IT UNIVERSITY OF COPENHAGEN

Functional Programming

Jesper Bengtson

Patrick Bahr

Who are we?

Patrick Bahr

- At ITU since 2015
- Member of the Programming, Logics and Semantics research group
- Research
  - functional programming languages
  - type theory & formal verification
  - compilers

Who are we?

Jesper Bengtson

- Associate professor at ITU since 2013
- Member of the Programming, Logics and Semantics research group
- Researches formal verification of software
  - Java
  - Distributed systems
  - Message-passing languages

Who are we?

Teaching assistants

- Alexander Berg — email: alebe@itu.dk
- Anders Aversen — email: aarv@itu.dk
- Mathias Scholz — email: matsc@itu.dk
- Line Mei Tong Mao Dejgaard — email: ldej@itu.dk
- Johannes Porsgaard — email: jpor@itu.dk
- Omid Sabihi Marfavi — email: omma@itu.dk

Lectures and exercises

- **Assignment for the coming week**: Tuesdays 08:00
- **Lectures**: Wednesdays, 08:15-10:00, Aud1
- **Exercise sessions** (2A12-14 and 3A12-14)
  - Wednesday: 10:15-12:00
- **Hand-ins**: Tuesdays by 08:00

Textbook

Functional Programming using F#

- Written by Michael R. Hansen and Hans Rischel
- ISBN: 9781107019027
- http://www2.imm.dtu.dk/~mire/FSharpBook/
- Read the relevant chapters in advance of lectures!

Functional programming

Functional Programming Paradigm

- First class functions
- Higher-order functions
- Type inference and polymorphism
- Recursion and tail recursion
- Algebraic data types
- Strict and lazy evaluation

Functional programming

Memory management

- Garbage collection
- Reference types
- Mutable vs immutable data

Parallel programming

- Divide and conquer

Using F#

- F# is an open-source functional language integrated in the Visual Studio development platform.
- Access to all of the .NET libraries
- Available for Windows, Linux, and MacOS (https://fsharp.org/)

Mandatory activities

- 7 weekly **individual** assignments worth 2 points
- 1 one-month long **group** project worth 6 points (bigger assignment, no report)
- You must get 16 points to be eligible to attend the exam
- If you are eligible for the exam from a previous year, you are still eligible for this exam.

Colour coded

- Assignments are colour coded
  - green Basic exercises. Recommended if you're aiming to just pass the course. ← *These are mandatory!*
  - yellow Intermediate exercises. Recommended if you are aiming for a decent grade on the course
  - red Advanced exercises. Recommended if you are aiming for a high grade on the course.
- We highly recommend going for the yellow or red assignment.

Individual weekly assignments

- Due Tuesdays at 08:00
- Graded the same day(0-2 points)
- You can hand in the week after with the next assignment to increase your points
- After that your assignment will not be graded. **Respect the cut-off date.**
- Assignments will be handed in using CodeGrade (new for this year).
- **Work individually**

Individual weekly assignments

- Due Tuesdays at 08:00
- Graded the same day(0-2 points)
- Y… …h the n… …oints
- A… …be g… …
- Assignments will be handed in using CodeGrade (new for this year).
- **Work individually**

Some deadlines are pushed a week to not collide with the hand-ins with the second-year project

Individual weekly assignments

- Due Tuesdays at 08:00

You have two attempts for each assignment. As long as you have made serious work on one of these attempts (handing in something that nearly works) then you will be granted an extension to resubmit towards the end of the course

If you do not hand in, or hand in something that is barely working you will not get this resubmission attempt

The TAs will let you know immediately if you have been granted an extension or not

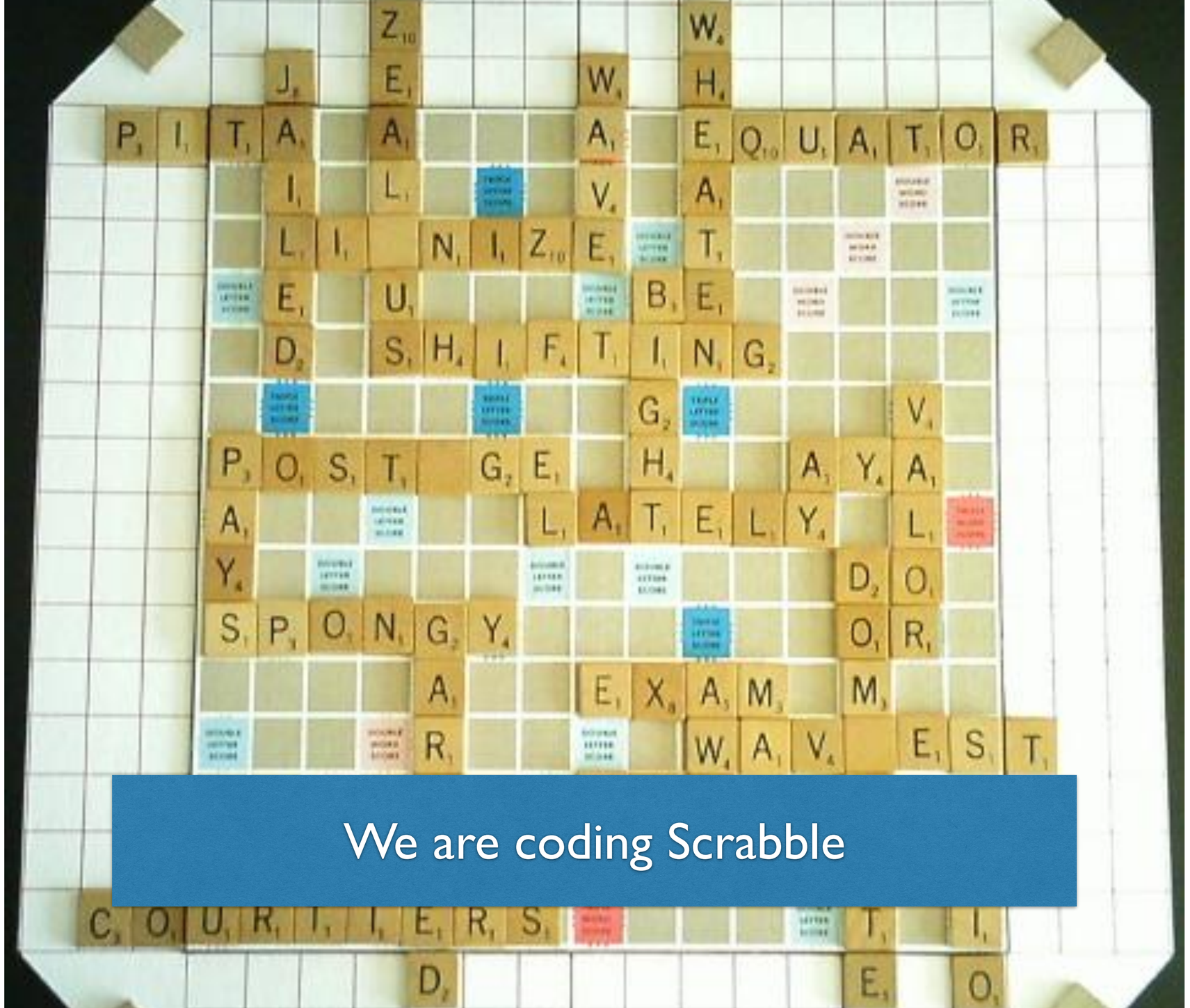
- **Work individually**

IT UNIVERSITY OF COPENHAGEN

Week	Date	Month	Second Year Project	Functional Programming
5	29.-2.	February	Feb 3: Group formation event	
6	5.-9.		Feb 8: Team formation event	Assignment 1 hand-in
7	12.-16.		Project kickoff	Assignment 2 hand-in
8	19.-23.			Assignment 3 hand-in
9	26.-1.	March		Assignment 4 hand-in
10	4.-8.			
11	11-15.		Submission for Check in session 1	Assignment 5 hand-in
12	18.-22.		Check in session 1	Assignment 6 hand-in
13	25.-29.		**Easter Holiday**	
14	1.-5.	April		Assignment 7 hand-in Project kick-off
15	8.-12.			
16	15.-19.		Submission for Check in session 2	
17	22.-26.		Check in session 2	
18	29.-3.	May		
19	6.-9.			Project submission
20	13.-17.		Exam submission	
21	20.-24.			
22	27.-31.		May 30: Celebration event	May 29: Exam
23	3.-7.	June	June 3, 4, 6, 7: Exam	
24	10.-14.		June 10, 14: Exam	

