Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

College 2009-2010
1.Introductie

Doaitse Swierstra

Utrecht University

September 13, 2010

Opzet van cursus

College: Doaitse Swierstra

2* per week op maandag (9.00-10.45) en woensdag (9-10.45)

Opzet van cursus

College: Doaitse Swierstra

2* per week op maandag (9.00-10.45) en woensdag (9-10.45)

Werkcollege/Praktium: **Jan Rochel (jan@cs.uu.nl) en Doaitse Swierstra (doaitse@cs.uu.nl)**

2* per week aansluitend aan hoorcollege Let ook hierbij op de zaalwisselingen!!

Lesmateriaal

- dictaat 2010-2011
- introductiepraktikum vanaf website
- errata op website (graag ook aan ons melden)

Lesmateriaal

- dictaat 2010-2011
- introductiepraktikum vanaf website
- errata op website (graag ook aan ons melden)

Website

Veel extra, handige links naar Haskell gerelateerde informatie zoals:

- de taaldefinitie van Haskell98, HackageDB
- de Glasgow Haskell Compiler, manual en bibiliotheken
- de snel groeiende Haskell wiki
- Hoogle, HaYoo

Practicum

Opgaven

- drie opgaven
- uitmondend in een (min of meer) realistische applicatie

Practicum

Opgaven

- drie opgaven
- uitmondend in een (min of meer) realistische applicatie

Hulp bij Practikum

- op werkcollege/praktikum
- mits de vraag goed is voorbereid (zet je vraag van tevoren klaar)
- veelgestelde vragen komen op de website, met hun antwoord

Gebruikte Compiler

De college gaat uit van de taal Haskell98:

Glasgow Haskell Compiler

- veel (belangrijke) uitbreidingen aan Haskell98
- optimaliseert
- ghci is de interpretator versie

Overloading

Omdat Haskell zo'n krachtige taal is:

- zijn veel concepten die normaal onderdeel van de taal zijn, in de taal zelf uitgedrukt
- aan de buitenkant is het verschil niet altijd te zien
- maar wel als het fout gaat

Overloading

Omdat Haskell zo'n krachtige taal is:

- zijn veel concepten die normaal onderdeel van de taal zijn, in de taal zelf uitgedrukt
- aan de buitenkant is het verschil niet altijd te zien
- maar wel als het fout gaat

Op grond van enquetes in voorgaande jaren heeft de huidige aanpak (eerst maar doen of onze neus bloedt, en later de puntjes op de i zetten) de voorkeur gekregen.

Waarom Functioneel Programmeren?

- wat is anders aan dan aan bijv. Java
- een stukje geschiedenis
- waarom nou juist Haskell

Waarom Functioneel Programmeren?

- wat is anders aan dan aan bijv. Java
- een stukje geschiedenis
- waarom nou juist Haskell

Een paar quotes:

- Advanced PL technology is a secret weapon in enterprise computing.

Waarom Functioneel Programmeren?

- wat is anders aan dan aan bijv. Java
- een stukje geschiedenis
- waarom nou juist Haskell

Een paar quotes:

- Advanced PL technology is a secret weapon in enterprise computing.
- Why is a commercial application written in XSLT 10.000 slower than a ML solution?

Wat is er "speciaal"

- in plaats van opdrachten die na elkaar moeten worden uitgevoerd, definiëren we alleen maar functies

Wat is er "speciaal"

- in plaats van opdrachten die na elkaar moeten worden uitgevoerd, definiëren we alleen maar functies
- in plaats van lussen, gebruiken we recursie

Wat is er "speciaal"

- in plaats van opdrachten die na elkaar moeten worden uitgevoerd, definiëren we alleen maar functies
- in plaats van lussen, gebruiken we recursie
- alle functies hebben precies één parameter (al lijkt dat niet zo)

Wat is er "speciaal"

- in plaats van opdrachten die na elkaar moeten worden uitgevoerd, definiëren we alleen maar functies
- in plaats van lussen, gebruiken we recursie
- alle functies hebben precies één parameter (al lijkt dat niet zo)
- functies hebben "alle burgerrechten" (first-class citizens)

Wat is er "speciaal"

