named after Haskell Brooks Curry (1900-1982)
(an American Logician)

Brief History

(FPCA '87) in Portland, Oregon, there was a strong consensus that a committee be formed to define an open standard for lazy functional languages. The committee's purpose was to consolidate existing functional languages into a common one to serve as a basis for future research in functional-language design.

Language Designers

Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, Simon Peyton Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, Philip Wadler

Haskell (The Programming Language)

- General Purpose, Statically Typed, Purely Functional
- The GHC (Glasgow Haskell Compiler) provides both native compilation and an interactive environment GHCi (GHC interactive), which functions as an REPL (Read-Eval-Print-Loop)
- features Type Inference and Lazy Evaluation

Lazy Evaluation

delays the evaluation of an expression until its value is needed

The 3 Main Categories of Types for Haskell

- Primitive Types
- Composite Types
- Algebraic Data Types (ADTs)

(1/3) Primitive Types

Built-in value types that represent singular pieces of data

- `Int` : Fixed-size whole numbers (e.g. 64-bit)
- `Integer` : Arbitrary-precision whole numbers
- `Float` : Single-precision floating-point numbers
- `Double` : Double-precision floating-point numbers
- `Bool` : Boolean values (e.g. `True`, `False`)
- `Char` : A single Unicode character

(2/3) Composite Types

Structures that build new types out of existing ones.

- Lists [] : Homogeneous e.g. [Int], [Double], [Char]
- Tuples () : Heterogeneous e.g. (Integer, Int, String)

Note: String is just a type synonym to [Char]

- Functions -> : Arrow is type constructor for function

Note: All functions are curried by default

E.g. Int -> Int -> Bool

(3/3) Algebraic Data Types (ADTs)

A combination of two basic mathematical concepts.

- Product Type : a value has all of these fields

E.g. `data Point = Point Double Double`

- Sum Type : a value is one of these fields

E.g. `data Direction = North | South | East | West`

Defining and Applying Functions

```
-- takes Int, then another Int, and returns Int

add :: Int -> Int -> Int

add x y = x + y


-- called using spaces, not parentheses.

add 1 2          -- evaluates to 3

add 1 (2+3)      -- evaluates to 6
```

` (Backticks) treats a function as an infix operator

```
add :: Int -> Int -> Int

add x y = x + y


-- prefix (the standard way)

add 1 2

-- infix (using backticks)

1 `add` 2
```

Pattern Matching

```
-- definitions checked sequentially from top to bottom

factorial :: Int -> Int

factorial 0 = 1                          -- Base Case

factorial n = n * factorial (n - 1)   -- Recursive Case
```

Significant Whitespace

```
-- where introduces local bindings (local variables)

calculatePrice :: Double -> Double

calculatePrice x = itemPrice + tax

  where

    itemPrice = x * 0.8                -- 20% Discount

    tax       = itemPrice * 0.07       --  7% VAT
```

$ (Application) and . (Composition)

```
-- $ feeds right side as the argument to the left side
putStrLn $ show (1 + 1)       -- feeds show (1+1) to putStrLn
putStrLn (show (1 + 1))       -- same as above

-- . chains functions together (no application yet)
f . g . h $ 10                -- feeds 10 into chained f.g.h
f (g (h 10))                  -- same as above
```

