

Functional Programming

WS 2012/13

Harald Zankl (VO)
Cezary Kaliszyk (PS) Thomas Sternagel (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 4

Week 4 - Trees

Summary of Week 3

L-Strings

- ▶ `strings` not functional in OCaml
- ▶ therefore use module `Strng`

L-Strings as character lists

```
type t = char list
val of_string : string -> char list
val to_string : char list -> string
val of_int : int -> char list
val print : char list -> unit
val toplevel_printer : Format.formatter -> char list -> unit
val blanks : int -> t
```

Setting Up the Interpreter

- ▶ `.ocamlinit` (searched in `.` and `~`)
- ▶ write modules for custom interpreter to `file.mltop`
- ▶ compile with `'ocamlbuild file.top'`
- ▶ start with `'./file.top'`

Example

```

AsciiArt
Lst
Picture
Strng
w03.mltop

```

This Week

Practice I

OCaml introduction, lists, strings, **trees**

Theory I

lambda-calculus, evaluation strategies, induction, reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, monads, ...

What Are Trees?

Definition (Tree)

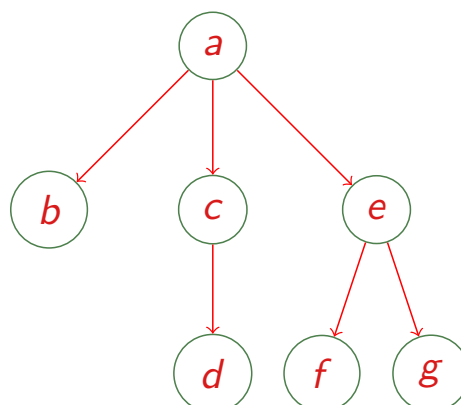
(rooted) tree $T = (N, E)$

- ▶ set of nodes N
- ▶ set of edges $E \subseteq N \times N$
- ▶ unique root of T ($root(T) \in N$) without predecessor
- ▶ all other nodes have exactly one predecessor
- ▶ leaf is node without successor

What Are Trees? (cont'd)

Example

- ▶ $N = \{a, b, c, d, e, f, g\}$
- ▶ $E = \{(a, b), (a, c), (a, e), (c, d), (e, f), (e, g)\}$
- ▶ $root(T) = a$
- ▶ $leaves(T) = \{b, d, f, g\}$
- ▶ $T =$

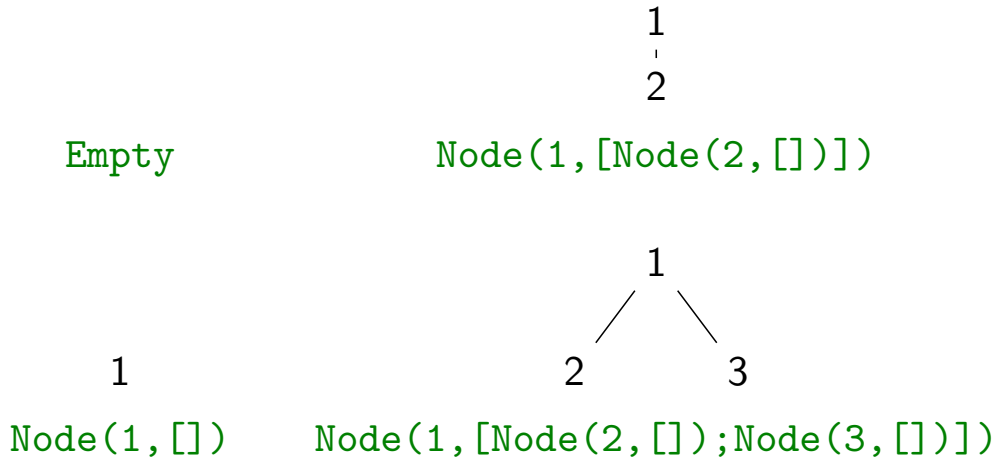


Trees in OCaml

Type

`type 'a tree =` empty tree `Empty |` node with content `Node of 'a * 'a tree list`

