Funktionell programmering

Programmeringsparadigm

<table>
<tr><td colspan="4">Programmeringsparadigm</td></tr>
<tr><td colspan="2">Imperativa</td><td colspan="2">Deklarativa</td></tr>
<tr><td>Procedurori-
enterade</td><td>Objektorien-
terade</td><td>Funktionella</td><td>Logiska</td></tr>
</table>

Imperativ programmering

Program består av explicita instruktioner för att ändra programmets tillstånd.

Exempel: RISC-V assembly

```
  .text
  .globl  _reset
_reset:
  /* initialize global pointer */
  .option push
  .option norelax
  la  gp, __global_pointer$
  .option pop

  /* clear bss */
  la  t2, __bss_start
  la  t3, __bss_end
  bge t2, t3, 1f
```

Deklarativ programmering

Koden är en beskrivning av programmets beteende/resultat snarare hur det kommer fram till det.

Exempel: Prolog

```
even(0) :- !.
even(1) :- false, !.
even(N) :- !, M is N - 2, even(M).
```

a = b + c

Imperativ programmering

En *förändring* av a:

$$a \leftarrow b + c$$

Deklarativ programmering

Ett *påstående* om a:

$$a = b + c$$

Idag: *Funktionell programmering*

<table>
<tr><td colspan="4">Programmeringsparadigm</td></tr>
<tr><td colspan="2">Imperativa</td><td colspan="2">Deklarativa</td></tr>
<tr><td>Procedurori-
enterade</td><td>Objektorien-
terade</td><td>Funktionella</td><td>Logiska</td></tr>
</table>

Funktionell programmering & Haskell

- Centrerat kring *rena funktioner*.
- Rena funktioner har inga sidoeffekter.
- Rena funktioner ger alltid samma resultat för samma indata.
- Det finns inget globalt tillstånd.
- Värden är oföränderliga (immutable).

Exempel på en funktion i Haskell

```
next :: Int -> Int
next x = x + 1
```

Pattern matching & rekursion

- *Rekursion* används istället för *iteration*
- Värden kan matchas beroende på deras struktur

```haskell
sumNums :: [Int] -> Int
sumNums [] = 0
sumNums (head : tail) = head + sumNums tail
```

```
ghci> sumNums [1, 4, 10, 3]
18
```

En funktion i en variabel skrivs som

$$f : X \to Y$$

$$y = f(x)$$

```
f :: X -> Y
y = f (x)
```

Funktioner i flera variabler kan skrivas som

$$f : X \times Y \to Z$$

$$z = f(x, y)$$

```
f :: (X, Y) -> Z
z = f (x, y)
```

I Haskell skrivs funktioner i flera variabler oftast i *curried form*

$$f : X \to Y \to Z$$

$$z = f(x)(y)$$

```
f :: X -> Y -> Z
z = f x y
```

f är en funktion i en variabel, och som returnerar en funktion i en variabel

Att gå från $f : X \times Y \to Z$ till $f : X \to Y \to Z$ kallas för *currying*

Värden beräknas först när de krävs

```
takeN :: [Int] -> Int -> [Int]
takeN [] _ = []
takeN _ 0 = []
takeN (hd : tl) n = hd : takeN tl (n - 1)
```

```
ghci> takeN [1..] 5
[1,2,3,4,5]
```

Funktioner är värden

```
myMap :: [a] -> (a -> b) -> [b]
myMap [] _ = []
myMap (hd : tl) f = f hd : myMap tl f
```

```
ghci> myMap [1, 4, 2] addOne
[2,5,3]
```

Partial application

`(+)` är en funktion som tar *två* parametrar

```
(+) :: Int -> Int -> Int

addOne :: Int -> Int
addOne x = 1 + x
```

```
ghci> addOne 2
3
```

Partial application

Vi ger `(+)` *ett* argument och får en funktion som tar *en* parameter

```
(+) :: Int -> (Int -> Int)

addOne :: Int -> Int
addOne = 1 +
```

```
ghci> addOne 2
3
```

