)) \\
 & \rightsquigarrow\rightsquigarrow & F\,(F\,(F\,((\lambda xy.y\,(xxy))\,(\lambda xy.y\,(xxy))\,F))) \\
 & \rightsquigarrow\rightsquigarrow & \ldots
\end{array}
$$

- Recall the distinction between *expressions* and *values* from the previous lecture *Computation*.
- The reduction rule for λ-calculus is just meant to determine which expressions are considered "equal" – it is highly *non-deterministic*, while on a computer, computation needs to go one way or another.
- Using the general reduction rule of λ-calculus, for a recursive definition, it is always possible to find an infinite reduction sequence (which means that you couldn't complain when a nasty λ-calculus compiler generates infinite loops for all recursive definitions).
  - Why?
- Therefore, we need more specific rules. For example, most languages use $(\texttt{fun}\ x\texttt{->}a)\, v \rightsquigarrow a[x := v]$, which is called *call-by-value*, or **eager** computation (because the program *eagerly* computes the arguments before starting to compute the function). (It's exactly the rule we introduced in *Computation* lecture.)

- What happens with call-by-value fixpoint combinator?

$$
\begin{aligned}
\mathrm{fix}\, F &\rightsquigarrow (\lambda f x. F\,(f f)\,x)\,(\lambda f x. F\,(f f)\,x) \\
&\rightsquigarrow \lambda x. F\,((\lambda f x. F\,(f f)\,x)\,(\lambda f x. F\,(f f)\,x))\,x
\end{aligned}
$$

Voila – if we use $(\texttt{fun}\ x\texttt{->}a)\,v \rightsquigarrow a[x := v]$ as the rule rather than $(\texttt{fun}\ x\texttt{->}a_1)\ a_2 \rightsquigarrow a_1[x := a_2]$, the computation stops. Let's compute the function on some input:

$$
\begin{aligned}
\mathrm{fix}\, F v &\rightsquigarrow (\lambda f x. F\,(f f)\,x)\,(\lambda f x. F\,(f f)\,x)\,v \\
&\rightsquigarrow (\lambda x. F\,((\lambda f x. F\,(f f)\,x)\,(\lambda f x. F\,(f f)\,x))\,x)\,v \\
&\rightsquigarrow F\,((\lambda f x. F\,(f f)\,x)\,(\lambda f x. F\,(f f)\,x))\,v \\
&\rightsquigarrow F\,(\lambda x. F\,((\lambda f x. F\,(f f)\,x)\,(\lambda f x. F\,(f f)\,x))\,x)\,v \\
&\rightsquigarrow \text{depends on } F
\end{aligned}
$$

- Why the name *fixpoint*? If you look at our derivations, you'll see that they show what in math can be written as $x = f(x)$. Such values x are called fixpoints of f. An arithmetic function can have several fixpoints, for example $f(x) = x^2$ (which $x$es are fixpoints?) or no fixpoints, for example $f(x) = x + 1$.

- When you define a function (or another object) by recursion, it has very similar meaning: there is a name that is on both sides of $=$.

- In λ-calculus, there are functions like Θ and $\boldsymbol{Y}$, that take *any* function as an argument, and return its fixpoint.

- We turn a specification of a recursive object into a definition, by solving it with respect to the recurring name: deriving $x = f(x)$ where x is the recurring name. We then have $x = \mathrm{fix}(f)$.

- Let's walk through it for the factorial function (we omit the prefix `cn_` – could be `pn_` if `pn1` was used instead of `cn1` – for numeric functions, and we shorten `if_then_else` into `if_t_e`):