- in plaats van opdrachten die na elkaar moeten worden uitgevoerd, definiëren we alleen maar functies
- in plaats van lussen, gebruiken we recursie
- alle functies hebben precies één parameter (al lijkt dat niet zo)
- functies hebben "alle burgerrechten" (first-class citizens)
  1. functies kunnen functies opleveren

Wat is er "speciaal"

- in plaats van opdrachten die na elkaar moeten worden uitgevoerd, definiëren we alleen maar functies
- in plaats van lussen, gebruiken we recursie
- alle functies hebben precies één parameter (al lijkt dat niet zo)
- functies hebben "alle burgerrechten" (first-class citizens)
  1. functies kunnen functies opleveren
  2. functies kunnen functies als parameter meekrijgen

Wat is er "speciaal"

- in plaats van opdrachten die na elkaar moeten worden uitgevoerd, definiëren we alleen maar functies
- in plaats van lussen, gebruiken we recursie
- alle functies hebben precies één parameter (al lijkt dat niet zo)
- functies hebben "alle burgerrechten" (first-class citizens)
  1. functies kunnen functies opleveren
  2. functies kunnen functies als parameter meekrijgen
- van vrijwel alles kunnen we abstraheren

Wat is er "speciaal"

- Haskell kent type inferentie: van de meeste waarden hoeft het type niet opgegeven te worden, maar toch is Haskell een sterk getypeerde taal.

Wat is er "speciaal"

- Haskell kent type inferentie: van de meeste waarden hoeft het type niet opgegeven te worden, maar toch is Haskell een sterk getypeerde taal.
- Haskell kent lazy evaluation: expressies worden pas uitgerekend als ze echt nodig zijn voor de voortgang van de berekening.

Wat is er "speciaal"

- Haskell kent type inferentie: van de meeste waarden hoeft het type niet opgegeven te worden, maar toch is Haskell een sterk getypeerde taal.
- Haskell kent lazy evaluation: expressies worden pas uitgerekend als ze echt nodig zijn voor de voortgang van de berekening.
- Haskell kent een uitgebreid systeem voor overloading: we kunnen verschillende functies toch dezelfde naam geven.

Wat is er "speciaal"

- Haskell kent type inferentie: van de meeste waarden hoeft het type niet opgegeven te worden, maar toch is Haskell een sterk getypeerde taal.
- Haskell kent lazy evaluation: expressies worden pas uitgerekend als ze echt nodig zijn voor de voortgang van de berekening.
- Haskell kent een uitgebreid systeem voor overloading: we kunnen verschillende functies toch dezelfde naam geven.
- via het class-system kan veel standaard-code door de compiler gengenereerd worden

Wat is er "speciaal"

- Haskell kent type inferentie: van de meeste waarden hoeft het type niet opgegeven te worden, maar toch is Haskell een sterk getypeerde taal.
- Haskell kent lazy evaluation: expressies worden pas uitgerekend als ze echt nodig zijn voor de voortgang van de berekening.
- Haskell kent een uitgebreid systeem voor overloading: we kunnen verschillende functies toch dezelfde naam geven.
- via het class-system kan veel standaard-code door de compiler gengenereerd worden
- uitstekende faciliteiten voor het schrijven van parallelle programma's

Wat is er "speciaal"

- Haskell kent type inferentie: van de meeste waarden hoeft het type niet opgegeven te worden, maar toch is Haskell een sterk getypeerde taal.
- Haskell kent lazy evaluation: expressies worden pas uitgerekend als ze echt nodig zijn voor de voortgang van de berekening.
- Haskell kent een uitgebreid systeem voor overloading: we kunnen verschillende functies toch dezelfde naam geven.
- via het class-system kan veel standaard-code door de compiler gengenereerd worden
- uitstekende faciliteiten voor het schrijven van parallelle programma's

Wat is er "speciaal"

- Haskell kent type inferentie: van de meeste waarden hoeft het type niet opgegeven te worden, maar toch is Haskell een sterk getypeerde taal.
- Haskell kent lazy evaluation: expressies worden pas uitgerekend als ze echt nodig zijn voor de voortgang van de berekening.
- Haskell kent een uitgebreid systeem voor overloading: we kunnen verschillende functies toch dezelfde naam geven.
- via het class-system kan veel standaard-code door de compiler gengenereerd worden
- uitstekende faciliteiten voor het schrijven van parallelle programma's

Simon Peyton-Jones: Haskell is the world's greatest imperative language!

