

Generics via Lisp-Like Primitives

go fold-free

Matt Roberts



Datatype Generic is Nice

Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming

Ralf Lämmel
Vrije Universiteit, Amsterdam

Simon Peyton Jones
Microsoft Research, Cambridge

```
salaryupdate.hs
1  {- (almost) verbatim from \cite{Laemmel03} -}
2  {-# LANGUAGE DeriveDataTypeable #-}
3
4  import Data.Generics
5
6  data Company = C [Dept]           deriving (Show, Data, Typeable)
7  data Dept   = D Name Manager [SubUnit] deriving (Show, Data, Typeable)
8  data SubUnit = PU Employee | DU Dept deriving (Show, Data, Typeable)
9  data Employee = E Person Salary deriving (Show, Data, Typeable)
10 data Person = P Name Address deriving (Show, Data, Typeable)
11 data Salary = S Int deriving (Show, Data, Typeable)
12 type Manager = Employee
13 type Name = String
14 type Address = String
15
16 increase :: Int -> Company -> Company
17 increase k = everywhere (mkT (incS k))
18
19 incS :: Int -> Salary -> Salary
20 incS k (S s) = S (s + k)
21
22 genCom :: Company
23 genCom = C [ D "Research" ralf [PU joost, PU marlow]
24           , D "Strategy" blair []
25           ]
26
27 ralf, joost, marlow, blair :: Employee
28 ralf = E (P "Ralf" "Amsterdam") (S 8000)
29 joost = E (P "Joost" "Amsterdam") (S 1000)
30 marlow = E (P "Marlow" "Cambridge") (S 2000)
31 blair = E (P "Blair" "London") (S 100000)
Line: 1 Column: 13 Haskell Soft Tabs: 2
```

```
everywhere.hs
1 everywhere :: Term a => (forall b.Term b => b -> b) -> a -> a
2 everywhere f x = f (gmapT (everywhere f) x)
Line: 1 Column: 1 Haskell Soft Tabs: 2 everyw...
```