Example



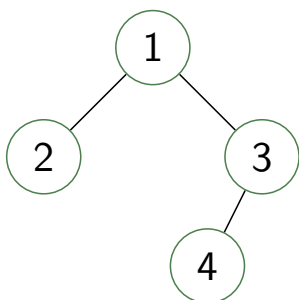
Restricting the Branching-Factor

Definition (Binary tree)

restrict number of successors (maximal 2)

Type

`type 'a t = Empty | Node of ('a t * 'a * 'a t)`



`Node(Node(Empty, 2, Empty),`
`1,`
`Node(Node(Empty, 4, Empty), 3, Empty))`

Functions on BinTrees

Definition (Size)

size of a tree equals number of nodes

```
let rec size = function Empty      -> 0
                    | Node(l,_,r) -> size l + size r + 1
```

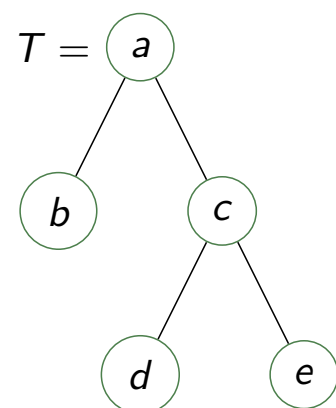
Definition (Height)

height of a tree is length of longest path from root to some leaf

```
let rec height = function
  | Empty      -> 0
  | Node(l,_,r) -> max (height l) (height r) + 1
```

Example

- ▶ convention: do not draw 'Empty' nodes
- ▶ **size** $T = 5$
- ▶ **height** $T = 3$

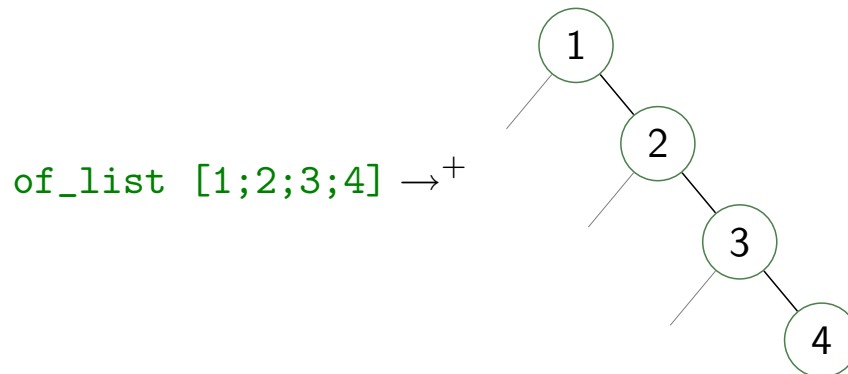


Creating Trees of Lists

The easy way

```
let rec of_list = function []    -> Empty
                       | x::xs -> Node(Empty,x,of_list xs)
```

Example

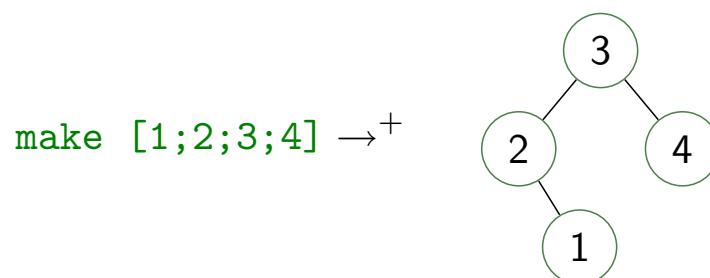


Creating Trees of Lists (cont'd)

The fair way

```
let rec make = function
  | [] -> Empty
  | xs ->
    let m = Lst.length xs / 2 in
    let (ys,zs) = Lst.split_at m xs in
    Node (make ys,Lst.hd zs,make(Lst.tl zs))
```

Example

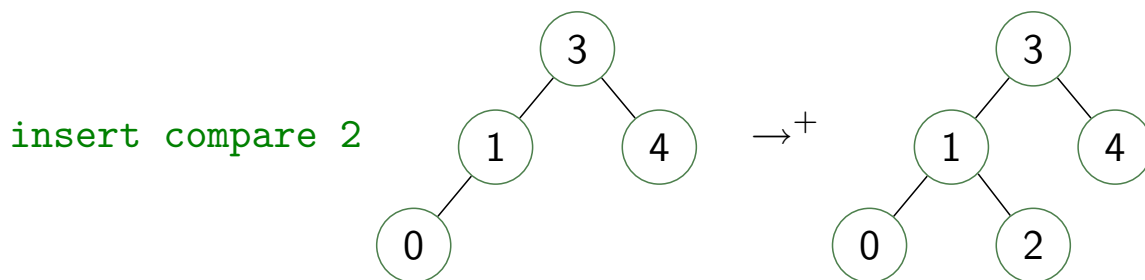


Creating Trees of Lists (cont'd)