Geschiedenis

1960 Lisp, is de eerste functionele programmeertaal; gebaseerd op werk van Haskell Curry en Schönfinkel uit de jaren '30

Geschiedenis

1960 Lisp, is de eerste functionele programmeertaal; gebaseerd op werk van Haskell Curry en Schönfinkel uit de jaren '30

1975 ML, heeft een geavanceerd type systeem en type inferentie

Geschiedenis

1960 Lisp, is de eerste functionele programmeertaal; gebaseerd op werk van Haskell Curry en Schönfinkel uit de jaren '30

1975 ML, heeft een geavanceerd type systeem en type inferentie

1992 begin definitie van Haskell, uitmondend in definitie van Haskell98, die vastgelegd wordt in 2002, en momenteel jaarlijks wordt herzien (de huidige versie is Haskell2010)

Waarom Haskell

- moderne taal, met krachtige bibiliotheken
- zeer geavanceerd type systeem

Waarom Haskell

- moderne taal, met krachtige bibiliotheken
- zeer geavanceerd type systeem
  - polymorfie, functional dependencies, type families, associated data types, generic programming libraries etc.
- snel goeiende verzameling gebruikers
- snel groeiende verzameling bibliotheken (`hackage.haskell.org`)

Domain Specific Embedded Languages

- het liefst willen we elk domein een op dat domein toegesneden specifieke taal aanbieden
- maar elke taal heeft wel types en functies, dus dat betekent een hoop dubbel werk

Domain Specific Embedded Languages

- het liefst willen we elk domein een op dat domein toegesneden specifieke taal aanbieden
- maar elke taal heeft wel types en functies, dus dat betekent een hoop dubbel werk
- dus willen we zo veel mogelijk hergebruiken

Domain Specific Embedded Languages

- het liefst willen we elk domein een op dat domein toegesneden specifieke taal aanbieden
- maar elke taal heeft wel types en functies, dus dat betekent een hoop dubbel werk
- dus willen we zo veel mogelijk hergebruiken
- we willen dus een bestaande taal uitbreiden m.b.v. bibiliotheken
- waarbij we wel nog programma's kunnen *typeren*, *analyseren*, *transformeren* etc

Domain Specific Embedded Languages

- het liefst willen we elk domein een op dat domein toegesneden specifieke taal aanbieden
- maar elke taal heeft wel types en functies, dus dat betekent een hoop dubbel werk
- dus willen we zo veel mogelijk hergebruiken
- we willen dus een bestaande taal uitbreiden m.b.v. bibiliotheken
- waarbij we wel nog programma's kunnen *typeren*, *analyseren*, *transformeren* etc
- Haskell is hier erg geschikt voor, omdat we de betekenis (*semantiek*) van allemaal constructies vast kunnen leggen als functies, en die worden nu juist zo goed ondersteund.

De interpretator ghci

In de Haskell interpretator kunnen we een lijst van commando's opvragen met `:h`:

```
Commands available from the prompt:

  <statement>                 evaluate/run <statement>
  :                           repeat last command
  :\n ..lines.. \n:\n       multiline command
  :browse[!] [[*]<mod>]       display the names defined by module <mod>
                              (!: more details; *: all top-level names)
  :cd <dir>                   change directory to <dir>
  :edit <file>                edit file
  :help, :?                   display this list of commands
  :info [<name> ...]          display information about the given names
  :load [*]<module> ...       load module(s) and their dependents
  :module [+/-] [*]<mod> ...  set the context for expression evaluation
  :quit                       exit GHCi
  :reload                     reload the current module set
  :type <expr>                show the type of <expr>
```

Prelude functies

- De prelude bevat veel functies die op lijsten werken, zoals: In bovenstaand voorbeeld is $[1..10]$ de Haskell-notatie voor de lijst getallen van 1 tot en met 10.