Explicit Spine View is Nicer

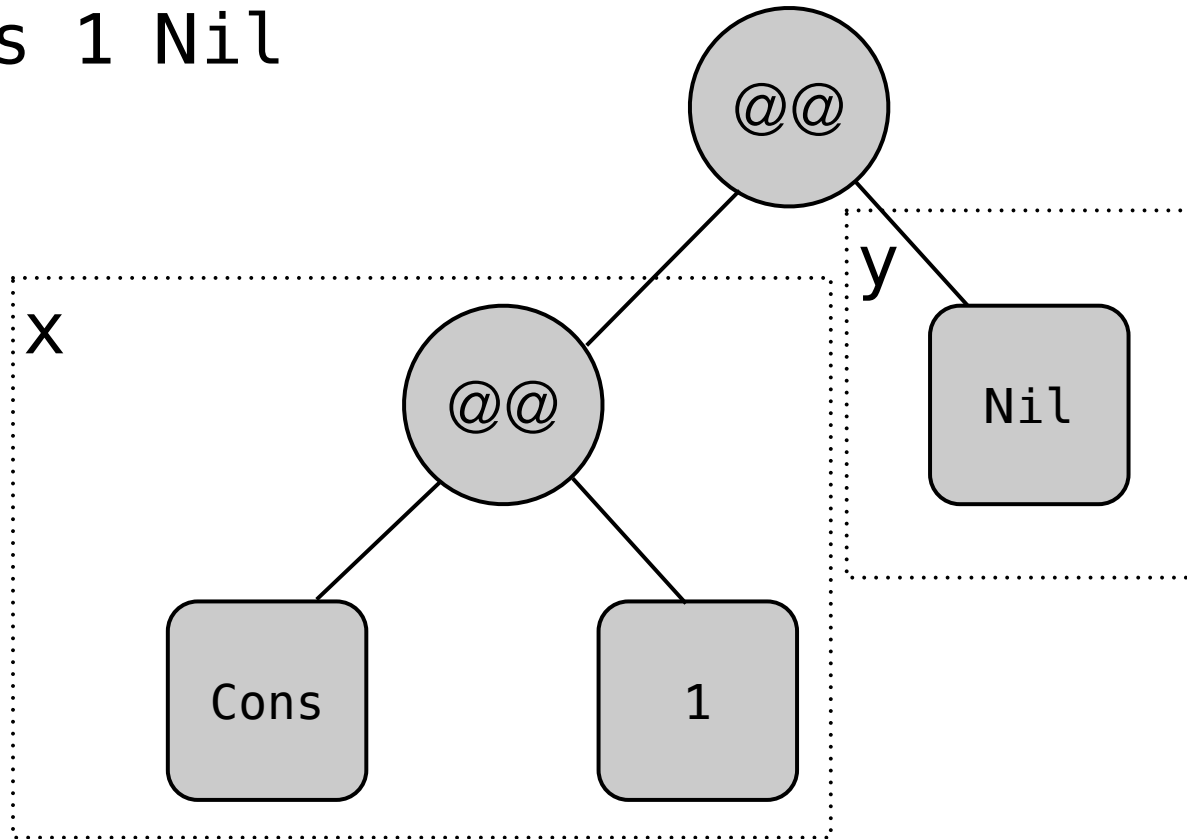
```
salaryupdate.hs
1  {- (almost) verbatim from \cite{Laemmel03} -}
2  {-# LANGUAGE DeriveDataTypeable #-}
3
4  import Data.Generics
5
6  data Company = C [Dept]           deriving (Show, Data, Typeable)
7  data Dept   = D Name Manager [SubUnit] deriving (Show, Data, Typeable)
8  data SubUnit = PU Employee | DU Dept deriving (Show, Data, Typeable)
9  data Employee = E Person Salary deriving (Show, Data, Typeable)
10 data Person = P Name Address deriving (Show, Data, Typeable)
11 data Salary = S Int deriving (Show, Data, Typeable)
12 type Manager = Employee
13 type Name = String
14 type Address = String
15
16 increase :: Int -> Company -> Company
17 increase k = everywhere (mkt (incS k))
18
19 incS :: Int -> Salary -> Salary
20 incS k (S s) = S (s + k)
21
22 genCom :: Company
23 genCom = C [ D "Research" ralf [PU joost, PU marlow]
24           , D "Strategy" blair []
25           ]
26
27 ralf, joost, marlow, blair :: Employee
28 ralf = E (P "Ralf" "Amsterdam") (S 8000)
29 joost = E (P "Joost" "Amsterdam") (S 1000)
30 marlow = E (P "Marlow" "Cambridge") (S 2000)
31 blair = E (P "Blair" "London") (S 100000)
Line: 1 Column: 13 Haskell Soft Tabs: 2
```

```
everywhere.hs
1  everywhere :: (forall b. b -> b) -> a -> a
2  everywhere f (x y) = (everywhere f x) @@ (everywhere f y)
3  everywhere f x     = f x
Line: 3 Column: 20 Haskell Sof
```

this is not the right everywhere - it needs an f at the front of the recursive call

Spine view (tuples of atoms)

