Programmering og Problemløsning

4.2+3: White-box testing, håndkøring, kaldestakken og -bunken, højereordens- og anonyme funktioner

White-box (unit) testing

1. Beslut hvilke units, der skal afprøves
2. Identificer forgreningspunkter
3. Lav inputeksempler for alle units, som afprøver hver forgreningsvej, og notér det forventede output
4. Skriv et program, som kører koden med alle inputeksempler, og sammenlign resultatet med det forventede output

```
module convert

/// Convert a non-negative integer into its
/// binary form. E.g., dec2bin 3 =  "0b11"
let dec2bin n =
  if n < 0                          (* WB: 1 *)
    "Illegal value"
  elif n = 0 then                   (* WB: 2 *)
    "0b0"
  else
    let mutable v = n
    let mutable str = ""
    while v > 0 do                  (* WB: 3 *)
      str <- (string (v % 2)) + str
      v <- v / 2
    "0b" + str
```

Unit	Branch	Condition	Input	Expected output	Comment
dec2bin	1	n < 0			
	1a	true	-1	"Illegal value"	
	1b	false			-> Branch 2
	2 (n>=0)	n = 0			
	2a	true	0	"0b0"	
	2b	false			-> Branch 3
	3 (n>0)	v > 0			
	3a	true	1	"0b1"	1 or more
	3b	false			0 times, impossible.

White-box (unit) testing

Unit	Branch	Condition	Input	Expected output	Comment
dec2bin	1	n < 0			
	1a	true	-1	"Illegal value"	
	1b	false			-> Branch 2
	2 (n>=0)	n = 0			
	2a	true	0	"0b0"	
	2b	false			-> Branch 3
	3 (n>0)	v > 0			
	3a	true	1	"0b1"	1 or more
	3b	false			0 times, impossible.

```
open convert

printfn "White-box testing of dec2bin.fsx"
printfn "  Unit: dec2bin"
printfn "    %5b: Branch 1a" (dec2bin -1 = "Illegal value")
printfn "    %5b: Branch 2a" (dec2bin 0 = "0b0")
printfn "    %5b: Branch 3a" (dec2bin 1 = "0b1")
```

```
$ fsharpc -a dec2binWhite.fs
$ fsharpc -r dec2binWhite.dll dec2binWhiteTest.fsx
$ mono dec2binWhiteTest.exe
White-box testing of dec2bin.fsx
  Unit: dec2bin
    true: Branch 1a
    true: Branch 2a
    true: Branch 3a
```

Closures = funktioner som værdier

En simple function:

```
let N = 3
let doit n =
  for i = 1 to n do
    let p = i * i
    printfn "%d: %d" i p

doit N
```

Closure notation:

Navn = (input, krop, virkefeltets værdier)

Værdier:

```
N = 3
doit = ((n), (for i = 1 to n do let p = i * i in printfn "%d: %d" i p), (N=3))
it = ()
```

Håndkøring: simulér computeren

```
1  let N = 3
2  let doit n =
3    for i = 1 to n do
4      let p = i * i
5      printfn "%d: %d" i p
6
7  doit N
```

```
$ fsharpi simpleForLoop.fsx
1: 1
2: 4
3: 9
```

- ~~E0~~: ()
1 N = 3
2 doit = ((n), doit-body, (N=3))
7 doit N = ~~?~~ ()
2 ~~E1~~: doit-body, (n = 3, N = 3)
3 ~~E2~~: for-body, (n = 3, N = 3, i = 1)
4 p = 1
5 output = “1: 1”
3 ~~E3~~: for-body, (n = 3, N = 3, i = 2)
4 p = 4
5 output = “2: 4”
3 ~~E4~~: for-body, (n = 3, N = 3, i = 3)
4 p = 9
5 output = “3: 9”
- return = ()
- return = ()

Leksikografisk versus Dynamisk Virkefelt

Leksikografisk

```
let testScope x =
  let a = 3.0
  let f z = a * z
  let a = 4.0
  f x
printfn "%A" (testScope 2.0)
```

Dynamisk

```
let testScope x =
  let mutable a = 3.0
  let f z = a * z
  a <- 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
let testScope x = let a = 3.0 in let f z = a * z in let a = 4.0 in f x in printfn "%A" (testScope 2.0)
```