Prelude functies

- De prelude bevat veel functies die op lijsten werken, zoals: In bovenstaand voorbeeld is $[1..10]$ de Haskell-notatie voor de lijst getallen van 1 tot en met 10.
- Behalve dat we lijsten als argument aan functies kunnen geven, kunnen functies ook weer lijsten opleveren:

```
? reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Nog een paar voorbeelden

- De namen van de standaardduncties die lijsten manipuleren spreken vaak voor zich:

```
? length [1,5,9,3]
4
? replicate 10 3
[3,3,3,3,3,3,3,3,3,3]
```

Nog een paar voorbeelden

- De namen van de standaardduncties die lijsten manipuleren spreken vaak voor zich:

```
? length [1,5,9,3]
4
? replicate 10 3
[3,3,3,3,3,3,3,3,3,3]
```

- Merk op dat als een functieaanroep meerdere argumenten heeft daar geen komma's tussen staan.

Nog een paar voorbeelden

- De namen van de standaardduncties die lijsten manipuleren spreken vaak voor zich:

```
? length [1,5,9,3]
4
? replicate 10 3
[3,3,3,3,3,3,3,3,3,3]
```

- Merk op dat als een functieaanroep meerdere argumenten heeft daar geen komma's tussen staan.
- In één expressie kunnen meerdere functieaanroepen voor komen:

```
? reverse (replicate 5 2)
[2,2,2,2,2]
```

Ons eerste programma

De faculteit van een getal n (vaak genoteerd als $n!$) is het product van de getallen van 1 tot en met n, bijvoorbeeld $4! = 1 * 2 * 3 * 4 = 24$:

$$fac\ n = product\ [1\,..\,n]$$

Draaien van programma

Voordat de nieuwe functie kan worden gebruikt, moet de interpreter weten dat de nieuwe file functiedefinities bevat. Dat kan hem meegedeeld worden met het commando `:l` (afkorting van 'load') dus:

```
? :l MijnEersteHaskellModule.hs
```

Draaien van programma

Voordat de nieuwe functie kan worden gebruikt, moet de interpreter weten dat de nieuwe file functiedefinities bevat. Dat kan hem meegedeeld worden met het commando `:l` (afkorting van 'load') dus:

```
? :l MijnEersteHaskellModule.hs
```

Daarna kan de nieuwe functie gebruikt worden:

```
? fac 6
720
```

Syntaxdiagrammen

In de appendix en op de website.

Syntaxdiagrammen

In de appendix en op de website.

Studeeraanwijzing

Ga regelmatig na welk stuk van de taal U inmiddels kent. Vink dit bijvoorbeeld aan in de diagrammen. Als het goed is staan aan het eind van het vak vrijwel overal vinkjes!

Gebruiken van functies

De functie 'n boven k': het aantal manieren waarop k objecten uit een verzameling van n gekozen kunnen worden.

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

Gebruiken van functies

De functie 'n boven k': het aantal manieren waarop k objecten uit een verzameling van n gekozen kunnen worden.

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

In Haskell:

$$boven\ n\ k = fac\ n\ `div`\ (fac\ k * fac\ (n-k))$$

Gebruiken van functies

De functie 'n boven k': het aantal manieren waarop k objecten uit een verzameling van n gekozen kunnen worden.

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

In Haskell:

$$boven\ n\ k = fac\ n\ `div`\ (fac\ k * fac\ (n-k))$$

Er kan nu bepaald worden op hoeveel manieren uit tien mensen een commissie van drie samengesteld kan worden:

```
? boven 10 3
120
```

Nog een keer ..

In Haskell kunnen we zelf operatoren te definiëren. De functie *boven* had misschien beter als operator gedefinieerd kunnen worden:

$$n\ !\hat{}!\ k = fac\ n\ `div`\ (fac\ k * fac\ (n - k))$$

Namen

In de functiedefinitie

$$fac\ n = product\ [1\,..\,n]$$

is *fac* de naam van een functie die gedefinieerd wordt, en n de naam van zijn parameter.

Namen

In de functiedefinitie

$$fac\ n = product\ [1\,..\,n]$$

is *fac* de naam van een functie die gedefinieerd wordt, en n de naam van zijn parameter.

- Namen van functies en parameters moeten met een kleine letter beginnen.
- Daarna mogen nog meer letters volgen (zowel kleine letters als hoofdletters),
- maar ook cijfers,
- het apostrof-teken $'$ en het onderstrepings-teken $\_$.

Namen

In de functiedefinitie

$$fac\ n = product\ [1\,..\,n]$$

is *fac* de naam van een functie die gedefinieerd wordt, en n de naam van zijn parameter.

- Namen van functies en parameters moeten met een kleine letter beginnen.
- Daarna mogen nog meer letters volgen (zowel kleine letters als hoofdletters),
- maar ook cijfers,
- het apostrof-teken $'$ en het onderstrepings-teken _.

Een paar voorbeelden van mogelijke namen zijn:

$$f\ sum\ x3\ g'\ tot\_de\_macht\ langeNaam$$

Gereserveerde woorden

Er zijn 22 namen die niet voor functies of variabelen gebruikt mogen worden; ze hebben een speciale betekenis:

case of
if then else
let in
where
do
data type newtype deriving
class instance
infix infixl infixr
module import
default
_

We zullen ze in dit vak bijna allemaal tegen komen.

Operatoren

Operatoren bestaan uit één of meer symbolen. Een operator kan uit één symbool bestaan (bijvoorbeeld $+$, maar ook uit twee $\wedge$of meer symbolen. De symbolen waaruit een operator opgebouwd kan worden zijn:

```
:  #  $  %  &  *  +  -  =
.  /  \  <  >  ?  !  @  ^  |
```

Toegestane operatoren in programmatekst zijn bijvoorbeeld:

```
+   *.   ++   &&   ||   <=   ==   /=   .   $   //
?   @@   -*-  \/   /\   ...  <+>  :->
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Notatie op slides en in dictaat

In het diktaat gebruiken we in de geformatteerde programma's hiervoor soms wiskundige symbolen, en zien de operatoren &&, ||, <=, == en =/ er uit als $\wedge$, $\vee$, $\leqslant$, $\equiv$ en $\not\equiv$.

Basistypes

Je kunt in Haskell ook met gehele getallen groter dan 2 miljard rekenen. Je moet dan echter aangeven dat het type van de getallen *Integer* is, in plaats van het gebruikelijke *Int*.

```
? 123456789 * 123456789 :: Integer
15241578750190521
```

Booleans

Waarheidswaarden kunnen gecombineerd worden met de operatoren $\wedge$ ('en', &&) en $\vee$ ('of', ||). De operator $\wedge$ geeft alleen *True* als resultaat als links èn rechts een ware uitspraak staat:

```
? 1<2 && 3<4
True
? 1<2 && 3>4
False
```

Booleans

Waarheidswaarden kunnen gecombineerd worden met de operatoren $\wedge$ ('en', &&) en $\vee$ ('of', ||). De operator $\wedge$ geeft alleen *True* als resultaat als links èn rechts een ware uitspraak staat:

```
? 1<2 && 3<4
True
? 1<2 && 3>4
False
```

Voor de 'of'-operator hoeft maar één van de twee uitspraken waar te zijn (maar allebei mag ook):

```
? 1==1 || 2==3
True
? not True
False
```

Lijsten

De operator : zet een extra element op kop van een lijst. De operator ++ plakt twee lijsten aan elkaar:

```
? 1 : [2,3,4]
[1, 2, 3, 4]
? [1,2] ++ [3,4,5]
[1, 2, 3, 4, 5]
```

Lijsten

De operator : zet een extra element op kop van een lijst. De operator ++ plakt twee lijsten aan elkaar:

```
? 1 : [2,3,4]
[1, 2, 3, 4]
? [1,2] ++ [3,4,5]
[1, 2, 3, 4, 5]
```

De functie *null* kijkt of een lijst leeg is (geen elementen bevat) en de functie *and* gaat na of alle elementen van de lijst *True* zijn:

```
? null [ ]
True
? and  [ 1<2, 2<3, 1==0 ]
False
```

Functies als argument

Een argument van een functie kan echter zelf ook een functie zijn!

De functie *map* past de zijn eerste argument (wat een functie moet zijn) toe op alle elementen van zijn tweede argument, wat een lijst moet zijn. Bijvoorbeeld:

```
? map fac [1,2,3,4,5]
[1, 2, 6, 24, 120]
? map sqrt [1.0,2.0,3.0,4.0]
[1.0, 1.41421, 1.73205, 2.0]
? map even [1..8]
[False, True, False, True, False,
 True, False, True]
```

Nieuwe functies uit oude ...