Data Structure: Lists and Tuples

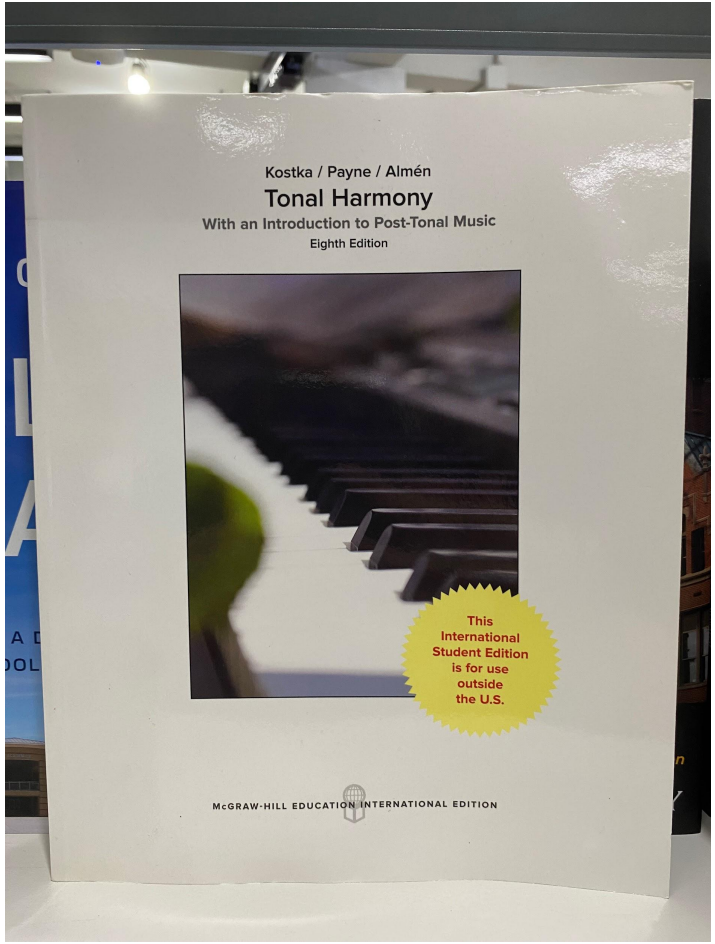
```
-- lists are homogeneous (all must be the same type)
listExample :: [Int]
listExample = [1, 2, 3, 4, 5]

-- tuples are heterogenous (multiple types allowed)
tupleExample :: (Int, Bool, Char, Double)
tupleExample = (1, True, 'a', 5.0)
```

List Comprehension

```
-- Syntax: [ expression | generator, filter(s) ]
[x * x | x <- [1..10], x 'mod' 2 == 0]
-- Evaluates to [4, 16, 36, 64, 100]
[x * x | x <- [1..10], x 'mod' 2 /= 0]
-- Evaluates to [1, 9, 25, 49, 81]
```

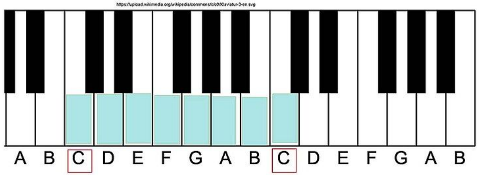
List Comprehension : Infinite List

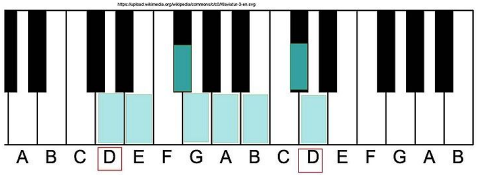
```
-- Syntax: [start,second..end], step is second - start
[2,4..10]  -- Evaluates to [2, 4, 6, 8, 10]
[1..]      -- Evaluates to [1, 2, 3, 4, 5, 6, 7, 8, 9, ..]

take 10 [x | x <- [1..], x 'mod' 7 == 0]
-- Evaluates to [7, 14, 21, 28, 35, 42, 49, 56, 63 ,70]
```

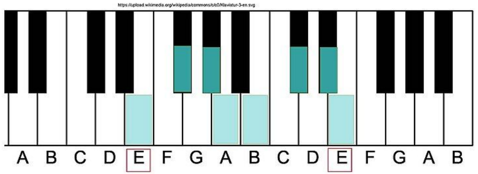

Kostka / Payne / Almén
Tonal Harmony
With an Introduction to Post-Tonal Music
Eighth Edition
This International Student Edition is for use outside the U.S.
McGRAW-HILL EDUCATION INTERNATIONAL EDITION

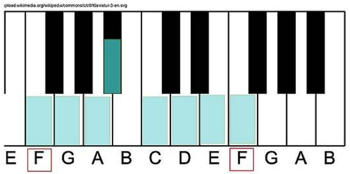
Tonal Music

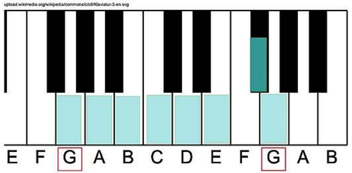
System where pitches and chords are organized hierarchically around a central note (the tonic)