Håndkøring: Leksikografisk virkefelt

```
1 let testScope x =
2   let a = 3.0
3   let f z = a * z
4   let a = 4.0
5   f x
6 printfn "%A" (testScope 2.0)
```

```
$ fsharpi lexicalScopeTracing.fsx
6.0
```

```
- E0:
1   testScope = ((x), testScope-body, ())
6   testScope 2.0 = ? 6.0
6   E1: testScope-body, (x = 2.0)
2     a = 3.0
3     f = ((z), a * z, (a -> 3.0))
4     a = 4.0
5     f 2.0 = ? 6.0
5     E2: f-body (z = 2.0, a = 3.0)
3       return = 6.0
5     return = 6.0
-   return = ()
```

Håndkøring: Dynamisk virkefelt

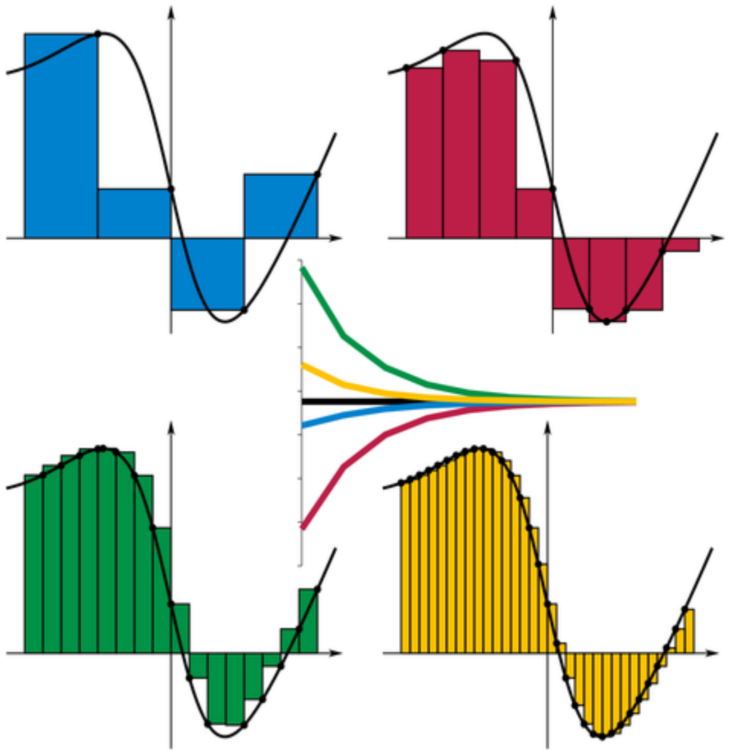
```
1  let testScope x =
2    let mutable a = 3.0
3    let f z = a * z
4    a <- 4.0
5    f x
6  printfn "%A" (testScope 2.0)
```

```
$ fsharpi dynamicScopeTracing.fsx
8.0
```

\- ~~E0~~:
1 testScope = ((x), testScope-body, ())
6 testScope 2.0 = ~~?~~ 8.0
6 ~~E1~~: testScope-body, (x = 2.0)
2 a = alpha
3 f = ((z), a * z, (a -> alpha))
5 f 2.0 = ~~?~~ 8.0
5 ~~E2~~: f-body (z = 2.0, a = alpha)
3 return = 8.0
5 return = 8.0
\- return = ()

Linje	Navn	Værdi
2	alpha	3.0
4	alpha	4.0

Højereordens funktioner




```
/// Estimate the integral of f
/// from a to b with stepsize d
let integrate f a b d =
  let mutable sum = 0.0
  let mutable x = a
  while x < b do
    sum <- sum + d * (f x)
    x <- x + d
  sum

let a = 0.0
let b = 1.0
let d = 0.01
let result = integrate exp a b d
printfn "Int_%g^%g exp(x) dx = %g" a b result
```

