Introduktion till Haskell

Elias Riedel Gårding

NMA11, Teknisk Fysik (KTH) 2014

18 februari 2018

Om Haskell

Historik

- 1927: Haskell Curry börjar undersöka kombinator-logik
- Ca 1930: Alonzo Church uppfinner λ-kalkyl
- Ca 1960: Lisp, "första funktionella språket"
- Ca 1980: Scheme (variant på Lisp) och ML uppfinns
- Ca 1980-1987: Många nya språk uppfinns, alla har sitt eget språk
- 1987: Gemensamt språk behövs! Första Haskell-mötet
- 1999: *The Haskell 98 Report*
- 2002: *The Revised Haskell 98 Report*
- 2002 och framåt: Svårt att överblicka...

Haskell B. Curry

$$((\mathbf{S}\ \mathbf{K}\ \mathbf{K})\ x) = (\mathbf{S}\ \mathbf{K}\ \mathbf{K}\ x) = (\mathbf{K}\ x\ (\mathbf{K}\ x)) = x$$

Alonzo Church

$$(\lambda x.(y\,x)\ \lambda x.(u\,x))$$

Varför lära sig Haskell?

- Elegant och roligt att programmera i
- Matematiskt sammanhängande och intressant
- Lätt att skriva välstrukturerad kod
- **Svårt att skriva buggig kod**
- Man blir bättre på andra språk
- Annorlunda; nya perspektiv på programmering!

Motiverande exempel!

Fibonacci-tal

Definition av talserien: $F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$

Motiverande exempel!

Fibonacci-tal

Definition av talserien: $F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$

De n första Fibonacci-talen?

```
def firstFibs(n):
    result = [0,1]
    for i in range(n-2):
        result.append(result[-2] + result[-1])
    return result
```

Motiverande exempel!

Fibonacci-tal

Definition av talserien: $F_0 = 0$ $F_1 = 1$ $F_n = F_{n-1} + F_{n-2}$

De n första Fibonacci-talen?

```
def firstFibs(n):
    result = [0,1]
    for i in range(n-2):
        result.append(result[-2] + result[-1])
    return result
```

Alla Fibonacci-tal mindre än n?

```
def fibsLessThan(n):
    result = []
    if n > 0:
        result.append(0)
    if n > 1:
        result.append(1)
    while result[-2] + result[-1] < n:
        result.append(result[-2] + result[-1])
    return result
```

Motiverande exempel!

Fibonacci-tal

Definition av talserien: $F_0 = 0$ $F_1 = 1$ $F_n = F_{n-1} + F_{n-2}$

De n första Fibonacci-talen?

```
def firstFibs(n):
    result = [0,1]
    for i in range(n-2):
        result.append(result[-2] + result[-1])
    return result
```

Alla Fibonacci-tal mindre än n?

```
def fibsLessThan(n):
    result = []
    if n > 0:
        result.append(0)
    if n > 1:
        result.append(1)
    while result[-2] + result[-1] < n:
        result.append(result[-2] + result[-1])
    return result
```

```
-- Trolleri som ger oändlig lista
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Motiverande exempel!

Fibonacci-tal

Definition av talserien: $F_0 = 0$ $F_1 = 1$ $F_n = F_{n-1} + F_{n-2}$

De n första Fibonacci-talen?

```
def firstFibs(n):
    result = [0,1]
    for i in range(n-2):
        result.append(result[-2] + result[-1])
    return result
```

Alla Fibonacci-tal mindre än n?

```
def fibsLessThan(n):
    result = []
    if n > 0:
        result.append(0)
    if n > 1:
        result.append(1)
    while result[-2] + result[-1] < n:
        result.append(result[-2] + result[-1])
    return result
```

```
-- Trolleri som ger oändlig lista
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

firstFibs n = take n fibs
```

Motiverande exempel!

Fibonacci-tal

Definition av talserien: $F_0 = 0$ $F_1 = 1$ $F_n = F_{n-1} + F_{n-2}$

De n första Fibonacci-talen?

```
def firstFibs(n):
    result = [0,1]
    for i in range(n-2):
        result.append(result[-2] + result[-1])
    return result
```

Alla Fibonacci-tal mindre än n?

```
def fibsLessThan(n):
    result = []
    if n > 0:
        result.append(0)
    if n > 1:
        result.append(1)
    while result[-2] + result[-1] < n:
        result.append(result[-2] + result[-1])
    return result
```

```
-- Trolleri som ger oändlig lista
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

firstFibs n = take n fibs

fibsLessThan n = takeWhile (< n) fibs
```

Klassificering av Haskell

Deklarativ programmering

Imperativ programmering

- Beskriv för datorn *hur man gör*
- Python, Java, C/C++, ...

Deklarativ programmering

- Beskriv för datorn *vad saker är*
- HTML, XML, SQL, Haskell, Prolog, ...

Klassificering av Haskell

Deklarativ programmering

Imperativ programmering

- Beskriv för datorn *hur man gör*
- Python, Java, C/C++, ...

Deklarativ programmering

- Beskriv för datorn *vad saker är*
- HTML, XML, SQL, Haskell, Prolog, ...

```
# Beräknar n! = 1*2*...*n
def fact(n):
    prod = 1
    for k in range(1, n+1):
        prod *= k
    return prod
```

Klassificering av Haskell

Deklarativ programmering

Imperativ programmering

- Beskriv för datorn *hur man gör*
- Python, Java, C/C++, ...

Deklarativ programmering

- Beskriv för datorn *vad saker är*
- HTML, XML, SQL, Haskell, Prolog, ...

```python
# Beräknar n! = 1*2*...*n
def fact(n):
    prod = 1
    for k in range(1, n+1):
        prod *= k
    return prod
```

```haskell
fact n = product [1..n]
```

Klassificering av Haskell

Deklarativ programmering

Imperativ programmering

- Beskriv för datorn *hur man gör*
- Python, Java, C/C++, ...

Deklarativ programmering

- Beskriv för datorn *vad saker är*
- HTML, XML, SQL, Haskell, Prolog, ...