De eenvoudigste manier om functies te definiëren is door een aantal andere functies, bijvoorbeeld standaardfuncties uit de prelude, te combineren:

$$
\begin{array}{lll}
fac & n & = product\ [1\,..\,n] \\
oneven & x & = \neg\ (even\ x) \\
kwadraat & x & = x * x \\
som\_van\_kwadraten & lijst & = sum\ (map\ kwadraat\ lijst)
\end{array}
$$

Nieuwe functies uit oude ...

De eenvoudigste manier om functies te definiëren is door een aantal andere functies, bijvoorbeeld standaardfuncties uit de prelude, te combineren:

$$
\begin{array}{lll}
fac & n & = product\ [1\,..\,n] \\
oneven & x & = \neg\ (even\ x) \\
kwadraat & x & = x * x \\
som\_van\_kwadraten & lijst & = sum\ (map\ kwadraat\ lijst)
\end{array}
$$

Functies kunnen ook meer dan één parameter krijgen:

$$
\begin{array}{lll}
boven & n\ k & = fac\ n\ `div`\ (fac\ k * fac\ (n-k)) \\
abcFormule & a\ b\ c & = [(-b + sqrt\ (b*b - 4.0*a*c))\ /\ (2.0*a) \\
 & & \ ,(-b - sqrt\ (b*b - 4.0*a*c))\ /\ (2.0*a) \\
 & & \ ]
\end{array}
$$

Algemene vorm functiedefinitie

Elke functie-definitie heeft (vooreerst) de volgende vorm:

- de naam van de functie
- de namen van eventuele parameters
- een $=$-teken
- een expressie waar de parameters, standaardfuncties, notaties voor waarden en zelf-gedefinieerde functies in mogen voorkomen.

= is niet ==

Bij een functie met als resultaat een Boolese waarde staat rechts van het $=$-teken een expressie met een Boolese waarde:

$$
\begin{aligned}
&\mathit{negatief}\ x = x < 0 \\
&\mathit{positief}\ x\ = x > 0 \\
&\mathit{isnul}\ x\quad\ = x \equiv 0
\end{aligned}
$$

Let in de laatste definitie op het verschil tussen de $=$ en de $\equiv$. Een enkel is-teken $(=)$ scheidt in functiedefinities de linkerkant van de rechterkant. Een dubbel is-teken $\equiv$ (in de code geschreven als (==)) is een operator, net zoals $<$ en $>$.

Where ...

In de definitie van de functie *abcFormule* komen de expressies $sqrt\ (b * b - 4.0 * a * c)$ en $(2.0 * a)$ twee keer voor. Dus beter:

$$
\begin{aligned}
abcFormule'\ a\ b\ c = [&(-b + x)\ /\ n \\
,&(-b - x)\ /\ n \\
]& \\
\textbf{where}\ &x = sqrt\ (b * b - 4.0 * a * c) \\
&n = 2.0 * a
\end{aligned}
$$

Gevalsonderscheid in Functiedefinities

Soms is het nodig om in de definitie van een functie meerdere gevallen te onderscheiden Dat gebeurt bijvoorbeeld in de definitie van de functie *signum*:

$$\begin{array}{lrcr} \mathit{signum}\ x & \mid x > 0 & = & 1 \\ & \mid x \equiv 0 & = & 0 \\ & \mid x < 0 & = & -1 \end{array}$$

De definities voor de verschillende gevallen worden ‘bewaakt’ door Boolese expressies, die dan ook guards worden genoemd.