Week	Date	Month	Second Year Project	Functional Programming
5	29.-2.	February	Feb 3: Group formation event	
6	5.-9.		Feb 8: Team formation event	Assignment 1 hand-in
7	12.-16.		Project kickoff	Assignment 2 hand-in
8	19.-23.			Assignment 3 hand-in
9	26.-1.	March		Assignment 4 hand-in
10	4.-8.			
11	11-15.		Submission for Check in session 1	Assignment 5 hand-in
12	18.-22.		Check in session 1	Assignment 6 hand-in
13	25.-29.		**Easter Holiday**	
14	1.-5.	April		Assignment 7 hand-in Project kick-off
15	8.-12.			
16	15.-19.		Submission for Check in session 2	
17	22.-26.		Check in session 2	
18	29.-3.	May		
19	6.-9.			Project submission
20	13.-17.		Exam submission	
21	20.-24.			
22	27.-31.		May 30: Celebration event	May 29: Exam
23	3.-7.	June	June 3, 4, 6, 7: Exam	
24	10.-14.		June 10, 14: Exam	

Mutual recursion is required for the final red exercise of assignment 5 and is covered in week 10. Try to finish Assignment 5 in week 10 and use week 11 for the second-year project


We are coding Scrabble

Project

Write a bot that plays Scrabble
(at least our warped version of it)

- Infinite boards
- Boards with holes in them
- Tiles contain sets of characters, not just single characters
- Arbitrary languages
- Powerful board elements

Project

Write a bot that plays Scrabble
(at least our warped version of it)

To pass you must not necessarily win,
not even play particularly well, but you
must find valid moves (not just passing)
and be able to finish a game on an
infinite board

Project

- No project report! Just hand in the source code
- We will provide a back-end and multi-player infrastructure
- You focus only on playing the game
- We will provide weekly assignments that you can use in your project

Project

And after you have handed in…

Project

Tournament

- Tournament ladder
- Battle Royal?
- Drinks, relaxation and good company

Plagiarism

- You work on the assignments **individually** and the project in groups of **two or three**
- Do not copy code from other groups!
- We will run plagiarism-detecting software to make sure that you don’t
- Using code from others or AI tools like Copilot or ChatGPT and presenting it as your own work is plagiarism.
- Plagiarism may lead to suspension from the exam, suspension for one or more semesters, or in some cases even expulsion.

Plagiarism

- You work on the assignments **individually** and the project in groups of **two or three**
- Do not co
- We will r make sure that you
- Using co t or ChatGPT and pres rism.
- Plagiarism may lead to suspension from the exam, suspension for one or more semesters, or in some cases even expulsion.

Discuss high-level ideas and implementation strategies with your peers

NEVER SHARE CODE

Examination

- Five hour open-book exam
- You may use the book and physical notes that you bring
- You may not use the internet other than LearnIT
- You will need to hand in functional software that follows a strict specification
- We will give you plenty of old exams
- Mandatory assignments dramatically improve results. Take them seriously.

Introduction to FP

These slides are based on slides by
Michael R. Hansen.

Lecture outline

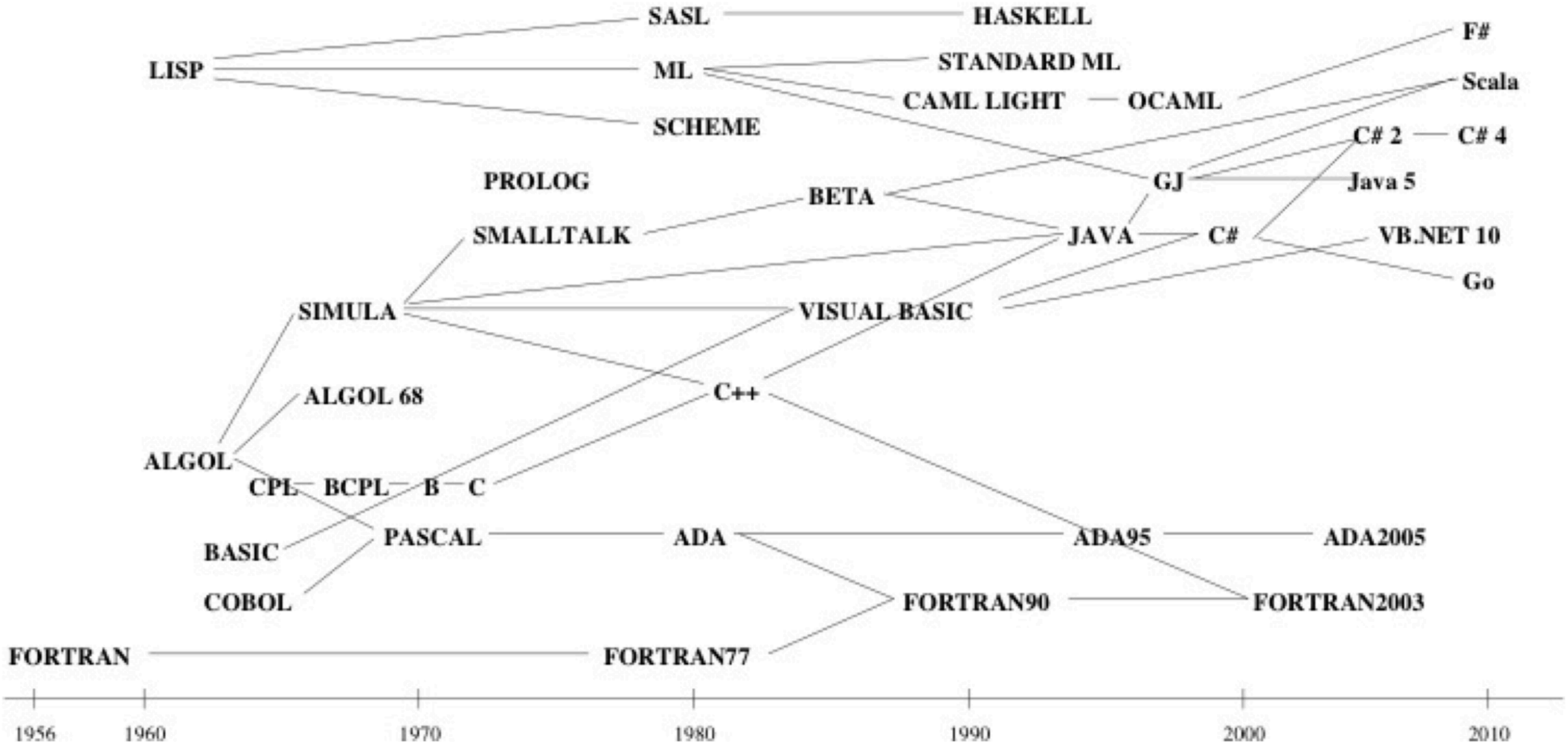
1. Introduction
2. Getting started with F#
3. Brief introduction to lists

Part 1

Introduction

- Historic perspective
- Imperative models
- Object-oriented models
- Declarative models
- Functional Programming background

Historic perspective



Imperative programming

Imperative programs are modelled using state and state-changing operations

```
fact(i):
  r := 1;
  while i > 0 do
     r := r*i;
     i := i-1;
  od
  return r;
```

Imperative models describe **how** a solution is obtained

OO programming

- Add structure to imperative models
- Objects are modelled using state and an interface of state-changing operations
- OO models include collections of object that communicate over these interfaces

Object-oriented models describe **how** a solution is obtained

Declarative programs

In declarative models, the focus is on **what** a solution is.