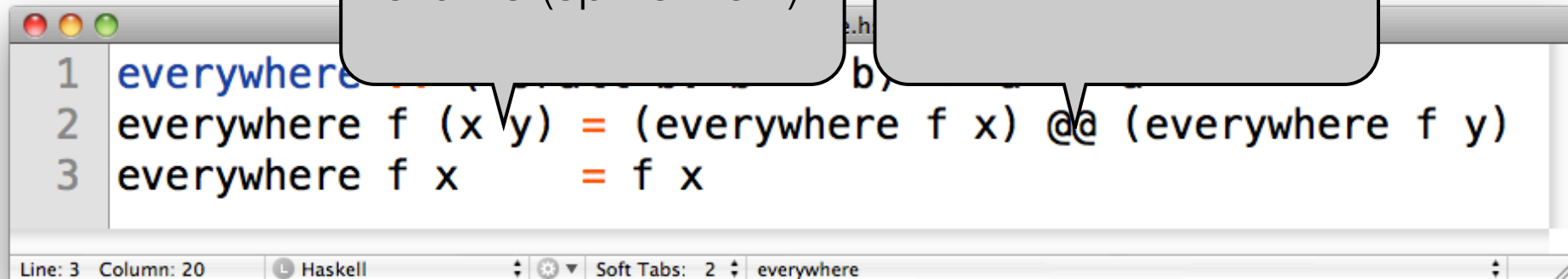
Cons 1 Nil



But this does not exist

All data is a tuple of atoms (spine view)

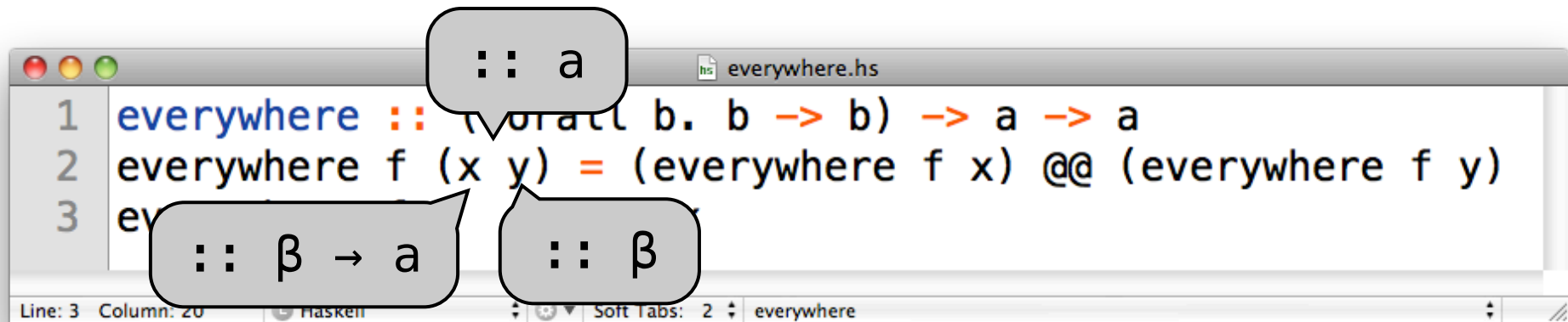
Data application



```
1 everywhere f x y = (everywhere f x) @@ (everywhere f y)
2 everywhere f (x y) = (everywhere f x) @@ (everywhere f y)
3 everywhere f x = f x
```

Line: 3 Column: 20 Haskell Soft Tabs: 2 everywhere

Types!



```
1 everywhere :: forall b. b -> b) -> a -> a
2 everywhere f (x y) = (everywhere f x) @@ (everywhere f y)
3 ev
```

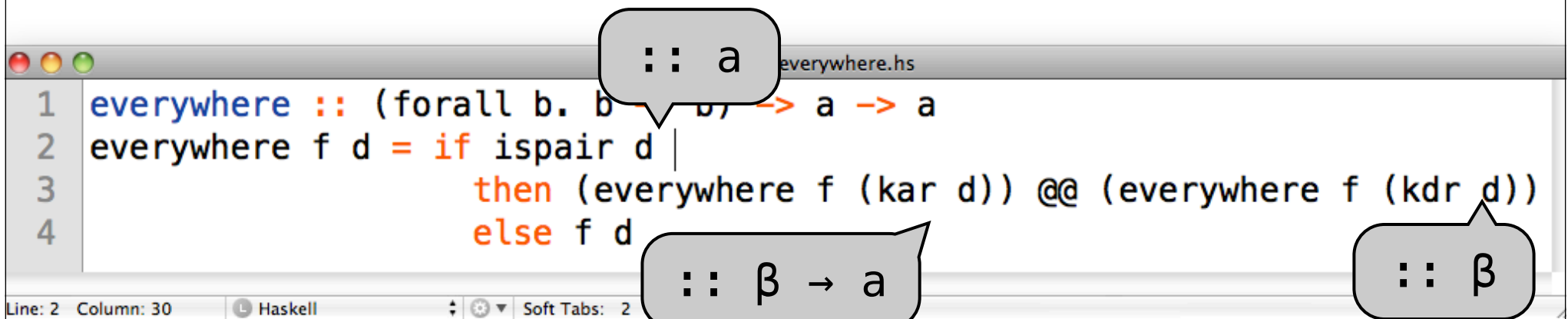
Callouts:

- `:: a` (pointing to the type of the lambda function)
- `:: $\beta \rightarrow a$` (pointing to the type of `f`)
- `:: β` (pointing to the type of `x` and `y`)

Editor status: Line: 3 Column: 20 Haskell Soft Tabs: 2 everywhere

The β s must be the same, but there is no link between them

Compiled version



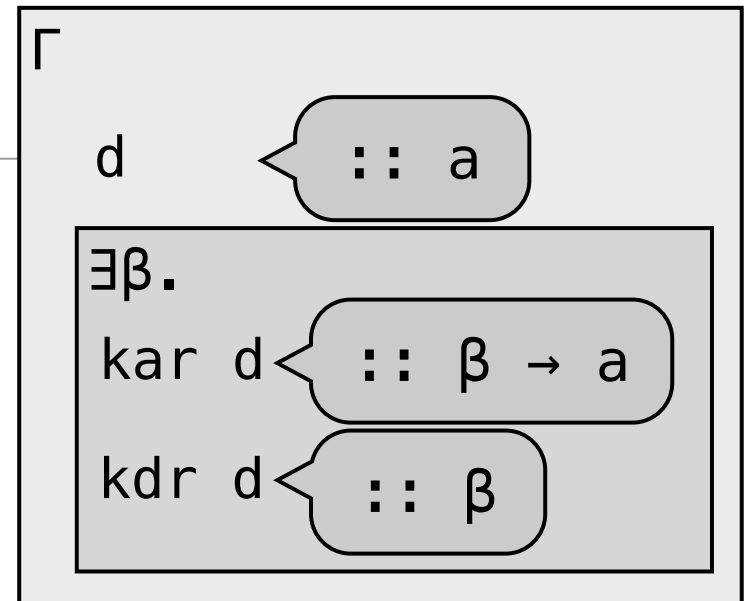
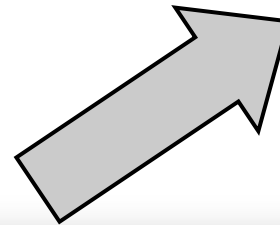
```
1 everywhere :: (forall b. b -> b) -> a -> a
2 everywhere f d = if ispair d |
3     then (everywhere f (kar d)) @@ (everywhere f (kdr d))
4     else f d
```

The screenshot shows a Haskell code editor window titled 'everywhere.hs'. The code defines the 'everywhere' function. Three callout boxes highlight type annotations: one above the function signature pointing to 'a', one below the function body pointing to the lambda expression '(forall b. b -> b) -> a -> a', and one to the right of the function body pointing to the type variable 'β'.

The β s must be the same, we link them at the ispair

Type algorithm

every
kar, kdr
witnessed by
an ispair



```
compiled_everywhere.hs
1 everywhere :: (forall b. b -> b) -> a -> a
2 everywhere f d = if ispair d |
3                   then (everywhere f (kar d)) @@ (everywhere f (kdr d))
4                   else f d
```