Referential transparency

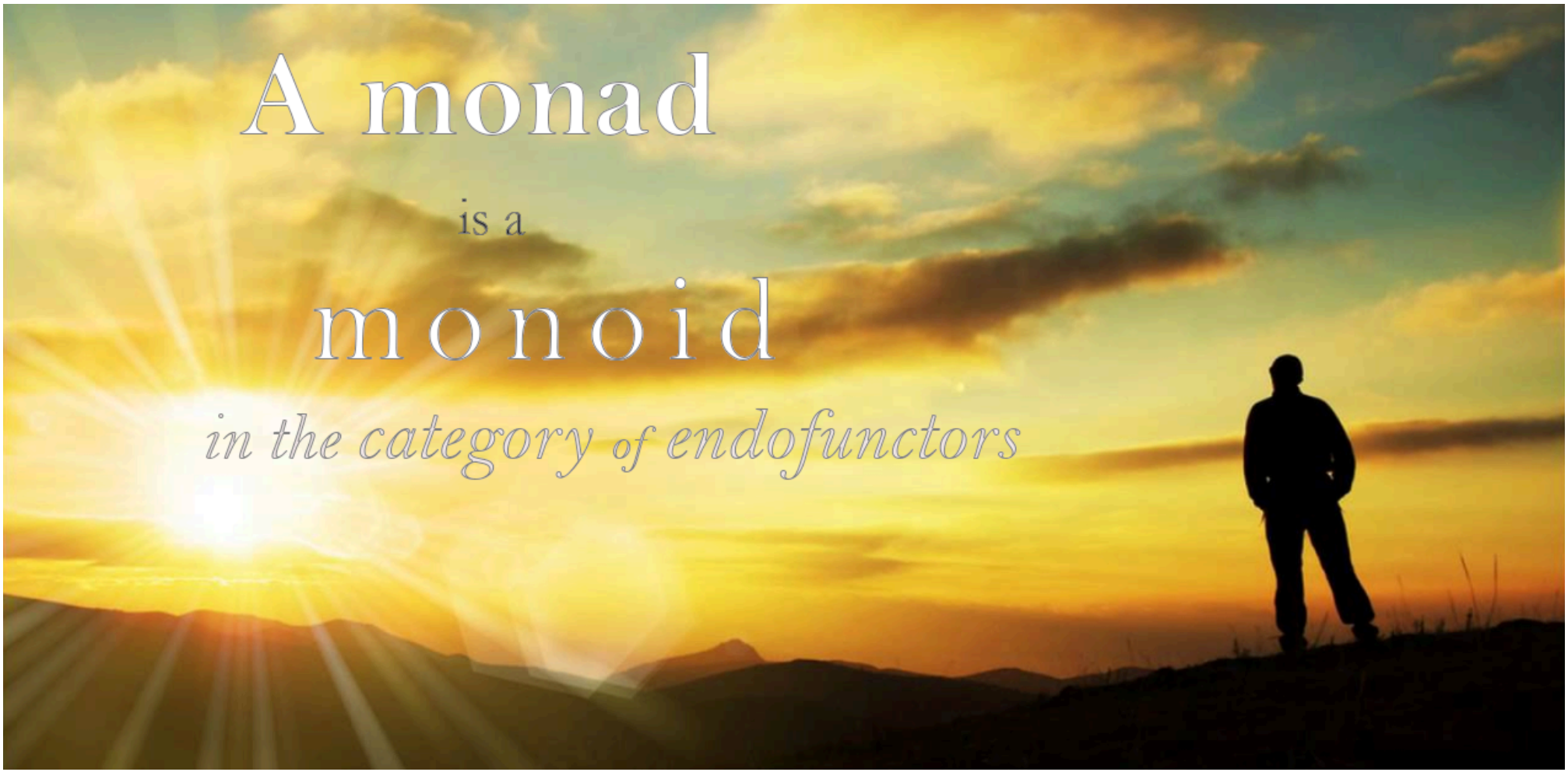

```haskell
addOne :: Int -> Int
addOne x = x + y
  where
    y = 1
```

```haskell
addOne :: Int -> Int
addOne x = x + 1
```

```c
int add_one(int x)
{
        int y = 0;
        y = y + 1;
        return x + y;
}
```

```c
int add_one(int x)
{
        0 = 0 + 1;
        return x + 0;
}
```

Monader


A monad
is a
monoid
in the category of endofunctors

Monader - motivering

```haskell
data Maybe a = Nothing | Just a
```

Motsvarar `Option<T>` i Rust, `std::optional<T>` i C++

Vi vill ha en funktion som adderar två `Maybe Int`

```
addMaybes :: Maybe Int -> Maybe Int -> Maybe Int
addMaybes a b = case a of
    Nothing  -> Nothing
    Just x   -> case b of
        Nothing -> Nothing
        Just y  -> Just (x + y)
```

Monader - definition

En `class` är samling av krav som definierar en typklass

```
class Monad m where
    (>>=)   :: m a -> (a -> m b) -> m b
    return  :: a -> m a
```

`m` är en `Monad` om det finns funktioner `>>=` (även kallad `bind`) och `return` som har de givna signaturerna

Motsvarar ungefär ett Rust `trait` eller C++ `concept`

```haskell
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

Syntaxen på sista raden kallas point-free style. Kom ihåg referential transparency:

- `return x = Just x` $\implies$ `(return 1) = (Just 1)`
- `return = Just` $\implies$ `(return) 1 = (Just) 1`

```haskell
addMaybe :: Maybe Int -> Int -> Maybe Int
addMaybe m y = m >>= (\x -> Just (x + y))
```

```haskell
addMaybes :: Maybe Int -> Maybe Int -> Maybe Int
addMaybes a b = a >>= (\x -> (b >>= (\y -> Just (x + y))))
```

`(\x -> ...)` är ett lambda (en anonym funktion)

Ett enklare sätt att använda monader är do-syntax.

Liknar imperativ syntax, men är ekvivalent med det funktionella exemplet.