```
# Beräknar n! = 1*2*...*n
def fact(n):
    prod = 1
    for k in range(1, n+1):
        prod *= k
    return prod
```

```
fact n = product [1..n]
```

```
fact 0 = 1
fact n = n * fact (n-1)
```

Procedurell programmering

- Dela upp kod i funktioner
- Python, lite Java, C/C++,

  ...

Klassificering av Haskell

Ren funktionell programmering

Procedurell programmering

- Dela upp kod i funktioner
- Python, lite Java, C/C++, ...

Funktionell programmering

- Dela upp kod i *riktiga* funktioner
- Varje funktion är *helt* isolerad
- Lisp (någorlunda), Haskell, Idris...

Klassificering av Haskell

Ren funktionell programmering

Procedurell programmering

- Dela upp kod i funktioner
- Python, lite Java, C/C++, ...

Funktionell programmering

- Dela upp kod i *riktiga* funktioner
- Varje funktion är *helt* isolerad
- Lisp (någorlunda), Haskell, Idris...

Matematik

$f(x) = \ldots$

Klassificering av Haskell

Ren funktionell programmering

Procedurell programmering

- Dela upp kod i funktioner
- Python, lite Java, C/C++, ...

Funktionell programmering

- Dela upp kod i *riktiga* funktioner
- Varje funktion är *helt* isolerad
- Lisp (någorlunda), Haskell, Idris...

Matematik

$f(x) = \ldots$

Haskell

```
f x = ...
```

Klassificering av Haskell

Ren funktionell programmering

Procedurell programmering

- Dela upp kod i funktioner
- Python, lite Java, C/C++, ...

Funktionell programmering

- Dela upp kod i *riktiga* funktioner
- Varje funktion är *helt* isolerad
- Lisp (någorlunda), Haskell, Idris...

Matematik

$f(x) = \ldots$

Haskell

```
f x = ...
```

Python

```
def f(x):
    # enbart return
    return ...
```

Klassificering av Haskell

Ren funktionell programmering

Procedurell programmering

- Dela upp kod i funktioner
- Python, lite Java, C/C++, ...

Funktionell programmering

- Dela upp kod i *riktiga* funktioner
- Varje funktion är *helt* isolerad
- Lisp (någorlunda), Haskell, Idris...

Matematik

$f(x) = \ldots$

Haskell

```
f x = ...
```

Python

```
def f(x):
    # enbart return
    return ...
```

Väldigt begränsat jämfört med Python.

Klassificering av Haskell

Ren funktionell programmering

Procedurell programmering

- Dela upp kod i funktioner
- Python, lite Java, C/C++, ...

Funktionell programmering

- Dela upp kod i *riktiga* funktioner
- Varje funktion är *helt* isolerad
- Lisp (någorlunda), Haskell, Idris...

Matematik

$f(x) = \ldots$

Haskell

```
f x = ...
```

Python

```
def f(x):
    # enbart return
    return ...
```

Väldigt begränsat jämfört med Python.

Men begränsningar är bra för kreativiteten!

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \ \texttt{+=} \ x\}$

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \ \texttt{+=} \ x\}$

$f(x) = x + y$ OK

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \ \texttt{+=} \ x\}$

$f(x) = x + y$ OK

*"Det finns ingen **tid/förändring** i Haskell"*

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \texttt{ += } x\}$

$f(x) = x + y$ OK

*"Det finns ingen **tid/förändring** i Haskell"*

Ingen tid $\implies$ inga `for`, `while`.

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \texttt{ += } x\}$

$f(x) = x + y$ OK

*"Det finns ingen **tid/förändring** i Haskell"*

Ingen tid $\implies$ inga `for`, `while`.

Inga sidoeffekter

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \texttt{ += } x\}$

$f(x) = x + y$ OK

*"Det finns ingen **tid/förändring** i Haskell"*

Ingen tid $\implies$ inga `for`, `while`.

Inga sidoeffekter

$f(x) = \{\texttt{print(``Hello world!'')}\texttt{; return } x + 3\}$

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \texttt{ += } x\}$

$f(x) = x + y$ OK

*"Det finns ingen **tid/förändring** i Haskell"*

Ingen tid $\implies$ inga `for`, `while`.

Inga sidoeffekter

$f(x) = \{\texttt{print("Hello world!"); return } x + 3\}$

Samma argument ger samma returvärde

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \texttt{ += } x\}$

$f(x) = x + y$ OK

*"Det finns ingen **tid/förändring** i Haskell"*

Ingen tid $\implies$ inga `for`, `while`.

Inga sidoeffekter

$f(x) = \{\texttt{print("Hello world!"); return } x + 3\}$

Samma argument ger samma returvärde

$f(x) = \texttt{x * random()}$

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \texttt{ += } x\}$

$f(x) = x + y$ OK

*"Det finns ingen **tid/förändring** i Haskell"*

Ingen tid $\implies$ inga `for`, `while`.

Inga sidoeffekter

$f(x) = \{\texttt{print(``Hello world!'')}\texttt{; return } x + 3\}$

Samma argument ger samma returvärde

$f(x) = \texttt{x * random()}$

$f(x) = \texttt{read(``file.txt'')}$

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \texttt{ += } x\}$

$f(x) = x + y$ OK

*"Det finns ingen **tid/förändring** i Haskell"*

Ingen tid $\Longrightarrow$ inga `for`, `while`.

Inga sidoeffekter

$f(x) = \{\texttt{print(``Hello world!'')}\texttt{; return } x + 3\}$

Samma argument ger samma returvärde

$f(x) = \texttt{x * random()}$

$f(x) = \texttt{read(``file.txt'')}$

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Klassificering av Haskell

Saker man inte kan göra i Haskell

Inga variabler, bara konstanter

$y = 5$

$y = 18$

$f(x) = \{y \texttt{ += } x\}$

$f(x) = x + y$ OK

*"Det finns ingen **tid/förändring** i Haskell"*

Ingen tid $\implies$ inga `for`, `while`.

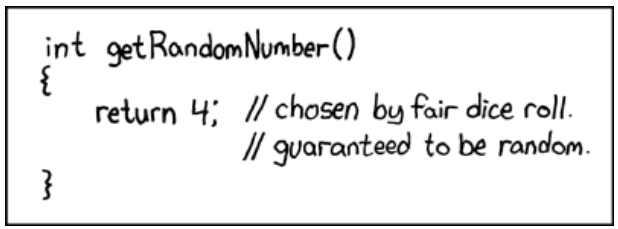
Inga sidoeffekter

$f(x) = \{\texttt{print(``Hello world!'')}\texttt{; return } x + 3\}$

Samma argument ger samma returvärde

$f(x) = \texttt{x * random()}$

$f(x) = \texttt{read(``file.txt'')}$



Nonsens i matematik $\implies$ nonsens i Haskell

Funktionsdefinition

Python