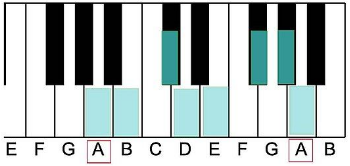

A B C D E F G A B C D E F G A B

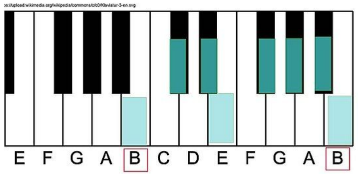

A B C D E F G A B C D E F G A B


A B C D E F G A B C D E F G A B

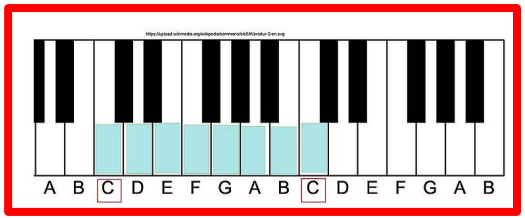

E F G A B C D E F G A B

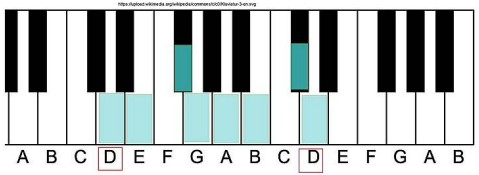

E F G A B C D E F G A B

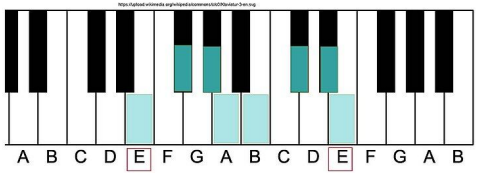

E F G A B C D E F G A B


E F G A B C D E F G A B

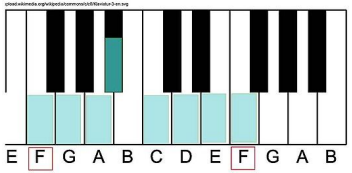
major scale

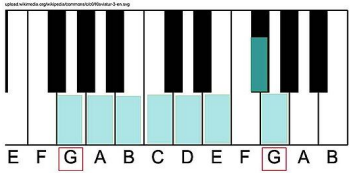

A B C D E F G A B C D E F G A B

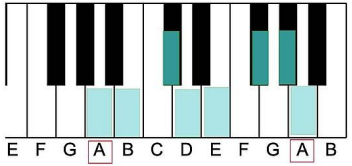

A B C D E F G A B C D E F G A B

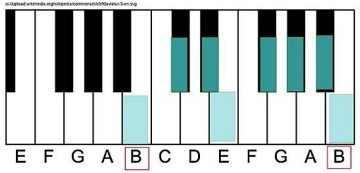

A B C D E F G A B C D E F G A B

major scale

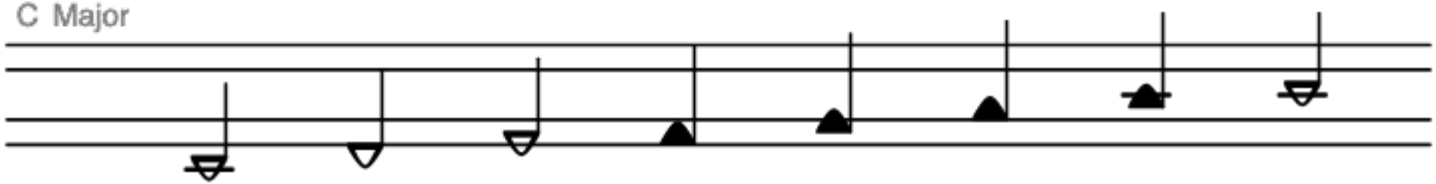

E F G A B C D E F G A B


E F G A B C D E F G A B


E F G A B C D E F G A B

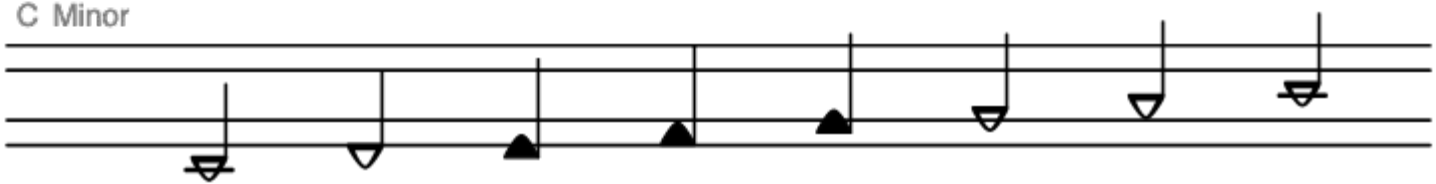

E F G A B C D E F G A B

C Major



Whole Step | Whole Step | Half Step | Whole Step | Whole Step | Whole Step | Half Step