In functional programming, a program is expressed as a mathematical function:

```
fact(0) = 1
fact(x) = x * fact(x - 1) [if x > 0]
```

Running a functional program means evaluating a function.

Benefits of FP

- Fast prototyping
- Easy to reason about smaller features (functions) to build lager features (application of functions)
- Relatively easy to parallelise
- Functions can be composed easily
- Smaller, easier to read programs

Theoretical foundations

Alonzo Church
1903-1995

Introduced the λ-calculus around 1930

`f(x) = x + 2` is written as `λx. x + 2`

Forms the foundational underpinnings of all functional programming languages

Functional languages

- Meta Language (ML) was originally designed for theorem proving

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a formal semantics

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- Functional languages are widely used e.g. for AI, the finance industry, and compilers

A major goal

Learn abstraction (not concrete languages)

- Modelling
- Design
- Programming

Let us solve programs in a succinct, elegant and *understandable* manner

A major goal

Once mastered, functional programming paradigms are highly useful even when programming using other paradigms

Mastering functional programming will make you a better programmer

A skilled programmer is not constrained by languages nor paradigms

Part 2

Getting started with F#

- The interactive environment
- Declarations
- Recursion
- Types

F# supports

- Functions as first-class citizens
- Structured data (lists, trees, …)
- Type inference and polymorphism
- Imperative and object-oriented programming (mutable data, loops arrays)

F# supports

- Functions as first-class citizens
-
-
- programming (mutable data, loops arrays)

Imperative features will be used very sparingly in this course, but they do have their place

The REPL

F# has an interactive environment (REPL)
to help develop and debug programs

```
> 2 * 3 + 4;;
val it : int = 10
```

Input to F#
Response from F#

- **val** indicates that a value is computed
- The integer **10** is the computed value
- **int** is the type of the computed value
- The identifier **it** names the last result

The REPL

F# has an interactive environment to help develop and debug programs

```
> 2 * 3 +
val it : i
```

Input to F#

onse from F#

We write

`it ↦ 10`

as `it` is bound to `10`

- **val** indic
- The integ
- **int** is the type of the computed value
- The identifier **it** names the last result

Value declarations

We can bind values to our own identifiers

```
> let price = 25 * 5;;
val price : int = 125
```

Input to F#

Response from F#

The effect of this declaration is a binding

price ↦ 125

Environments

A collection of bindings is called an *environment*

```
> let price = 25 * 5;;
val price : int = 125
> let newPrice = price * 2;;
val newPrice : int = 250
> newPrice > 500;;
val it : bool = false
```

The above produces the environment
[price ↦ 125; newPrice ↦ 250; it ↦ false]

Function declarations

Functions are declared in a similar fashion to values

```
> let circleArea r = System.Math.PI * r * r;;
val circleArea : float -> float
```

- System.Math is a program library
- PI is an identifier of type float in System.Math
- The function type is `float -> float` (*<argument type>* `->` *<return type>*)

Function application

```
> circleArea 1.0;
val it : float = 3.141592654

> circleArea 3.2;
val it : float = 32.16990877
```

Primitive types

F# supports the standard primitive types

- int (`1, 42, 123, -5, …`)
- float (`1.0, 42.0, 1.23, -5.34, …`)
- bool (`true, false`)
- string (`“Hello \”World\””`)
- char (`‘a’, ‘B’, ‘\n’, …`)
- …

Anonymous functions

Functions can be given inline using a *function expression:*

```
> fun r -> System.Math.PI * r * r;;
val it : float -> float
```

This expression is equivalent to the function `circleArea` we defined earlier:

```
> let circleArea r =
    System.Math.PI * r * r;;
```

Pattern Matching

Function expression with patterns

```
> function
    | true -> 0
    | false -> 1
val it : bool -> int

> it false
val it : int = 1
```

A functional expression that converts booleans to integers

Pattern Matching

Function expression with patterns

```
> function
    | 0 -> true
    | 1 -> false
val it : int -> bool

> it 0
val it : bool = true
```

A functional expression that converts integers to booleans

Pattern Matching

Function expression with patterns

```
> function
    | 0 -> true
    | 1 -> false
val it : int -> bool

> it 0
val it : bool = true
```

A functional expression that converts integers to booleans

This program is problematic. Why?

Pattern Matching

Function expression with patterns

```
> function
    | 0 -> true
    | 1 -> false
val it : int -> bool

> it 0
val it : bool = true
```

A functional expression that converts integers to booleans

There are more integers than 0 and 1

Pattern Matching

Function expression with patterns

```
> function
    | 0 -> true
    | _ -> false
val it : int -> bool

> it -578
val it : bool = false
```

A functional expression that converts integers to booleans

Recursive functions

Recall mathematical definition of factorial:

```
fact(0) = 1
fact(x) = x * fact(x - 1) [if x > 0]
```

Recursive functions are declared in F# using the **rec** keyword:

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;
val fact : int -> int
```

Evaluation

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;

fact 3
```

Evaluation

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;
```

```
fact 3 ⇝ 3 * fact (3 - 1)
```

Evaluation

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;
```

```
fact 3 ⇝ 3 * fact (3 - 1) ⇝
3 * fact 2
```

Evaluation

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;
```

```
fact 3 ⇝ 3 * fact (3 - 1) ⇝
3 * fact 2 ⇝ 3 * 2 * fact(2 - 1)
```

Evaluation

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;
```

```
fact 3 ⇝ 3 * fact (3 - 1) ⇝
3 * fact 2 ⇝ 3 * 2 * fact(2 - 1) ⇝
3 * 2 * fact 1
```

Evaluation

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;
```

```
fact 3 ⇝ 3 * fact (3 - 1) ⇝
3 * fact 2 ⇝ 3 * 2 * fact(2 - 1) ⇝
3 * 2 * fact 1 ⇝
3 * 2 * 1 * fact (1 - 1)
```

Evaluation

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;
```

```
fact 3 ⇝ 3 * fact (3 - 1) ⇝
3 * fact 2 ⇝ 3 * 2 * fact(2 - 1) ⇝
3 * 2 * fact 1 ⇝
3 * 2 * 1 * fact (1 - 1) ⇝
3 * 2 * 1 * fact 0
```

Evaluation

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;
```

```
fact 3 ⇝ 3 * fact (3 - 1) ⇝
3 * fact 2 ⇝ 3 * 2 * fact(2 - 1) ⇝
3 * 2 * fact 1 ⇝
3 * 2 * 1 * fact (1 - 1) ⇝
3 * 2 * 1 * fact 0 ⇝
3 * 2 * 1 * 1
```

Evaluation

```
> let rec fact =
    function
    | 0 -> 1
    | x -> x * fact(x - 1);;
```

```
fact 3 ⇝ 3 * fact (3 - 1) ⇝
3 * fact 2 ⇝ 3 * 2 * fact(2 - 1) ⇝
3 * 2 * fact 1 ⇝
3 * 2 * 1 * fact (1 - 1) ⇝
3 * 2 * 1 * fact 0 ⇝
3 * 2 * 1 * 1 ⇝ 6
```

Pairs

Pairs consist of two values