```
def myfunc(x, y, z):
    return x*y*z + 3*(x + z)
```

Haskell

```
myfunc x y z = x*y*z + 3*(x + z)
```

Syntax

`if`-uttryck

```
if x > 0 then "Positivt!" else "Icke-positivt!"
```

Syntax

`if`-uttryck

```
if x > 0 then "Positivt!" else "Icke-positivt!"
```

Notera: `if`-**uttryck**, inte `if`-satser.

Syntax

`if`-uttryck

```
if x > 0 then "Positivt!" else "Icke-positivt!"
```

Notera: `if`-**uttryck**, inte `if`-satser.
Man *måste* ha med `else`-delen.

Syntax

if-uttryck

```
if x > 0 then "Positivt!" else "Icke-positivt!"
```

Notera: if-**uttryck**, inte if-satser.
Man *måste* ha med else-delen.

I gengäld kan if användas varsomhelst:

```
12 + (if x > 0 then 7 else 14 - x)
```

Syntax

if-uttryck

```
if x > 0 then "Positivt!" else "Icke-positivt!"
```

Notera: if-**uttryck**, inte if-satser.
Man *måste* ha med else-delen.

I gengäld kan if användas varsomhelst:

```
12 + (if x > 0 then 7 else 14 - x)
```

Python har också detta: `12 + (7 if x > 0 else 14 - x)`

Matematik

$$|x| = \begin{cases} x & \text{om } x \geq 0 \\ -x & \text{annars} \end{cases}$$

Syntax

Guards

Matematik

$$|x| = \begin{cases} x & \text{om } x \geq 0 \\ -x & \text{annars} \end{cases}$$

Haskell

```
abs x = if x >= 0 then  x
                  else -x
```

Syntax

Guards

Matematik

$$|x| = \begin{cases} x & \text{om } x \geq 0 \\ -x & \text{annars} \end{cases}$$

Haskell

```
abs x = if x >= 0 then  x
                  else -x

abs x
  | x >= 0    =  x
  | otherwise = -x
```

Datatyper

Deklaration och annotation

Haskell har ett *starkt*, *statiskt typsystem* med *typinferens*

Typdeklaration

```
myNumber :: Int -- Valfritt! (men rekommenderat)
myNumber = 17 + 97

myNumber = "ett par sockor" -- Går inte att kompilera
```

Typannotation (“type hints”, inte jätteviktigt)

```
1/2 + 5/7 -- 1.2142857...
(1/2 :: Rational) + 5/7 -- 17 % 14
```

Datatyper

Några inbyggda datatyper

`Bool`: `True` eller `False`

`Int`: 64-bitars heltal (mellan -2^{63} och $2^{63} - 1$)
`Integer`: obegränsade heltal
`Double`: flyttal
`Rational`: exakta rationella tal

`Char`: tecken (`'a'`, `'2'`, `'ö'`, `'%'`, ...)
`String = [Char]`: strängar (`"I am a list of chars."`)

`[Int]`, `[String]`: listor (mer om dessa senare)
`(Int,Int)`, `(String,Int,Char)`: tupler (par, tripler etc)

Datatyper

Funktioner

```haskell
myPolynomial :: Double -> Double
myPolynomial x = x^2 + 2*x + 3

-- argument -> argument -> argument -> returtyp
addAndMultiply :: Int -> Int -> Int -> Int
addAndMultiply a b c = (a + b) * c

ignoreSecond :: String -> Int -> String
ignoreSecond s n = s

-- Funktioner är också datatyper!
applyTo3and4 :: (Int -> Int) -> Int
applyTo3and4 f = f 3 + f 4
```

Datatyper

Typvariabler (polymorfi)

```haskell
-- Typer börjar med stora bokstäver
ignoreSecond :: String -> Int -> String
ignoreSecond s n = s

-- Typvariabler börjar med små bokstäver
ignoreSecond2 :: a -> b -> a
ignoreSecond2 s n = s
```

Lite mer syntax i funktioner

Pattern matching

```haskell
-- Matcha specifika värden
opposite :: Bool -> Bool
opposite False = True
opposite True = False

-- Lägg till ett generellt fall i slutet
isOdd :: Int -> Bool
isOdd 0 = False
isOdd 1 = True
isOdd n = isOdd (n - 2)
```

Labb!

Definiera funktioner

Gå till `http://rextester.com/l/haskell_online_compiler` och implementera så många ni hinner:

```
sign :: Int -> Int
```

$$\text{sign}(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

```
triangle :: Int -> Int
```

$\text{triangle}(n) = \sum_{k=0}^{n} k$

Ledtråd: Samma tanke som `fact`