Ordered insertion

```
let rec insert c v = function
| Empty      -> Node(Empty,v,Empty)
| Node(l,w,r) -> if c v w < 1 then Node(insert c v l,w,r)
                  else Node(l,w,insert c v r)
```

Example

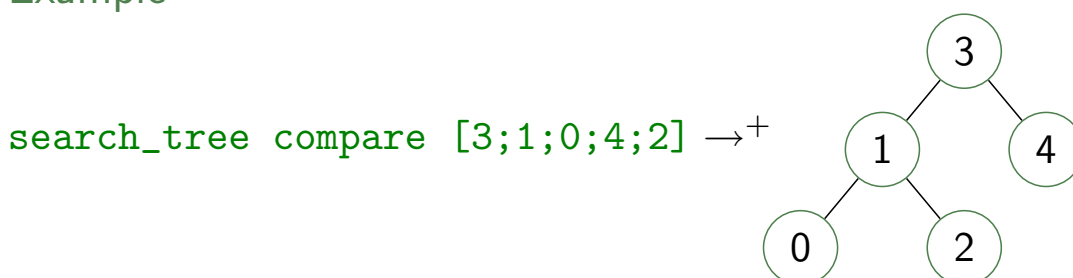


Creating Trees of Lists (cont'd)

Search trees

```
let search_tree c = Lst.foldl (fun t v -> insert c v t) Empty
```

Example

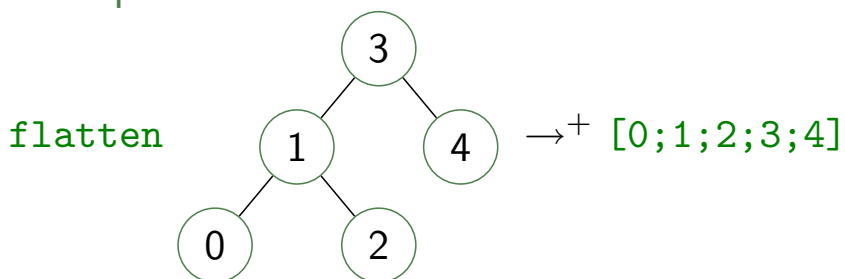


Transforming Trees Into Lists

Flatten

```
let rec flatten = function
  | Empty      -> []
  | Node(l,v,r) -> (flatten l)@(v::flatten r)
```

Example



A Sorting Algorithm for Lists

```
let sort c xs = BinTree.flatten(BinTree.search_tree c xs)
```


The Idea

Reduce storage size

- ▶ ASCII uses **1 byte** per character
- ▶ encode frequent characters '**short**'

Example

Text: 'text'

- ▶ **32 bits** in ASCII (**01110100011001010111100001110100**)

- ▶ using

t	↦	0
e	↦	10
x	↦	11

6 bits needed (**010110**)

Some More Useful List Functions

```
let concat xs = foldr (@) [] xs
```

```
let rec take_while p = function
| []      -> []
| x::xs -> if p x then x :: take_while p xs else []
```

```
let rec drop_while p = function
| []      -> []
| x::xs as list -> if p x then drop_while p xs else list
```

```
let span p xs = (take_while p xs, drop_while p xs)
```

```
let rec until p f x = if p x then x else until p f (f x)
```

Counting Symbol Frequency

Collate

```
let rec collate = function
| []          -> []
| w::ws as xs ->
  let (ys,zs) = Lst.span ((=)w) xs in
  (Lst.length ys,w) :: collate zs
```

Example

```
collate ['a';'a';'b';'c';'c';'c'] →+
  [(2,'a');(1,'b');(3,'c')]

collate ['a';'a';'b';'a';'a';'a'] →+
  [(2,'a');(1,'b');(3,'a')]
```