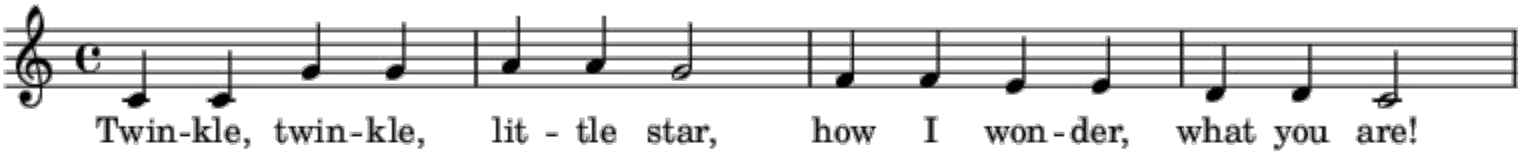
C Minor



Whole Step | Half Step | Whole Step | Whole Step | Half Step | Whole Step | Whole Step

Motif

Twinkle Twinkle Little Star



[1, 1, 5, 5, 6, 6, 5, 4, 4, 3, 3, 2, 2, 1]

```haskell
module Music where

data Note = C | Cs | D | Ds | E | F | Fs | G | Gs | A | As | B
  deriving (Eq, Ord, Enum, Bounded, Show)

data Interval = H | W deriving (Show)

type ScalePattern = [Interval]

majorPattern :: ScalePattern
majorPattern = [W, W, H, W, W, W, H]

minorPattern :: ScalePattern
minorPattern = [W, H, W, W, H, W, W]

```

:

```haskell
intervalToInt :: Interval -> Int
intervalToInt H = 1
intervalToInt W = 2

intToNote :: Int -> Note
intToNote n = toEnum (n `mod` 12)

buildScale :: Note -> ScalePattern -> [Note]
buildScale root pattern =
  init $ map intToNote $ scanl (+) (fromEnum root) (map intervalToInt pattern)

type Motif = [Int]

transposeMotif :: Motif -> [Note] -> [Note]
transposeMotif motif scale =
  map (\degree -> scale !! (degree - 1)) motif

```

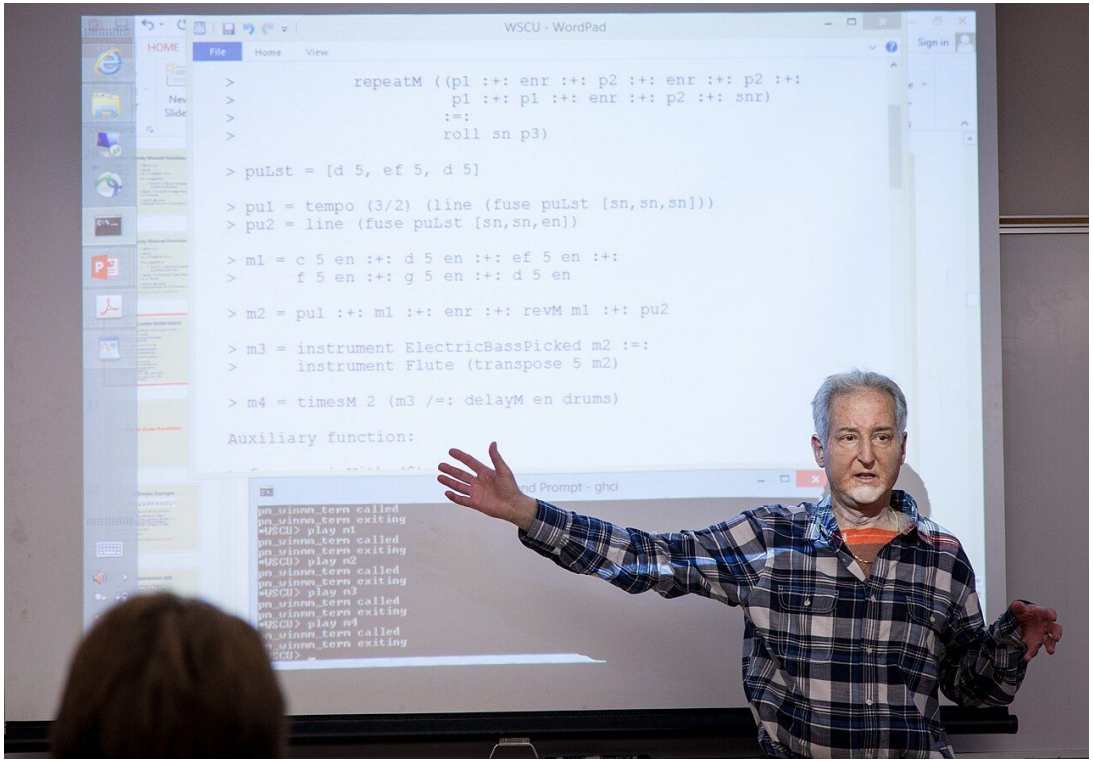
:

```haskell
main :: IO ()
main = do
  let twinkleMotif = [1, 1, 5, 5, 6, 6, 5, 4, 4, 3, 3, 2, 2, 1]

  let cMajorScale = buildScale C majorPattern
  let eMajorScale = buildScale E majorPattern
  let aMinorScale = buildScale A minorPattern

  putStrLn "==> Scales"
  putStr "C Major Scale:   " >> print cMajorScale
  putStr "E Major Scale:   " >> print eMajorScale
  putStr "A Natural Minor: " >> print aMinorScale
  putStrLn ""

  putStrLn "==> Transposed Motif"
  putStr "Twinkle in C Major: " >> print (transposeMotif twinkleMotif cMajorScale)
  putStr "Twinkle in E Major: " >> print (transposeMotif twinkleMotif eMajorScale)
  putStr "Twinkle in A Minor: " >> print (transposeMotif twinkleMotif aMinorScale)
```