```
> (5, 3);;
val it : int * int = (5, 3)

> (System.Math.PI, false)
val it : float * bool = (3.141592654, false)

> (“A function!!!”, fact)
val it : string * (int -> int) =
…
```

Pairs

```
> (5,
val it

> (Sys
val it                                    lse)

> ("A function!!", fact)
val it : string * (int -> int) =
…
```

Functions are first-class citizens.

They can be used as data just as well as any other value

There is conceptually no difference between providing an string as an argument or a function as an argument, as demonstrated here

If-expressions

If-expressions have the form

```
if b then e1 else e2
```

Evaluation of if-expressions:

```
if true then e1 else e2 ⇝ e1
if false then e1 else e2 ⇝ e2
```

Example

```
> 1 + (if 1 > 1 then 1 else 2);;
val it : int = 3
```

If-expressions

A comparison

```
let rec fact =
function
| 0 -> 1
| x -> x * fact(x - 1)
```

```
let rec fact x =
    if x = 0 then
        1
    else
        x * fact(x - 1)
```

If-expressions

A comparison

```
let rec fact =
function
| 0 -> 1
| x -> x * fact(x - 1)
```

```
let rec fact x =
    if x = 0 then
        1
    else
        x * fact(x - 1)
```

If-expressions naturally have their uses, but pattern matching is often cleaner

Currying functions

How many arguments does the following function take?

```
let f (x, y) = x + y
```

Currying functions

How many arguments does the following function take?

```
let f (x, y) = x + y
```

It takes one argument!!!
It takes one pair of two integers
`f : (int * int) -> int`

Currying functions

How many arguments does the following function take?

```
let f (x, y) = x + y
```

It takes one argument!!!
It takes one pair of two integers
`f : (int * int) -> int`

Why do we care?

Currying functions

How many arguments does the following function take?

```
let f x y = x + y
```

Currying functions

How many arguments does the following function take?

```
let f x y = x + y
```

It takes two arguments (sort of)

```
f : int -> int -> int
```

Currying functions

How many arguments does the following function take?

```
let f x y = x + y
```

```
f 5 ⇝ ???
```

Currying functions

How many arguments does the following function take?

```
let f x y = x + y
```

```
f 5 ⇝ fun y -> 5 + y
```

Currying functions

How many arguments does the following function take?

```
let f x y = x + y
```

```
f 5 ⇝ fun y -> 5 + y
```

This is called partial application

Currying functions

This is important because

- Partial application allows us to specialise a function for future use
- Immensely powerful with first class functions

Currying functions

This is important because

- There are always edge-cases, but (with very few exceptions) functions should be curried rather than taking one giant tuple as an argument.
- s

Doing so is simply too restrictive

Type inference

What is the type of this function?

```
let rec pow =
  function
  | (x, 0) -> 1.0
  | (x, n) -> x * pow (x, n - 1)
```

Type inference

What is the type of this function?

```
let rec pow =
  function
  | (x, 0) -> 1.0
  | (x, n) -> x * pow (x, n - 1)
```

- `pow` must have type `t1 * t2 -> t3`
  - t3 must be float (1.0 is float)
  - t2 must be int (0 is int)
  - `x * pow (x, n - 1)` must have type float (t3 is float)

Type inference

What is the type of this function?

```
let rec pow =
  function
  | (x, 0) -> 1.0
  | (x, n) -> x * pow (x, n - 1)
```

- `pow` must have type `t1 * int -> float`
  - Since `x * pow (x, n - 1)` must have type float, we know `x` has type float
  - Therefore, `t1` is float

Type inference

What is the type of this function?

```
let rec pow =
  function
```

Type inference is done automatically by F#

- Since x * pow (x, n - 1) must have type float, we know x has type float
- Therefore, t1 is float

Part 3

Brief introduction to lists

- Values
- Types
- Recursion on lists

Lists

- Lists work similarly to linked lists in Java or C#
- Recursion follows the structure of the lists
- Lists contain elements of a certain type

Syntax

`[e1; e2; …; en]` (`[]` is the empty list)

Lists

- Lists work similarly to linked lists in
- e
- ype

This is a high-level overview. We will cover things in more detail later

Syntax

`[e1; e2; …; en]` (`[]` is the empty list)

List examples

```
> [5; 3; -23];;
val it : int list = [5; 3; -23]

> [System.Math.PI; 0.0]
val it : float list = [3.141592654; 0.0]
```

List constructors

Lists are generated as follows:

- `[]` is the empty list
- `x :: xs` is a list with *head* `x` and *tail* `xs`

e.g `5 :: [3; -23] = [5; 3; -23]`

```
> 5 :: 3 :: -23 :: [];;
val it : int list = [5; 3; -23]
```

Summing a list

How do we define a function `sumList` that adds up the elements of a list?

```
> sumList [5; 3; 2];;
val it : int = 10
```

What should the results of the following two calls of `sumList` be?

```
sumList []        = ???
sumList (x :: xs) = ???
```

Summing a list

What should the result of the following be?

```
sumList []          = 0
sumList (x :: xs) = x + sumList xs
```

From these two equations we can read off the recursive implementation:

Summing a list

What should the result of the following be?

```
sumList []          = 0
sumList (x :: xs) = x + sumList xs
```

From these two equations we can read off the recursive implementation:

```
let rec sumList =
  function
  | []      -> 0
  | x :: xs -> x + sumList xs
```

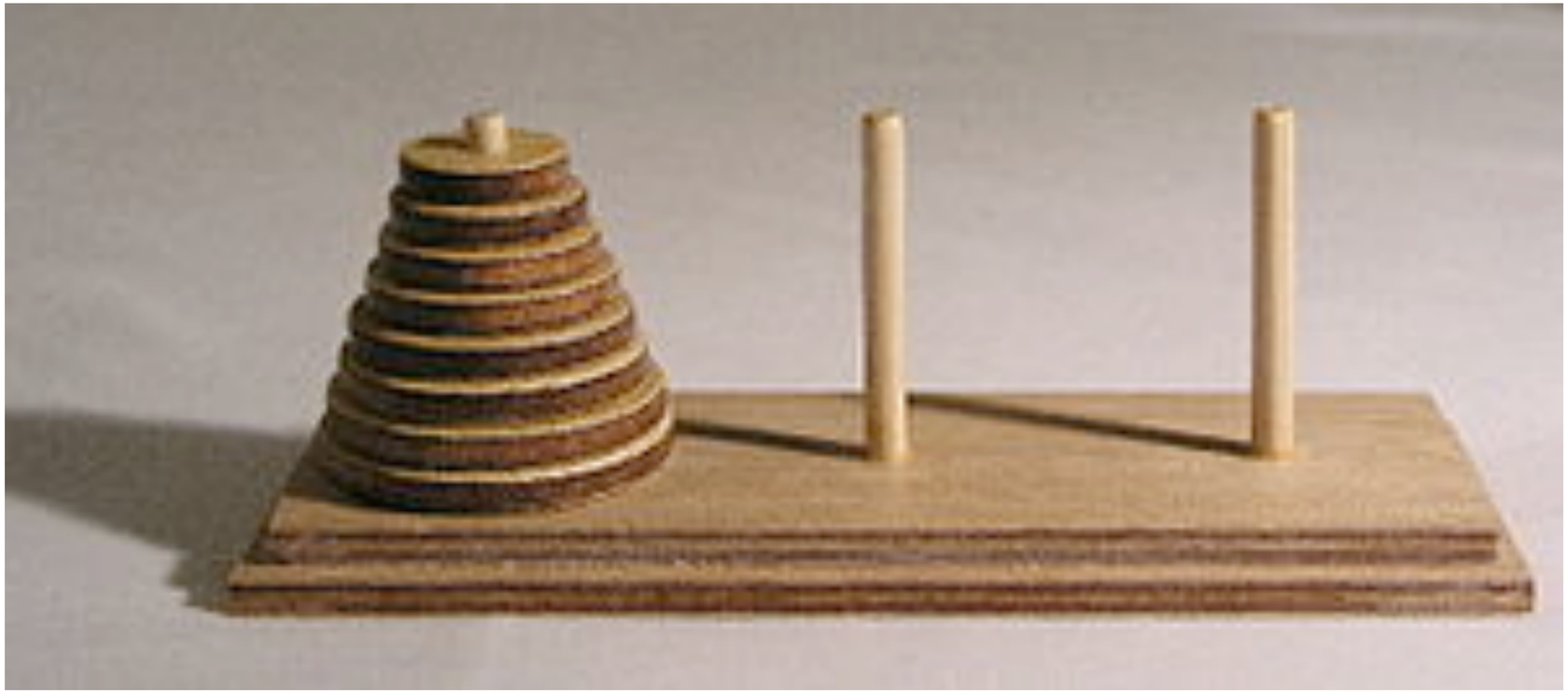
Appending lists

We can also join lists using @

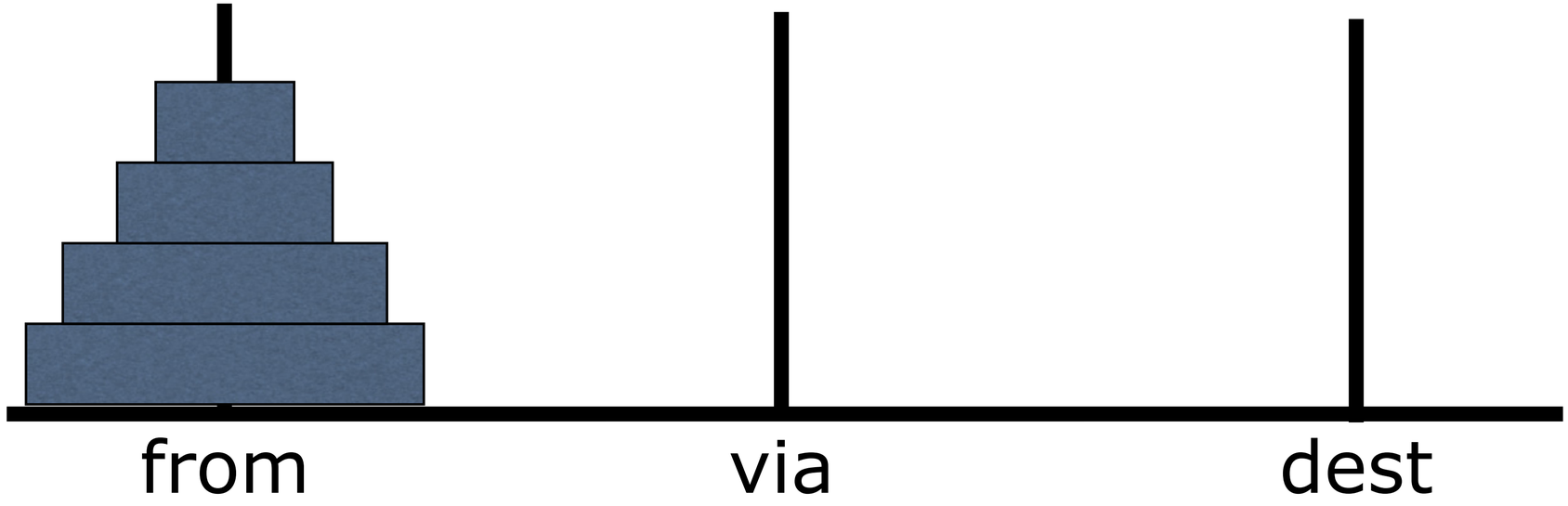
```
> [5; 3]@(-23::[]);;
val it : int list = [5; 3; -23]
```

Questions?

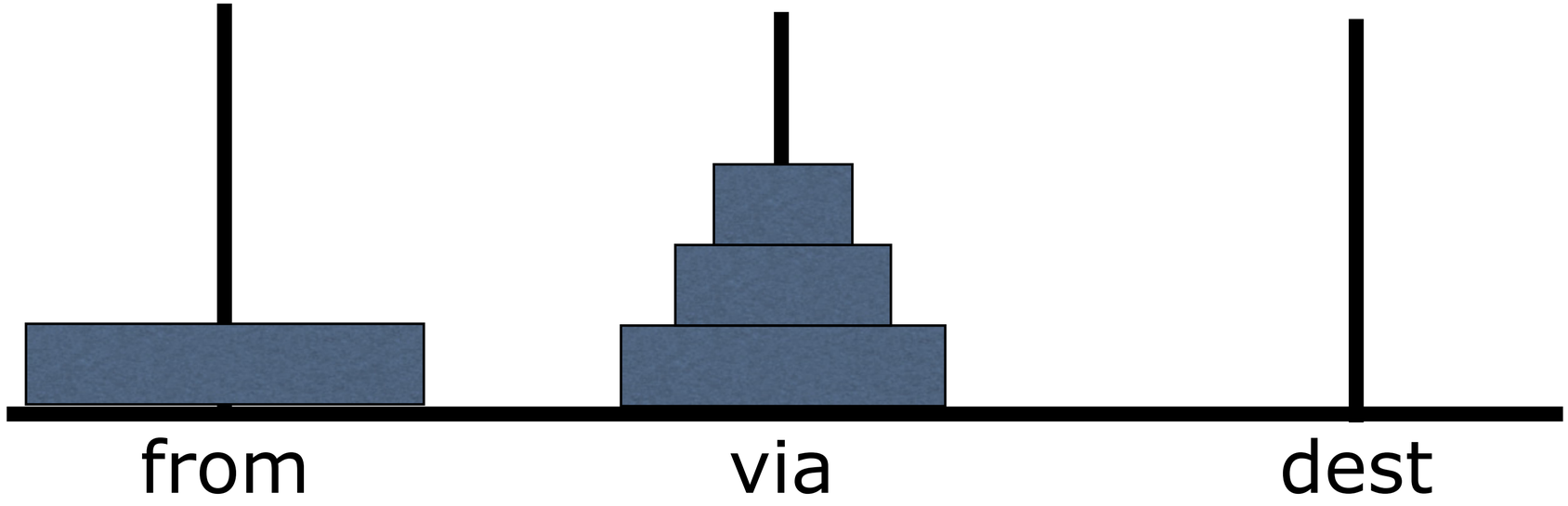
Bonus: The towers of Hanoi



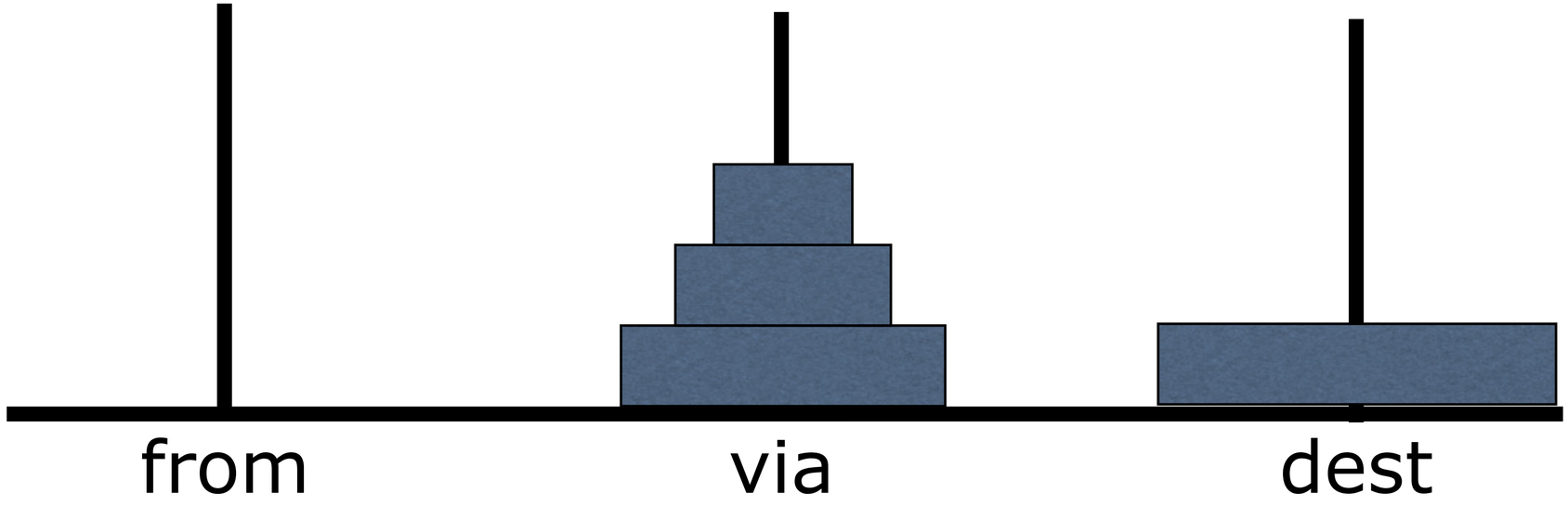
Informal reasoning



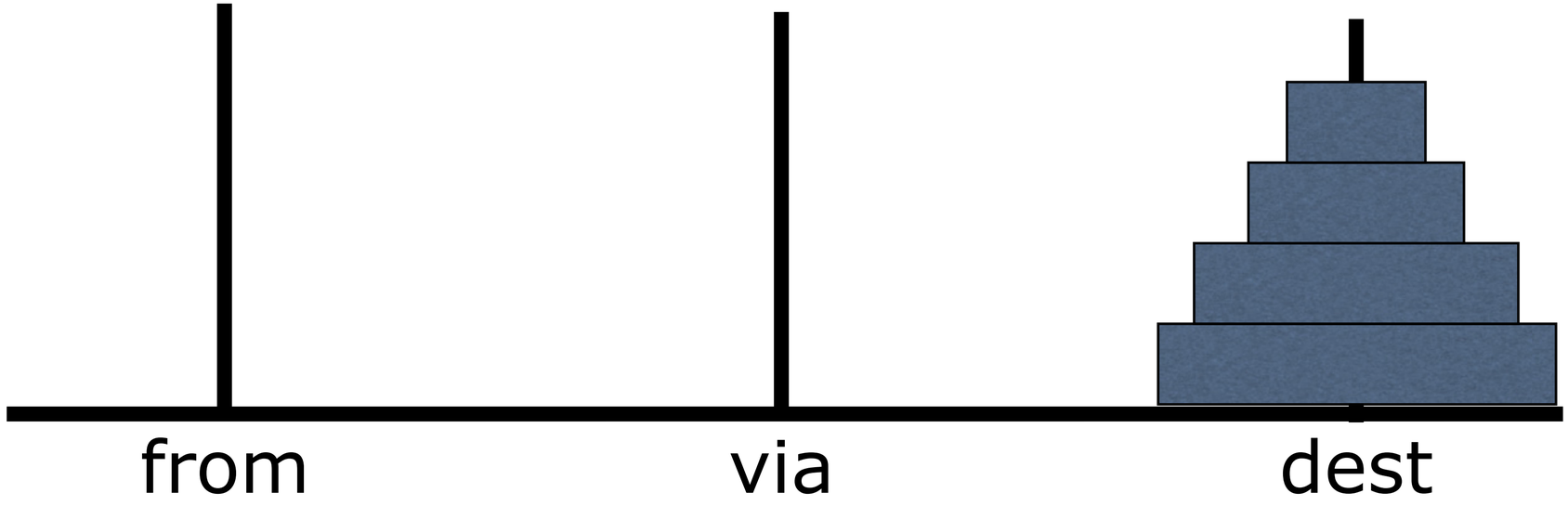
Informal reasoning



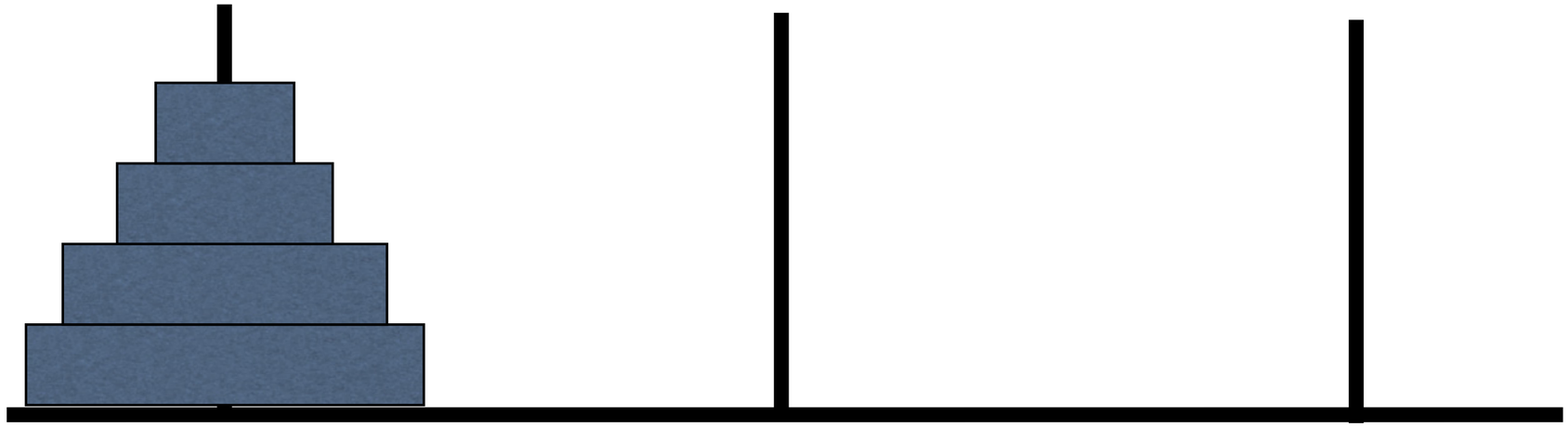
Informal reasoning



Informal reasoning



Informal reasoning

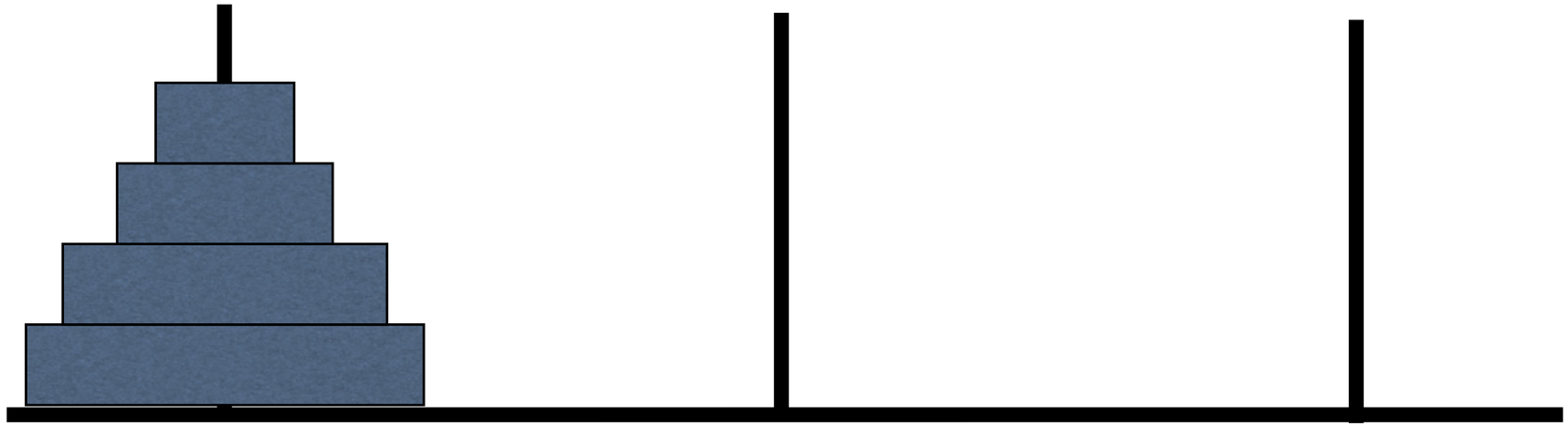


Write a function