```
comb :: Int -> Int -> Int
```

$$\text{comb}(n, k) = \binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Ledtråd: Skriv `fact` *separat. Heltalsdivision:* $a/b =$ `a div b`

Listor

Vad är en lista?

En lista av `Int` (en `[Int]`) är

- den tomma listan `[]`, eller
- en `Int` följt av en ny lista: `first : rest`
  (namnkonvention `x : xs`)

Listor

Vad är en lista?

En lista av `Int` (en `[Int]`) är

- den tomma listan `[]`, eller
- en `Int` följt av en ny lista: `first : rest` (namnkonvention `x : xs`)

T.ex. `[1,2,3] == 1 : (2 : (3 : []))`

Matematik

$\{1, \ldots, 6\} = \{1, 2, 3, 4, 5, 6\}$

Matematik

$\{1, \ldots, 6\} = \{1, 2, 3, 4, 5, 6\}$

Haskell

```
[1..6] == [1,2,3,4,5,6]
```

Matematik

$\{1, \ldots, 6\} = \{1, 2, 3, 4, 5, 6\}$

$\{n \in \{1, \ldots, 6\} \mid n + 2 \geq 6\} = \{4, 5, 6\}$

Haskell

```
[1..6] == [1,2,3,4,5,6]
```

Matematik

$\{1, \ldots, 6\} = \{1, 2, 3, 4, 5, 6\}$

$\{n \in \{1, \ldots, 6\} \mid n + 2 \geq 6\} = \{4, 5, 6\}$

Haskell

```
[1..6] == [1,2,3,4,5,6]
```

```
[n | n <- [1..6], n + 2 >= 6] == [4,5,6]
```

Matematik

$\{1, \ldots, 6\} = \{1, 2, 3, 4, 5, 6\}$

$\{n \in \{1, \ldots, 6\} \mid n + 2 \geq 6\} = \{4, 5, 6\}$

$\left\{n^2 \mid n \in \{1, \ldots, 6\},\ n + 2 \geq 6\right\} = \{16, 25, 36\}$

Haskell

```
[1..6] == [1,2,3,4,5,6]
```

```
[n | n <- [1..6], n + 2 >= 6] == [4,5,6]
```

Matematik

$\{1, \ldots, 6\} = \{1, 2, 3, 4, 5, 6\}$
$\{n \in \{1, \ldots, 6\} \mid n + 2 \geq 6\} = \{4, 5, 6\}$
$\left\{n^2 \mid n \in \{1, \ldots, 6\},\ n + 2 \geq 6\right\} = \{16, 25, 36\}$

Haskell

```
[1..6] == [1,2,3,4,5,6]

[n | n <- [1..6], n + 2 >= 6] == [4,5,6]

[n^2 | n <- [1..6], n + 2 >= 6] == [16,25,36]
```

Matematik

Haskell

Matematik

$\#\{1, \ldots, 1000\} = 1000$

Haskell

Matematik

$$\#\{1, \ldots, 1000\} = 1000$$

Haskell

```
length [1..1000] == 1000
```

Listor

Funktioner på listor

Matematik

$$\#\{1, \ldots, 1000\} = 1000$$

$$\sum_{x=1}^{1000} x = 500500$$

Haskell

```
length [1..1000] == 1000
```

Listor

Funktioner på listor

Matematik

$$\#\{1, \ldots, 1000\} = 1000$$

$$\sum_{x=1}^{1000} x = 500500$$

Haskell

```
length [1..1000] == 1000

sum [1..1000] == 500500
```

Listor

Funktioner på listor

Matematik

$$\#\{1, \ldots, 1000\} = 1000$$

$$\sum_{x=1}^{1000} x = 500500$$

???

Haskell

```
length [1..1000] == 1000
```

```
sum [1..1000] == 500500
```

Listor

Funktioner på listor

Matematik

$\#\{1, \ldots, 1000\} = 1000$

$\sum_{x=1}^{1000} x = 500500$

???

Haskell

```
length [1..1000] == 1000

sum [1..1000] == 500500

map f [1,7,4] == [f 1, f 7, f 4]
```

Listor

Funktioner på listor

Matematik

$\#\{1, \ldots, 1000\} = 1000$

$$\sum_{x=1}^{1000} x = 500500$$

???

Haskell

```
length [1..1000] == 1000

sum [1..1000] == 500500

map f [1,7,4] == [f 1, f 7, f 4]

take 2 [1,7,4,3,2] == [1,7]
```

Listor

Funktioner på listor

Matematik

$\#\{1, \ldots, 1000\} = 1000$

$$\sum_{x=1}^{1000} x = 500500$$

???

Haskell

```
length [1..1000] == 1000

sum [1..1000] == 500500

map f [1,7,4] == [f 1, f 7, f 4]

take 2 [1,7,4,3,2] == [1,7]

filter odd [1,7,4,3,2] == [1,7,3]
```

Hur definieras de?

Hur definieras de?

```
length :: [a] -> Int
```

Hur definieras de?

```
length :: [a] -> Int
length [] = 0
```

Hur definieras de?

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Hur definieras de?

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

sum :: [Int] -> Int
```

Hur definieras de?

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

sum :: [Int] -> Int
sum [] = 0
```

Hur definieras de?

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

Hur definieras de?

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

map :: (a -> b) -> [a] -> [b]
```

Hur definieras de?

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

map :: (a -> b) -> [a] -> [b]
map f [] = []
```

Hur definieras de?

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Hur definieras de?

Hur definieras de?

```
take :: Int -> [a] -> [a]
```

Hur definieras de?

```
take :: Int -> [a] -> [a]
take 0 xs = []
```

Hur definieras de?

```
take :: Int -> [a] -> [a]
take 0 xs = []
take n [] = []
```

Hur definieras de?

```
take :: Int -> [a] -> [a]
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs
```

Hur definieras de?