Functieapplicatie

- functieapplicatie associeert naar links

Functieapplicatie

- functieapplicatie associeert naar links
- eigenlijk neemt elke functie precies een argument

Functieapplicatie

- functieapplicatie associeert naar links
- eigenlijk neemt elke functie precies een argument
- functie applicatie bindt sterker dan alle operatoren

Functieapplicatie

- functieapplicatie associeert naar links
- eigenlijk neemt elke functie precies een argument
- functie applicatie bindt sterker dan alle operatoren

Functieapplicatie

- functieapplicatie associeert naar links
- eigenlijk neemt elke functie precies een argument
- functie applicatie bindt sterker dan alle operatoren

Dus vergeet dit nooit:

$$f\ x\ y\ z - 1 + g\ 4 \Rightarrow (((f\ x)\ y)\ z) - 1 + (g\ 4)$$

Patronen

Je kunt een functie definiëren met verschillende patronen als formele parameter:

$$
\begin{aligned}
som\ [\,] &= 0 \\
som\ [x] &= x \\
som\ [x,y] &= x+y \\
som\ [x,y,z] &= x+y+z
\end{aligned}
$$

Patronen

Je kunt een functie definiëren met verschillende patronen als formele parameter:

$$
\begin{aligned}
som\ [\,] &= 0 \\
som\ [x] &= x \\
som\ [x,y] &= x+y \\
som\ [x,y,z] &= x+y+z
\end{aligned}
$$

Bij aanroep van de functie kijkt de interpreter of de parameter 'past' op een van de definities; de aanroep $som\ [3,4]$past bijvoorbeeld op de derde regel van de definitie.

Soorten van patronen

O.a. de volgende constructies zijn toegestaan als patroon:

- Getallen (bijvoorbeeld 3)
- De constanten *True* en *False*;
- Namen (bijvoorbeeld x);
- Lijsten, waarvan de elementen ook weer patronen zijn (bijvoorbeeld $[1, x, y]$);
- De operator : met patronen links en rechts (bijvoorbeeld $a : b$);

Voorbeelden uit de Prelude

Met behulp van patronen zijn een aantal belangrijke functies te definiëren:

$$\begin{aligned} \mathit{True} \wedge \mathit{True} &= \mathit{True} \\ \_ \quad \wedge \_ \quad &= \mathit{False} \end{aligned}$$

Voorbeelden uit de Prelude

Met behulp van patronen zijn een aantal belangrijke functies te definiëren:

$$\begin{aligned} \mathit{True} \wedge \mathit{True} &= \mathit{True} \\ \_ \quad \wedge \_ \quad &= \mathit{False} \end{aligned}$$

Met de operator : kunnen lijsten worden opgebouwd. Hiermee kunnen twee nuttige standaardfuncties geschreven worden:

$$\begin{aligned} \mathit{head}\ (x:y) &= x \\ \mathit{tail}\ \ (x:y) &= y \end{aligned}$$

Recursie, en daar gaat het om ...

Recursieve functies (d.w.z. functies die zichzelf aanroepen, zijn echter zinvol onder de volgende twee voorwaarden:

- de parameter van de recursieve aanroep is *eenvoudiger* (bijvoorbeeld: numeriek kleiner, of een kortere lijst) dan de parameter van de te definiëren functie;
- voor een *basis-geval* is er een niet-recursieve definitie.

Recursie, en daar gaat het om ...

Recursieve functies (d.w.z. functies die zichzelf aanroepen, zijn echter zinvol onder de volgende twee voorwaarden:

- de parameter van de recursieve aanroep is *eenvoudiger* (bijvoorbeeld: numeriek kleiner, of een kortere lijst) dan de parameter van de te definiëren functie;
- voor een *basis-geval* is er een niet-recursieve definitie.

Recursieve definities van de faculteit-functie zijn:

$$\begin{array}{l} fac\ n \mid n \equiv 0 = 1 \\ \qquad\quad \mid n > 0 = n * fac\ (n-1) \end{array}$$

$$\begin{array}{l} fac\ 0 = 1 \\ fac\ n = n * fac\ (n-1) \end{array}$$

Recursie gaat goed met lijsten samen ..

Vrijwel alle functies of lijsten volgen een speciaal patroon:

$$\begin{aligned} som\ [\,] &= 0 \\ som\ (kop : staart) &= kop + som\ staart \end{aligned}$$

Recursie gaat goed met lijsten samen ..

Vrijwel alle functies of lijsten volgen een speciaal patroon:

$$\begin{aligned} som\ [\,] &= 0 \\ som\ (kop : staart) &= kop + som\ staart \end{aligned}$$

Tip: als je zelf zo'n functie schrijft, begin dan altijd met het lege geval, controleer het type, en kijk of dat je aanstaat.