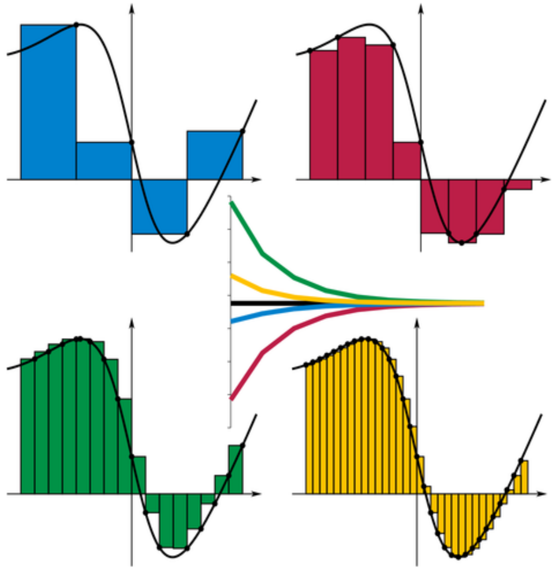
Højereordens funktioner

```
/// Estimate the integral of f
/// from a to b with stepsize d
let integrate f a b d =
  let mutable sum = 0.0
  let mutable x = a
  while x < b do
    sum <- sum + d * (f x)
    x <- x + d
  sum

let a = 0.0
let b = 1.0
let d = 0.01
let result = integrate exp a b d
printfn "Int_%g^%g exp(x) dx = %g" a b
result
```

```
/// Estimate the integral of f
/// from a to b with stepsize d
let integrate f a b d =
  let mutable sum = 0.0
  let mutable x = a
  while x < b do
    sum <- sum + d * (f x)
    x <- x + d
  sum

let a = 0.0
let b = 1.0
let truth = exp 1.0 - 1.0
for e = 0 to 6 do
  let d = 10.0**(float -e)
  let result = truth - integrate exp a b d
  printfn "d = %e: exp 1.0 - 1.0 - Int_%g^%g exp(x) dx = %g" d a b result
```



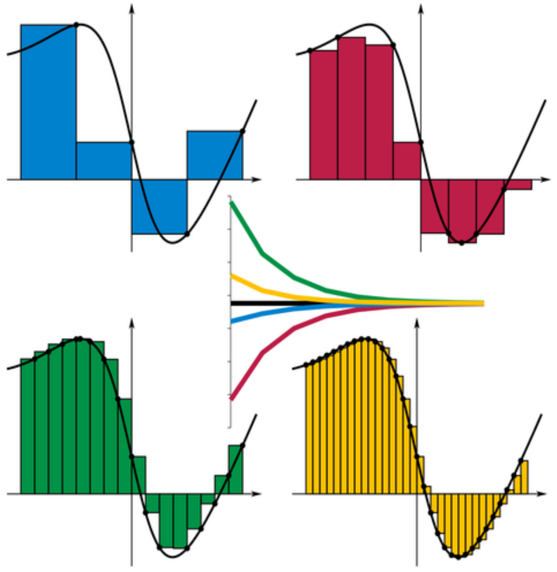
Anonyme funktioner

```
let f x = x * exp(x)
f 3.0
```

```
let f = fun x -> x * exp(x)
f 3.0
```

```
/// Estimate the integral of f
/// from a to b with stepsize d
let integrate f a b d =
  let mutable sum = 0.0
  let mutable x = a
  while x < b do
    sum <- sum + d * (f x)
    x <- x + d
  sum

let a = 0.0
let b = 1.0
let d = 1e-5
let result = integrate (fun x -> x * exp(x)) a b d
printfn "Int_%g^%g f(x) dx = %g" a b result
```



Kald-stakken (værdier og variable)

Stakken (The Stack)

```
1 let f x =
2   x*x
3 let g x =
4   let a = -1.0/2.0
5   exp (a * f x)
6 printfn "%g" (g 2.0)
```

g 2.0 = ?

x = 2.0 a = -0.5 retur = udtryk i l. 6
g 2.0 = ?

x = 2.0 retur = udtryk i l. 5
f x = ? x = 2.0 a = -0.5 retur = udtryk i l. 6
g 2.0 = ?

4.0
f x = ? x = 2.0 a = -0.5 retur = udtryk i l. 6
g 2.0 = ?

0.135335
g 2.0 = ?

Referenceceller

Bunken (The Heap)

```
1 let g a x =
2   a := -1.0/2.0
3   exp (!a * x * x)
4 let a = ref -1.0
5 printfn "%g" (g a 2.0)
6 printfn "%g" !a
```

a = ref 1
g a 2.0 = ?

a = ref 1
x = 2.0
retur = udtryk i l. 5

a = ref 1
g a 2.0 = ?

0.135335

a = ref 1
g a 2.0 = ?

ref 1 = ~~-1.0~~ -0.5

Hvad sker der?

```
let incr =
  let mutable counter = 0
  fun () ->
    counter <- counter + 1
    counter

printfn "%d" (incr ())
printfn "%d" (incr ())
printfn "%d" (incr ())
```

```
$ fsharpi inc.fsx
1
2
3
```