Line: 2 Column: 30 Haskell Soft Tabs: 2 everywhere

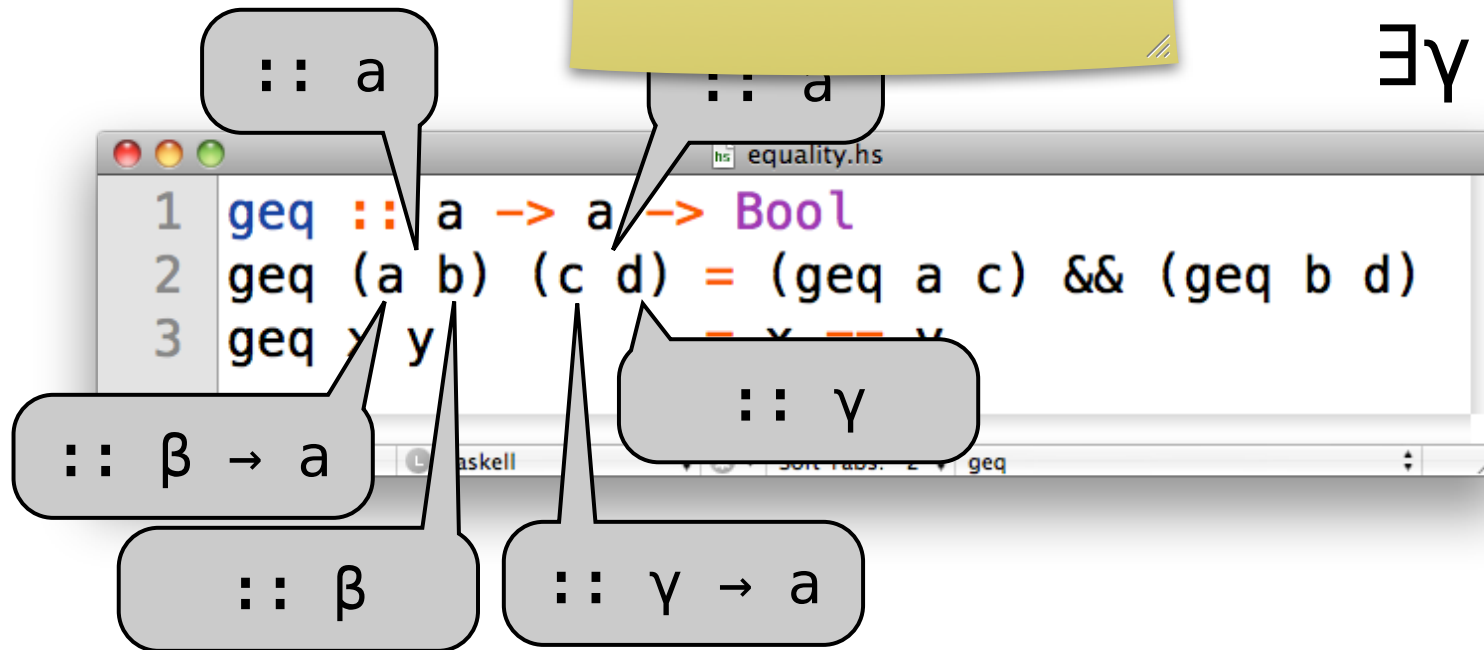
the triple, ispair, kar and kdr are acting like a single function (a fold) for the type system.

Generic Equality

This slide is incorrect and should be ignored.

*the same but
on is lost*

$\exists \alpha$
 $\exists \gamma$



The image shows a screenshot of a Haskell code editor window titled "equality.hs". The code contains three lines:

```
1 geq :: a -> a -> Bool
2 geq (a b) (c d) = (geq a c) && (geq b d)
3 geq y y = y == y
```

Several callout boxes point to parts of the code:

- A box containing `:: a` points to the type signature of the first line.
- A box containing `:: a` points to the variable `a` in the function body of the first line.
- A box containing `:: $\beta \rightarrow a$` points to the variable `a` in the function body of the second line.
- A box containing `:: β` points to the variable `a` in the function body of the second line.
- A box containing `:: γ` points to the variable `c` in the function body of the second line.
- A box containing `:: $\gamma \rightarrow a$` points to the variable `c` in the function body of the second line.