:

```
~/HaskellMusic> runghc Music.hs
==> Scales
C Major Scale:    [C,D,E,F,G,A,B]
E Major Scale:    [E,Fs,Gs,A,B,Cs,Ds]
A Natural Minor: [A,B,C,D,E,F,G]

==> Transposed Motif
Twinkle in C Major: [C,C,G,G,A,A,G,F,F,E,E,D,D,C]
Twinkle in E Major: [E,E,B,B,Cs,Cs,B,A,A,Gs,Gs,Fs,Fs,E]
Twinkle in A Minor: [A,A,E,E,F,F,E,D,D,C,C,B,B,A]
~/HaskellMusic>
```

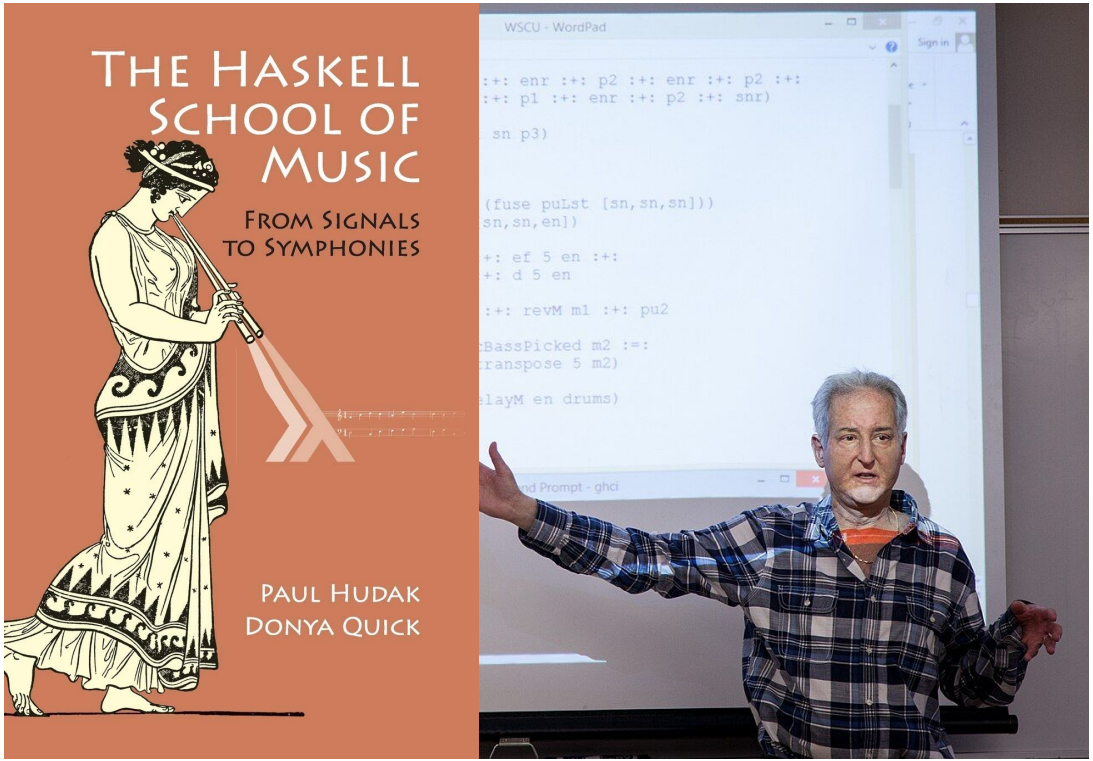
Euterpea



Paul Hudak (1952-2015) in 2013

Euterpea

Yale University



Paul Hudak (1952-2015) in 2013

Brief History

(FPCA '87) in Portland, Oregon, there was a strong consensus that a committee be formed to define an open standard for lazy functional languages. The committee's purpose was to consolidate existing functional languages into a common one to serve as a basis for future research in functional-language design.