```
hanoi : char -> char -> char ->
        int -> (char * char) list
```

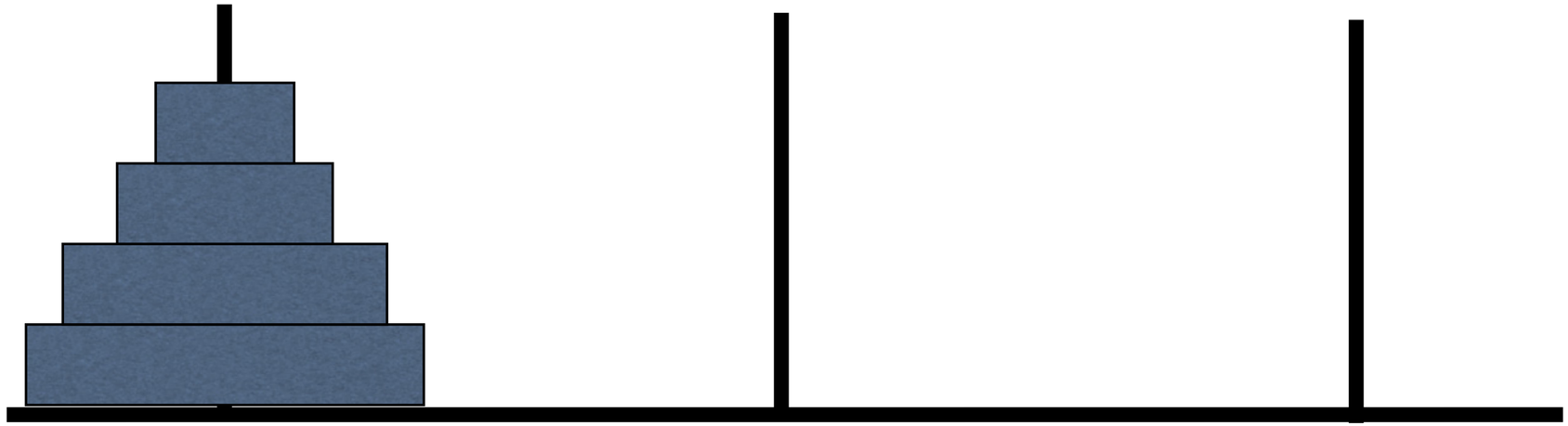
That given three pegs (named by characters) and a tower height returns a list of moves for a winning game

Informal reasoning



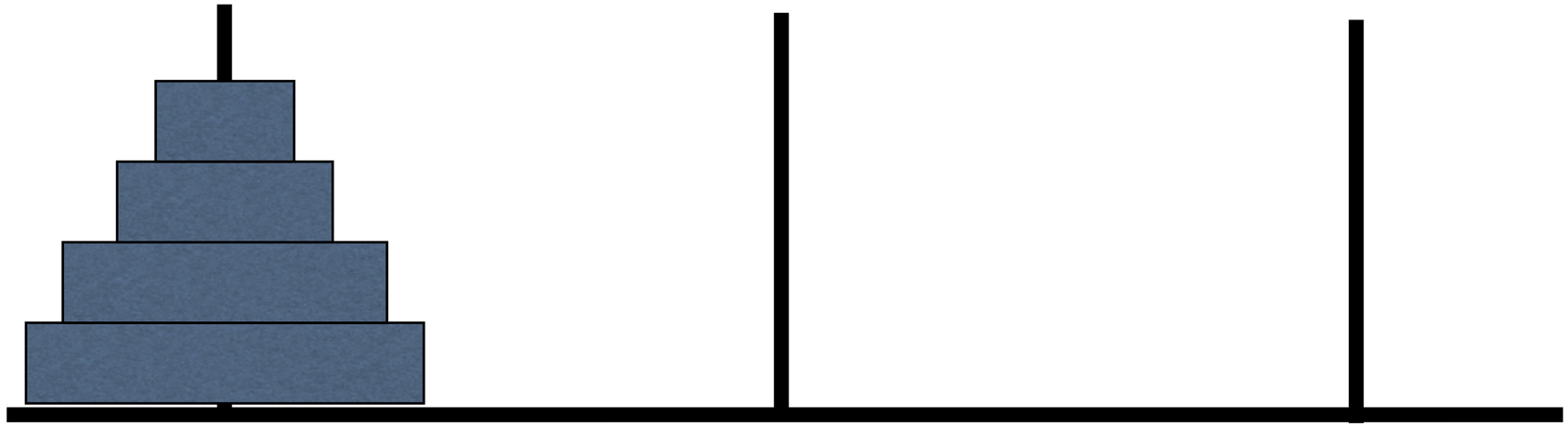