```
take :: Int -> [a] -> [a]
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs


filter :: (a -> Bool) -> [a] -> [a]
```

Hur definieras de?

```
take :: Int -> [a] -> [a]
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
```

Hur definieras de?

```
take :: Int -> [a] -> [a]
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs
```

Labb!

Rekursiva funktioner på listor

Gå till `http://rextester.com/l/haskell_online_compiler` och implementera så många ni hinner:

```
myproduct :: [Int] -> Int
myproduct [] = ?
myproduct (x:xs) = ?

get :: Int -> [a] -> a
get i ls = ? -- ta fram element nr i (ls !! i)

mydrop :: Int -> [a] -> [a]
mydrop n ls = ? -- ta bort n element i början av ls

myreplicate :: Int -> a -> [a]
myreplicate n x = ? -- [x,x,x,...] n gånger

myreverse :: [a] -> [a]
myreverse ls = ? -- vänd listan bak-och-fram (hint: använd ++ eller myplus)

myplus :: [a] -> [a] -> [a]
myplus as bs = ? -- slå ihop listorna (as ++ bs, hint: rekursion på as, inte bs)

mylast :: [a] -> a
mylast [] = error "empty list"
mylast ls = ? -- returnera sista elementet
```

Annat häftigt: Lat evaluering

Tunga beräkningar

Python:

```
x = sum(list(range(1, 10000001)))
```

Haskell:

```
x = sum [1..10000000]
```

Annat häftigt: Lat evaluering

Tunga beräkningar

Python:

```
x = sum(list(range(1, 10000001)))
```

Haskell:

```
x = sum [1..10000000]
```

Är Haskell snabbare?

Annat häftigt: Lat evaluering

Tunga beräkningar

Python:

```
x = sum(list(range(1, 10000001)))
```

Haskell:

```
x = sum [1..10000000]
```

Är Haskell snabbare? Nej, bara latare!

Annat häftigt: Lat evaluering

Tunga beräkningar

Python:

```
x = sum(list(range(1, 10000001)))
```

Haskell:

```
x = sum [1..10000000]
```

Är Haskell snabbare? Nej, bara latare!

Haskell räknar bara ut något när det behövs för en annan uträkning eller när vi frågar om det.

Annat häftigt: Lat evaluering

Oändliga listor

Vi nämnde aldrig något om att listor behövde vara ändliga. . .

Annat häftigt: Lat evaluering

Oändliga listor

Vi nämnde aldrig något om att listor behövde vara ändliga...

```
integers :: [Int]
integers = [1..]
```

Annat häftigt: Lat evaluering

Oändliga listor

Vi nämnde aldrig något om att listor behövde vara ändliga...

```
integers :: [Int]
integers = [1..]

-- Rekursiv definition
integers :: [Int]
integers = integersFrom 1
    where integersFrom n = n : integersFrom (n+1)
```

Annat häftigt: Funktioner, överkurs

Currying: Alla funktioner har *exakt ett argument*!

Annat häftigt: Funktioner, överkurs

Currying: Alla funktioner har *exakt ett argument*!

```
mult :: Int -> Int -> Int
mult x y = x * y
```

Annat häftigt: Funktioner, överkurs

Currying: Alla funktioner har *exakt ett argument*!

```
mult :: Int -> (Int -> Int)
 (mult x) y = x * y
```

Annat häftigt: Funktioner, överkurs

Currying: Alla funktioner har *exakt ett argument*!

```
mult :: Int -> (Int -> Int)
 (mult x) y = x * y
```

Python:

```
def mult(x):
    def multWithX(y):
        return x * y
    return multWithX
```

Annat häftigt: Funktioner, överkurs

Currying: Alla funktioner har *exakt ett argument*!

```
mult :: Int -> (Int -> Int)
 (mult x) y = x * y
```

Python:

```
def mult(x):
    def multWithX(y):
        return x * y
    return multWithX
```

Partiell applicering:

```
triple :: Int -> Int
triple = mult 3
```

Annat häftigt: Funktioner, överkurs

Currying: Alla funktioner har *exakt ett argument*!

```
mult :: Int -> (Int -> Int)
 (mult x) y = x * y
```

Python:

```
def mult(x):
    def multWithX(y):
        return x * y
    return multWithX
```

Partiell applicering:

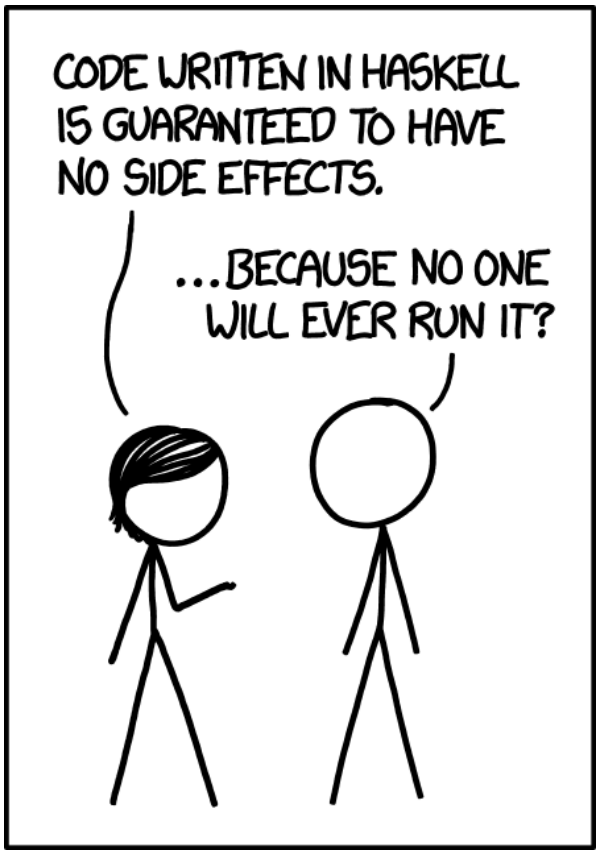
```
triple :: Int -> Int
triple = mult 3
```

Operatorer är också funktioner:

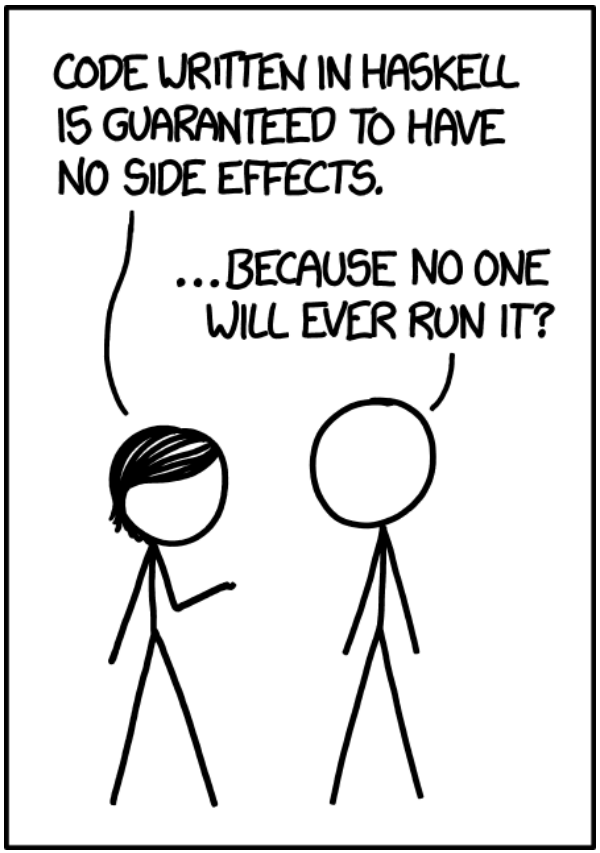
```
(+) :: Int -> Int -> Int
-- (+) a b är samma som a + b
```

Annat häftigt: Input/Output

Hur kan ett Haskell-program *göra* något?



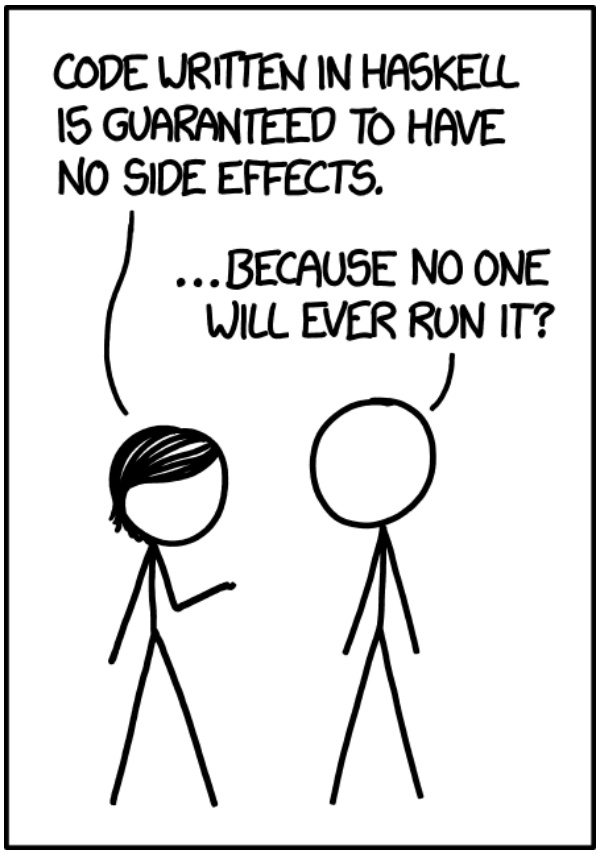
Annat häftigt: Input/Output



Hur kan ett Haskell-program *göra* något?

Fake-datatyp: `World`

Annat häftigt: Input/Output

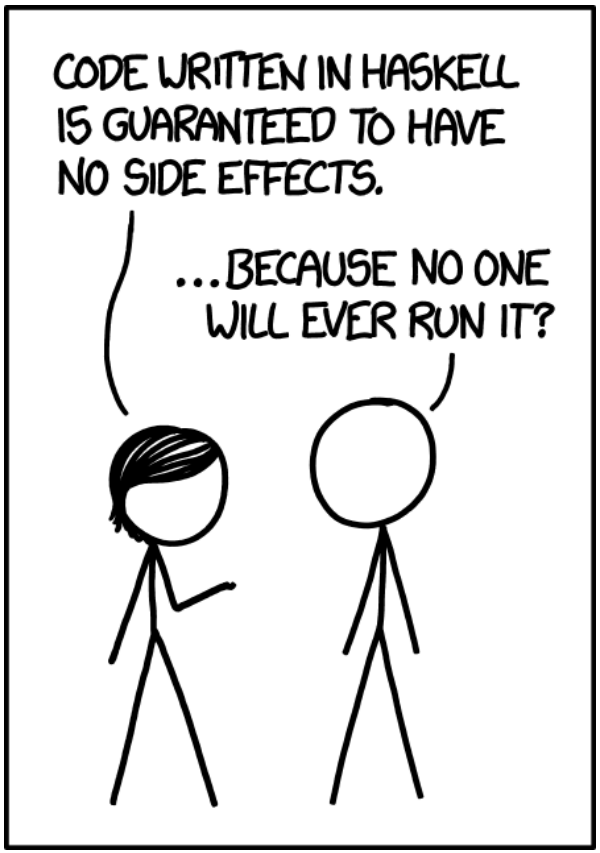


Hur kan ett Haskell-program *göra* något?

Fake-datatyp: `World`