$$
\begin{aligned}
\mathtt{fact}\,n &= \mathtt{if\_t\_e}\,(\mathtt{is\_zero}\,n)\,\mathtt{cn1}\,(\mathtt{mult}\,n\,(\mathtt{fact}\,(\mathtt{pred}\,n)))\\
\mathtt{fact} &= \lambda n.\mathtt{if\_t\_e}\,(\mathtt{is\_zero}\,n)\,\mathtt{cn1}\,(\mathtt{mult}\,n\,(\mathtt{fact}\,(\mathtt{pred}\,n)))\\
\mathtt{fact} &= (\lambda f n.\mathtt{if\_t\_e}\,(\mathtt{is\_zero}\,n)\,\mathtt{cn1}\,(\mathtt{mult}\,n\,(f\,(\mathtt{pred}\,n))))\,\mathtt{fact}\\
\mathtt{fact} &= \mathrm{fix}\,(\lambda f n.\mathtt{if\_t\_e}\,(\mathtt{is\_zero}\,n)\,\mathtt{cn1}\,(\mathtt{mult}\,n\,(f\,(\mathtt{pred}\,n))))
\end{aligned}
$$

  The last specification is a valid definition: we just give a name to a (*ground*, a.k.a. *closed*) expression.

- We have seen how `fix` works already!
  - Compute `fact cn2`.
- What does `fix (fun x -> cn_succ x)` mean?

Encoding of Lists and Trees

- A list is either empty, which we often call `Empty` or `Nil`, or it consists of an element followed by another list (called "tail"), the other case often called `Cons`.
- Define `nil`$=\lambda xy.y$ and `cons` $HT = \lambda xy.xHT$.
- Add numbers stored inside a list:

$$\texttt{addlist}\, l \;=\; l\,(\lambda ht.\texttt{cn\_add}\, h\,(\texttt{addlist}\, t))\,\texttt{cn0}$$

To make a proper definition, we need to apply fix to the solution of above equation.

$$\texttt{addlist} \;=\; \mathrm{fix}\,(\lambda fl.l\,(\lambda ht.\texttt{cn\_add}\, h\,(ft))\,\texttt{cn0})$$

- For trees, let's use a different form of binary trees than so far: instead of keeping elements in inner nodes, we will keep elements in leaves.
- Define $\texttt{leaf}\, n = \lambda xy.xn$ and $\texttt{node}\, L\,R = \lambda xy.yLR$.
- Add numbers stored inside a tree:

$$\texttt{addtree}\, t \;=\; t\,(\lambda n.n)\,(\lambda lr.\texttt{cn\_add}\,(\texttt{addtree}\, l)\,(\texttt{addtree}\, r))$$

and, in solved form:

$$\texttt{addtree} \;=\; \mathrm{fix}\,(\lambda ft.t\,(\lambda n.n)\,(\lambda lr.\texttt{cn\_add}\,(fl)\,(fr)))$$

```
let nil = fun x y -> y
let cons h t = fun x y -> x h t
let addlist l =
  fix (fun f l -> l (fun h t -> cn_add h (f t)) cn0) l
;;
decode_cnat
  (addlist (cons cn1 (cons cn2 (cons cn7 nil))));;
let leaf n = fun x y -> x n
let node l r = fun x y -> y l r
let addtree t =
  fix (fun f t ->
    t (fun n -> n) (fun l r -> cn_add (f l) (f r))
  ) t
;;
decode_cnat
  (addtree (node (node (leaf cn3) (leaf cn7))
                 (leaf cn1)));;
```

- Observe a regularity: when we encode a variant type with n variants, for each variant we define a function that takes n arguments.

- If the kth variant C_k has m_k parameters, then the function c_k that encodes it will have the form:

$$C_k(v_1, \ldots, v_{m_k}) \sim c_k\, v_1 \ldots v_{m_k} = \lambda x_1 \ldots x_n . x_k\, v_1 \ldots v_{m_k}$$

- The encoded variants serve as a shallow pattern matching with guaranteed exhaustiveness: kth argument corresponds to kth branch of pattern matching.