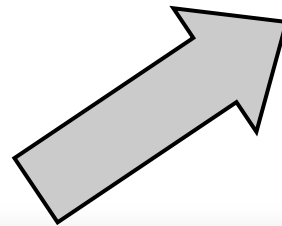
Existentially quantified variables have to be treated as constants - they can only unify with themselves.

Type algorithm

This slide is incorrect and should be ignored.

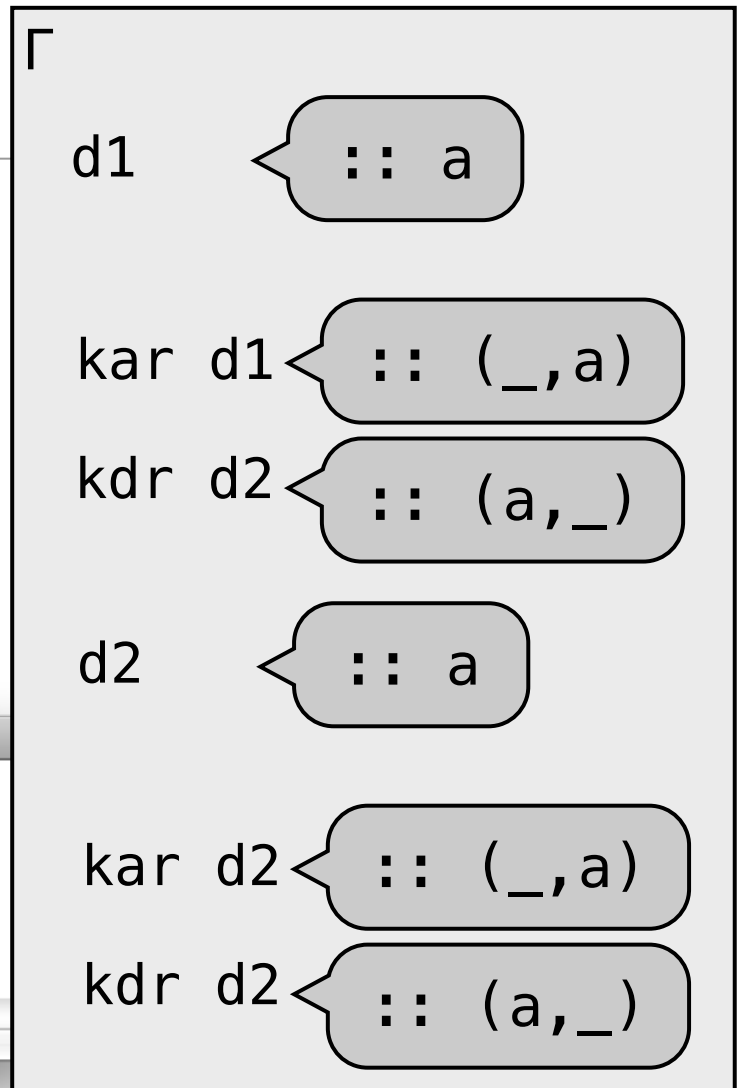
info

he
ously



```
geq d1 -> d2 Bool
geq (a b) (c d) = (geq a c) && (geq b d)
geq x y       = x == y
```

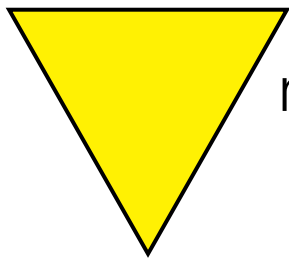
column: 33 Haskell Soft Tabs: 2 geq



the triple, ispair, kar and kdr are acting like a single function (a fold) for the type system.

Opens up a whole world

We don't have time to cover more here, but you will find all these details and more in my upcoming dissertation



warning! you
need to learn a
new syntax

You can play with dgen at
<http://dgen.science.mq.edu.au:8080>