```
getLine :: World -> (String, World)
```

Annat häftigt: Input/Output

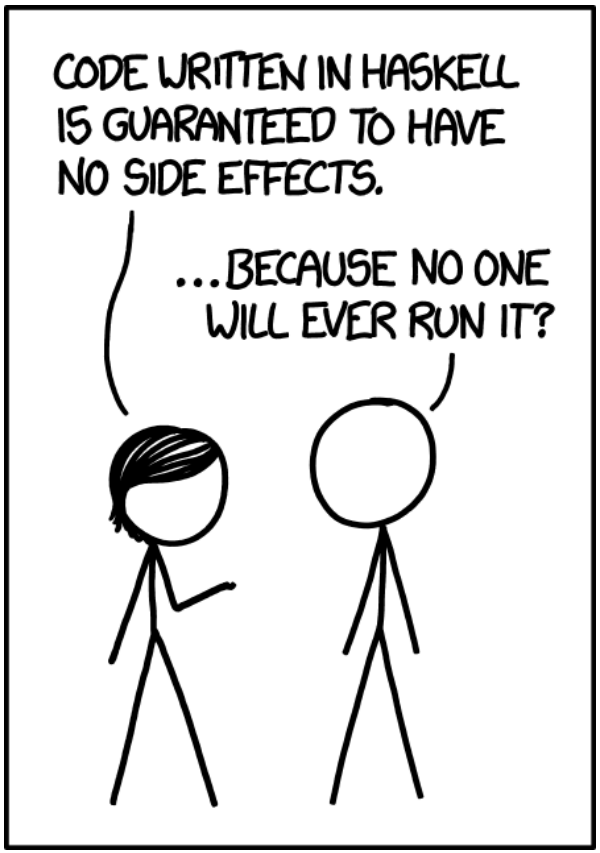


Hur kan ett Haskell-program *göra* något?

Fake-datatyp: `World`

```
getLine :: World -> (String, World)
putStr :: String -> World -> ((), World)
```

Annat häftigt: Input/Output



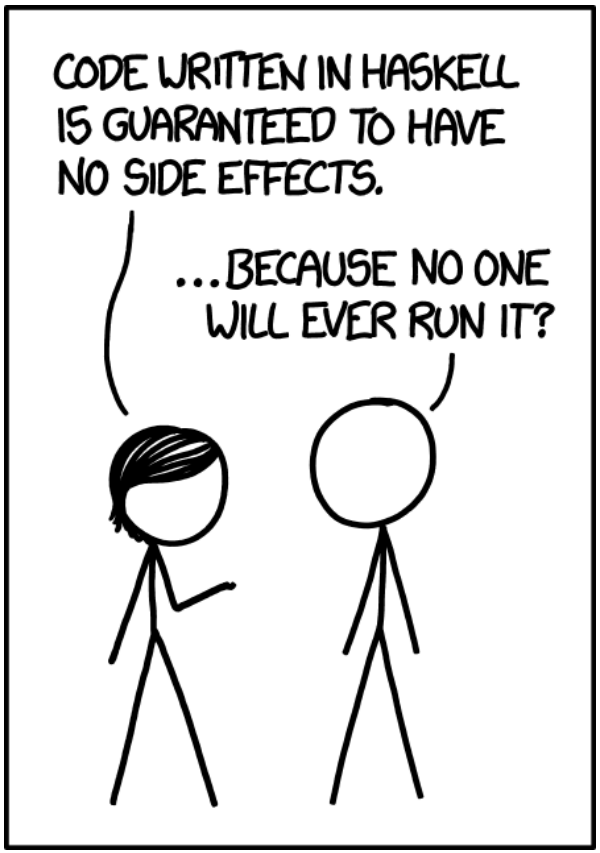
Hur kan ett Haskell-program *göra* något?

Fake-datatyp: `World`

```
getLine :: World -> (String, World)
putStr :: String -> World -> ((), World)
```

`World -> (a, World)` döps till `IO a`

Annat häftigt: Input/Output



Hur kan ett Haskell-program *göra* något?

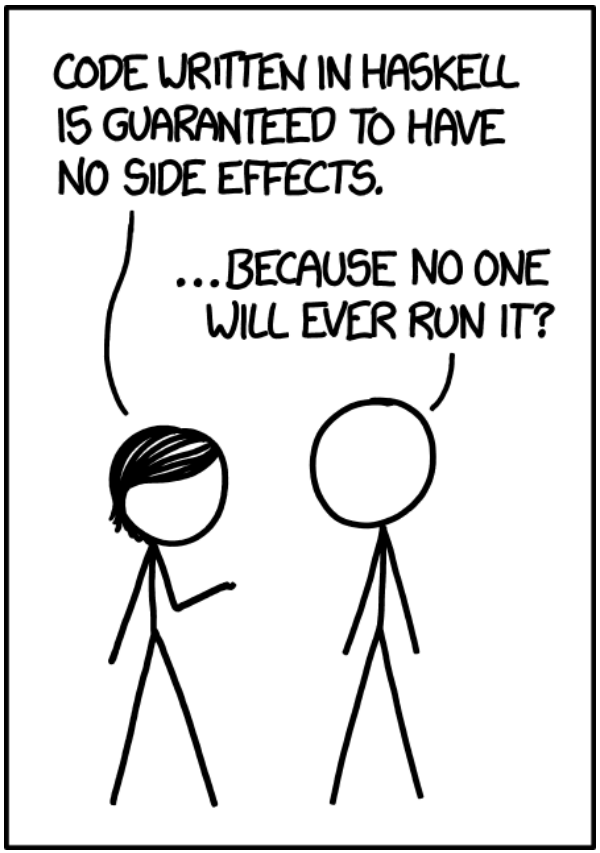
Fake-datatyp: `World`

```
getLine :: World -> (String, World)
putStr :: String -> World -> ((), World)
```

`World -> (a, World)` döps till `IO a`