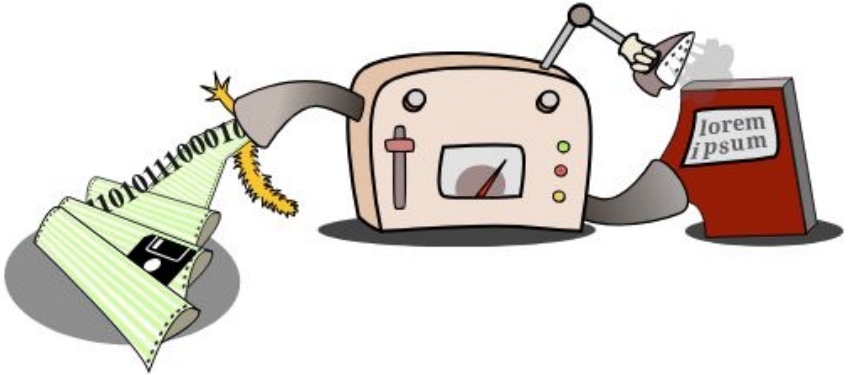
Language Designers

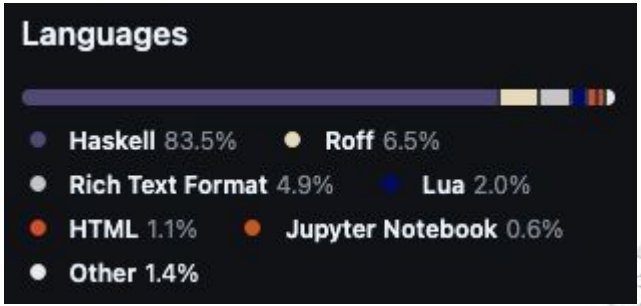
Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, Simon Peyton Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, Philip Wadler

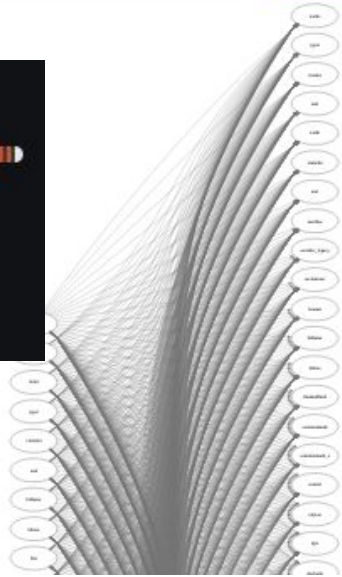
Pandoc a universal document converter

Donate Sponsor Please

About Installing Demos ▼ Documentation Help Extras Releases Search







If you need to convert files from one markup format into another, pandoc is your swiss-army knife. Pandoc can convert between the following formats:

(← = conversion from; → = conversion to; ↔ = conversion from and to)

Haskell in Industry

Many companies have used Haskell for a range of projects, including:

- ABN AMRO Amsterdam, The Netherlands

> ABN AMRO is an international bank headquartered in Amsterdam. For its investment banking activities it needs to measure the counterparty risk
>
> on portfolios of financial derivatives.

ABN AMRO's CUFP talk.

- AT&T

> Haskell is being used in the Network Security division to automate processing of internet abuse complaints. Haskell has allowed us to easily meet very tight deadlines with reliable results.

- Aetion Technologies LLC, Columbus, Ohio

> Aetion was a defense contractor in operation from 1999 to 2011, whose applications use artificial intelligence. Rapidly changing priorities make it important to minimize the code impact of changes, which suits Haskell well. Aetion developed three main projects in Haskell, all successful. Haskell's concise code was perhaps most important for rewriting: it made it practicable to throw away old code occasionally. DSELs allowed the AI to be specified very declaratively.

Aetion's CUFP talk.

“Haskell’s concise code was perhaps most important for rewriting”

is now my Favorite Language
(I’m not good at it, but I still like it.)