```
let rec hanoi from via dest = ???
```

Informal reasoning



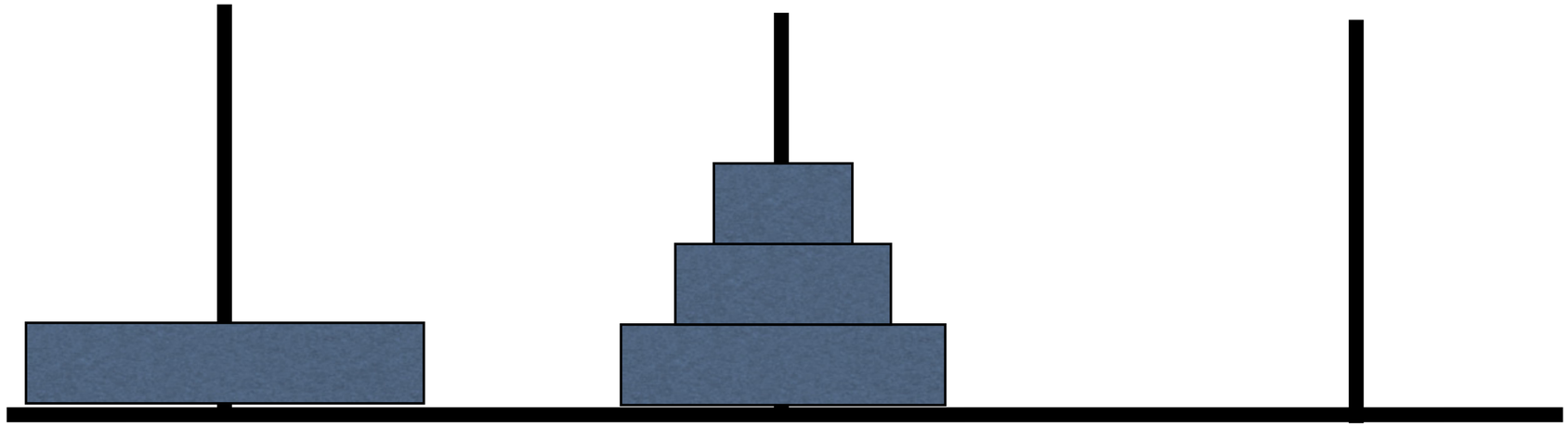