```
getLine :: IO String
```

Annat häftigt: Input/Output



Hur kan ett Haskell-program *göra* något?

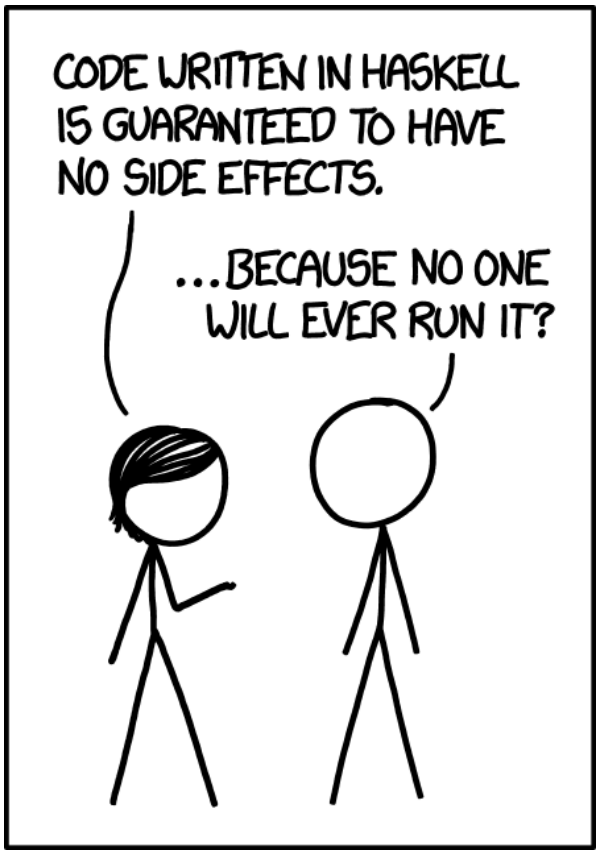
Fake-datatyp: `World`

```
getLine :: World -> (String, World)
putStr :: String -> World -> ((), World)
```

`World -> (a, World)` döps till `IO a`

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

Annat häftigt: Input/Output



Hur kan ett Haskell-program *göra* något?

Fake-datatyp: `World`

```
getLine :: World -> (String, World)
putStr :: String -> World -> ((), World)
```

`World -> (a, World)` döps till `IO a`

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

Inbyggda operatorer för att kombinera:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>) :: IO a -> IO b -> IO b
```

Annat häftigt: Input/Output

Överkurs: Hello World :)

```
main :: IO ()
main = putStrLn "Hej, vad heter du?"
         >> getLine >>= greetName

greetName :: String -> IO ()
greetName name = putStrLn ("Hej, " ++ name ++ "!")
```

Annat häftigt: Input/Output

Överkurs: Hello World :)

```
main :: IO ()
main = putStrLn "Hej, vad heter du?"
         >> getLine >>= greetName

greetName :: String -> IO ()
greetName name = putStrLn ("Hej, " ++ name ++ "!")
```

Speciell syntax:

```
main :: IO ()
main = do
    putStrLn "Hej, vad heter du?"
    name <- getLine
    putStrLn ("Hej, " ++ name ++ "!")
```

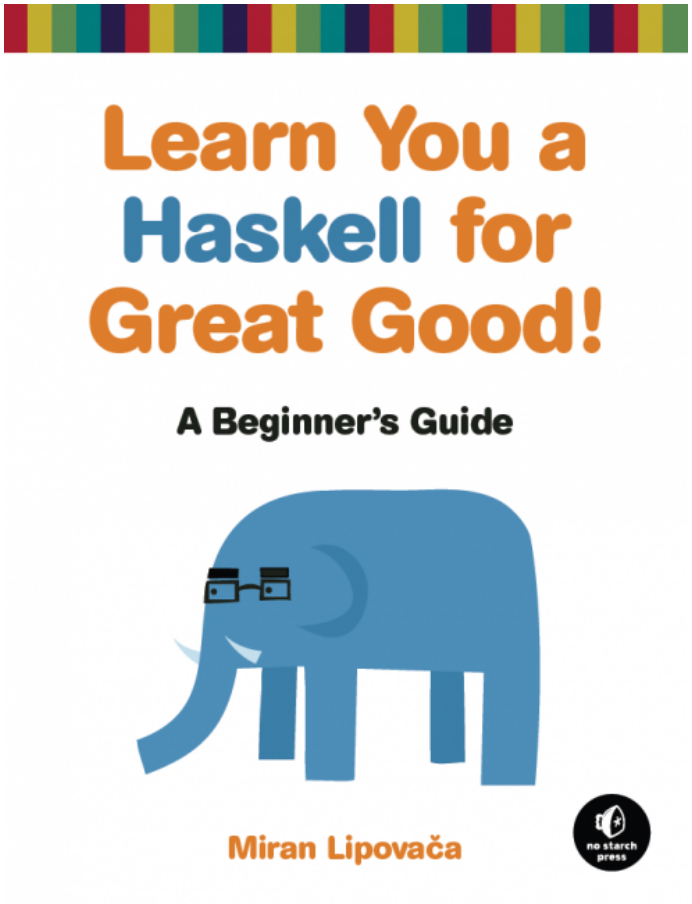
Annat häftigt: Verktyg

Nödvändigt
Kompilator: GHC (*Glasgow Haskell Compiler*)
Interaktivt läge: GHCi
(ingår i **Haskell Platform**, `www.haskell.org`)

Användbart
Hoogle: Sökmotor för funktioner (`www.haskell.org/hoogle`)
Stack: Pakethantering (äldre och sämre: *Cabal*)
QuickCheck (bibliotek): Testa funktioner på slumpade indata
hlint: Föreslå omskrivningar som gör koden snyggare

Slut!

Lära sig mer?



Learn you a Haskell for Great Good
`learnyouahaskell.com`

Real World Haskell (jobbigare)
`book.realworldhaskell.org`