```haskell
addMaybes :: Maybe Int -> Maybe Int -> Maybe Int
addMaybes a b = do
    x <- a
    y <- b
    return (x + y)
```

```
readInput :: () -> String
readInput () = ???
```

Vi har bara rena funktioner. `readInput` returnerar alltid samma värde. Hur kan vi göra IO?

En abstraktion

`readInput` är en funktion av omvärlden innan indata som ger omvärlden efter indata

```
data World = ...

readInput :: World -> (World, String)
readInput world = (new_world, input_string)
```

Den nya omvärlden är i denna modell en ren funktion av den gamla omvärlden

Om vi kapslar in funktioner mellan världstillstånd i monader kan vi enkelt göra sekvenser och behöver aldrig röra `World`

```
newtype WorldAction a = Act (World -> (World, a))

readInput :: WorldAction String
procedure :: WorldAction ()
procedure = do
    a <- readInput
    b <- readInput
    return ()
```

`World` kan vara en abstrakt datatyp utan storlek

I Haskell heter den här abstraktionen egentligen `IO`

```haskell
getLine :: IO String
putStrLn :: String -> IO ()

main :: IO ()
main = do
    msg <- getLine
    putStrLn msg
```

Med `Just`-konstruktorn kan vi ta ett värde ur en `Maybe`

```
getValue :: Maybe a -> a
getValue m = x
    where (Just x) = m
```

Kan vi få en `String` från en `IO String`?

```
getLinePure :: IO String -> String
getLinePure m = x
    where (??? x) = m
```

Rekursion *vs* Iteration

```
void clear(List *l)
{
    if (l == NULL)
        return;

    l->data = 0;
    l = l->next;

    clear(l);
}
```

```
void clear(List *l)
{
    while (l != NULL)
    {
        l->data = 0;
        l = l->next;
    }
}
```

Är denna funktion svansrekursiv?

```
sumNums :: [Int] -> Int
sumNums [] = 0
sumNums (hd : tl) = hd + sum tl
```

Den här då?

```
sumNums :: [Int] -> Int -> Int
sumNums [] a = a
sumNums (hd : tl) a = sumNums tl (hd + a)
```

```
ghci> sumNums [1, 2, 3, 4] 0
10
```

Blir det snabbare?

Svansrekursion & strikt evaluering


```haskell
sumNums :: [Int] -> Int -> Int
sumNums [] a = a
sumNums (hd : tl) a = sumNums tl (hd + a)
```

En kedja av *thunks* byggs upp:

```
a =   0
a =   0 + 1
a =  (0 + 1) + 2
a = ((0 + 1) + 2) + 3
          ...
```

Strikt evaluering kan användas för att kringgå lazy evaluation

```
sumNums :: [Int] -> Int -> Int
sumNums [] a = a
sumNums (hd : tl) a = sumNums tl $! (hd + a)
```

```
ghci> sumNums [1, 2, 3, 4] 0
10
```

$!

Svansrekursion & strikt evaluering


```
Benchmark 1: ./without-tl
  Time (mean ± σ):      58.3 ms ±   6.9 ms    [User: 41.0 ms, System: 10.9 ms]
  Range (min … max):    43.3 ms …  75.5 ms    52 runs

Benchmark 2: ./with-tl
  Time (mean ± σ):      25.3 ms ±   5.1 ms    [User: 14.4 ms, System: 4.4 ms]
  Range (min … max):    22.4 ms …  43.1 ms    113 runs

Benchmark 3: ./with-tl-strict
  Time (mean ± σ):      24.9 ms ±   4.8 ms    [User: 13.7 ms, System: 4.6 ms]
  Range (min … max):    22.3 ms …  44.6 ms    108 runs

Summary
  ./with-tl-strict ran
    1.02 ± 0.28 times faster than ./with-tl
    2.34 ± 0.53 times faster than ./without-tl
```

Läxa

1. Skriv Haskellfunktioner för följande problem:
   - Beräkna ett fibonaccital F_n för något n.
   - Vänd på en lista.
   - Hitta medianlängden av alla ord i en lista.
2. Lös två Kattis-uppgifter i Haskell
   - Välj ett problem från tidigare läxor eller från Open Kattis.
   - Länka till Kattislösningen i en kommentar i koden.
3. Lägg allt i ett repo med namnet `<kth-id>-functional`.