Looping Recursion

- Let's come back to numbers defined as lengths lists and define addition:

```
let pn_add m n =
  fix (fun f m n ->
    if_then_else (pn_is_zero m)
      n (pn_succ (f (pn_pred m) n))
  ) m n;;
decode_pnat (pn_add pn3 pn3);;
```

- Oops… OCaml says:
  `Stack overflow during evaluation (looping recursion?).`

- What is wrong? Nothing as far as λ-calculus is concerned. But OCaml and F# always compute arguments before calling a function. By definition of `fix`, `f` corresponds to recursively calling `pn_add`. Therefore, `(pn_succ (f (pn_pred m) n))` will be called regardless of what `(pn_is_zero m)` returns!

- Why `addlist` and `addtree` work?

- `addlist` and `addtree` work because their recursive calls are "guarded" by corresponding `fun`. What is inside of `fun` is not computed immediately, only when the function is applied to argument(s).

- To avoid looping recursion, you need to guard all recursive calls. Besides putting them inside `fun`, in OCaml or F# you can also put them in branches of a `match` clause, as long as one of the branches does not have unguarded recursive calls!

- The trick to use with functions like `if_then_else`, is to guard their arguments with `fun x ->`, where `x` is not used, and apply the *result* of `if_then_else` to some dummy value.
    - In OCaml or F# we would guard by `fun () ->`, and then apply to `()`, but we do not have datatypes like `unit` in λ-calculus.

```
let pn_add m n =
  fix (fun f m n ->
    (if_then_else (pn_is_zero m)
       (fun x -> n) (fun x -> pn_succ (f (pn_pred m) n)))
      id
  ) m n;;
decode_pnat (pn_add pn3 pn3);;
decode_pnat (pn_add pn3 pn7);;
```

In-class Work and Homework

Define (implement) and verify:

1. `c_or` and `c_not`;
2. exponentiation for Church numerals;
3. is-zero predicate for Church numerals;
4. even-number predicate for Church numerals;
5. multiplication for pair-encoded natural numbers;
6. factorial $n!$ for pair-encoded natural numbers.
7. Construct λ-terms $m_0, m_1, \ldots$ such that for all n one has:

$$\begin{aligned} m_0 &= x \\ m_{n+1} &= m_{n+2}\, m_n \end{aligned}$$

(where equality is after performing β-reductions).

8. Define (implement) and verify a function computing: the length of a list (in Church numerals);
9. `cn_max` – maximum of two Church numerals;
10. the depth of a tree (in Church numerals).
11. Representing side-effects as an explicitly “passed around” state value, write combinators that represent the imperative constructs:
    a. `for`...`to`...
    b. `for`...`downto`...
    c. `while`...`do`...
    d. `do`...`while`...
    e. `repeat`...`until`...

Rather than writing a λ-term using the encodings that we've learnt, just implement the functions in OCaml / F#, using built-in `int` and `bool` types. You can use `let rec` instead of `fix`.

- For example, in exercise (a), write a function `let rec for_to f beg_i end_i s =`... where `f` takes arguments `i` ranging from `beg_i` to `end_i`, state `s` at given step, and returns state `s` at next step; the `for_to` function returns the state after the last step.
- And in exercise (c), write a function `let rec while_do p f s =`... where both `p` and `f` take state `s` at given step, and if `p s` returns true, then `f s` is computed to obtain state at next step; the `while_do` function returns the state after the last step.

Do not use the imperative features of OCaml and F#, we will not even cover them in this course!

Despite we will not cover them, it is instructive to see the implementation using the imperative features, to better understand what is actually required of a solution to the last exercise.

a)
```
let for_to f beg_i end_i s =
    let s = ref s in
    for i = beg_i to end_i do
      s := f i !s
    done;
    !s
```

b)
```
let for_downto f beg_i end_i s =
    let s = ref s in
    for i = beg_i downto end_i do
      s := f i !s
    done;
    !s
```

c)
```
let while_do p f s =
  let s = ref s in
  while p !s do
    s := f !s
  done;
  !s
```

d)
```
let do_while p f s =
  let s = ref (f s) in
  while p !s do
    s := f !s
  done;
  !s
```

e)
```
let repeat_until p f s =
  let s = ref (f s) in
  while not (p !s) do
    s := f !s
  done;
  !s
```