```
let rec hanoi from via dest =
    function
    | 0 -> []
```

Informal reasoning



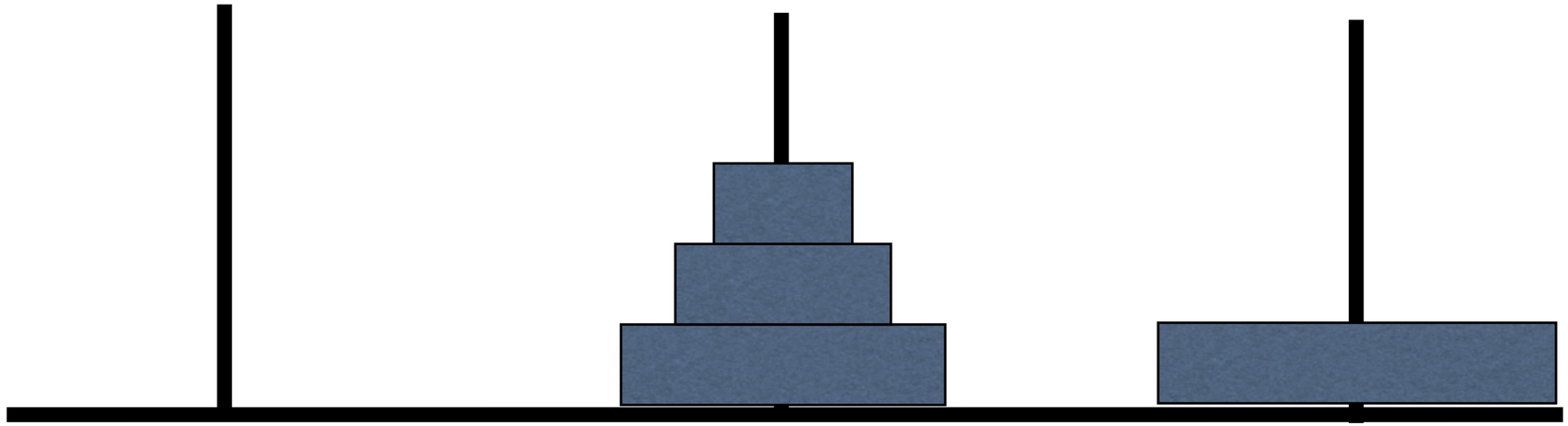

```
let rec hanoi from via dest =
    function
    | 0 -> []
```

Informal reasoning



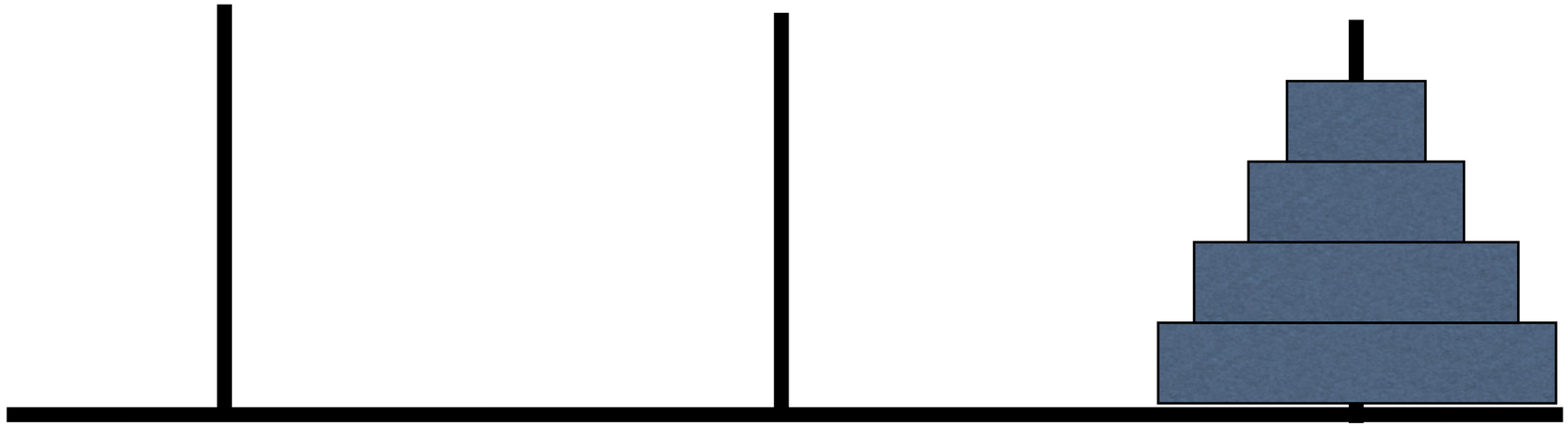

```
let rec hanoi from via dest =
    function
    | 0 -> []
    | x -> hanoi from dest via (x - 1) @
```

Informal reasoning




```
let rec hanoi from via dest =
    function
    | 0 -> []
    | x -> hanoi from dest via (x - 1) @
                (from, dest)
```

Informal reasoning




```
let rec hanoi from via dest =
    function
    | 0 -> []
    | x -> hanoi from dest via (x - 1) @
           (from, dest) ::
           hanoi via from dest (x - 1)
```

```
> hanoi 'A' 'B' 'C' 1
val it: (char * char) list = [('A', 'C')]


> hanoi 'A' 'B' 'C' 2
val it: (char * char) list =
   [('A', 'B'); ('A', 'C'); ('B', 'C')]


> hanoi 'A' 'B' 'C' 3
val it: (char * char) list =
  [('A', 'C'); ('A', 'B'); ('C', 'B');
   ('A', 'C'); ('B', 'A'); ('B', 'C'); ('A', 'C')]
```

```
> hanoi 'A' 'B' 'C' 6
val it: (char * char) list =
  [('A', 'B'); ('A', 'C'); ('B', 'C'); ('A', 'B'); ('C', 'A'); ('C', 'B');
   ('A', 'B'); ('A', 'C'); ('B', 'C'); ('B', 'A'); ('C', 'A'); ('B', 'C');
   ('A', 'B'); ('A', 'C'); ('B', 'C'); ('A', 'B'); ('C', 'A'); ('C', 'B');
   ('A', 'B'); ('C', 'A'); ('B', 'C'); ('B', 'A'); ('C', 'A'); ('C', 'B');
   ('A', 'B'); ('A', 'C'); ('B', 'C'); ('A', 'B'); ('C', 'A'); ('C', 'B');
   ('A', 'B'); ('A', 'C'); ('B', 'C'); ('B', 'A'); ('C', 'A'); ('B', 'C');
   ('A', 'B'); ('A', 'C'); ('B', 'C'); ('B', 'A'); ('C', 'A'); ('C', 'B');
   ('A', 'B'); ('C', 'A'); ('B', 'C'); ('B', 'A'); ('C', 'A'); ('B', 'C');
   ('A', 'B'); ('A', 'C'); ('B', 'C'); ('A', 'B'); ('C', 'A'); ('C', 'B');
   ('A', 'B'); ('A', 'C'); ('B', 'C'); ('B', 'A'); ('C', 'A'); ('B', 'C');
   ('A', 'B'); ('A', 'C'); ('B', 'C')]
```