Generating a Symbol-Frequency List

Sample

```
let sample xs = sort compare (collate(sort compare xs))
```

Example

```
sample ['t';'e';'x';'t'] →+ [(1,'e');(1,'x');(2,'t')]
```

Huffman Trees

- ▶ **leaf nodes** contain weight (= frequency) + character
- ▶ **other nodes** store sum of weights of subtrees

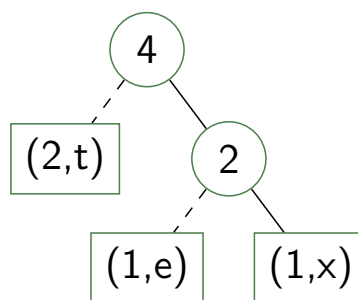
Type

```
type 'a option = None | Some of 'a (predefined)
```

```
type node = (int * char option)
```

```
type t = node btree
```

Example



Building the Huffman Tree

Step 1

- ▶ transform the symbol-frequency list into a list of Huffman trees

```
let mknode (w,c) = Node(Empty,(w,Some c),Empty)
```

Example

```
Lst.map mknode [(1,'e');(1,'x');(2,'t')]
→+ [(1,e);(1,x);(2,t)]
```

Building the Huffman Tree (cont'd)

Step 2

- ▶ combine first two trees until only one left

```
let weight = function
| Node(_, (w, _), _) -> w
| _                   -> failwith "empty tree"

let combine = function
| xt::yt::xts -> let w = weight xt + weight yt in
  insert (Node(xt, (w, None), yt)) xts
| _        -> failwith "length has to be greater than 1"

let insert vt wts =
  let (xts, yts) =
    Lst.span (fun x -> weight x <= weight vt) wts in
  xts@(vt::yts)
```

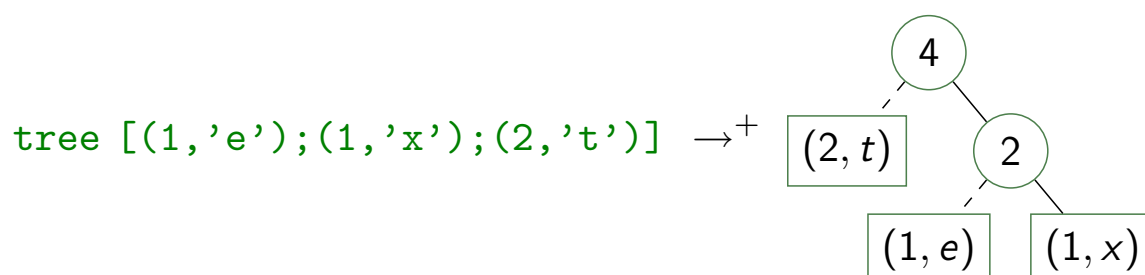
Building the Huffman Tree (cont'd)

Step 2 (cont'd)

- ▶ combine first two trees until only one left

```
let tree xs =
  Lst.hd(Lst.until is_singleton combine (Lst.map mknode xs))
```

Example

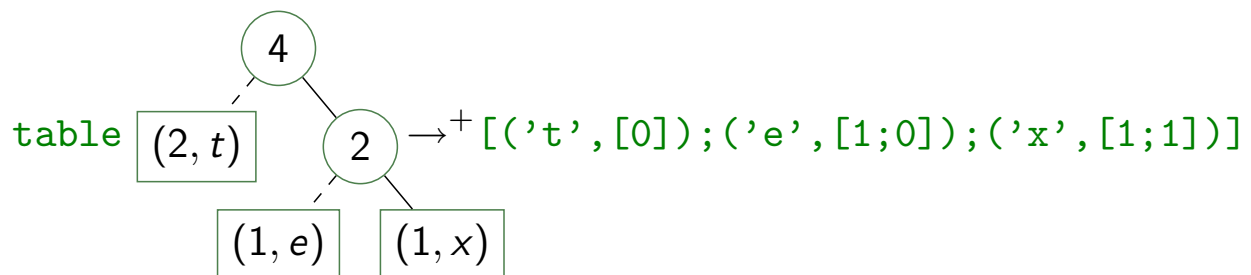


Generating a Code-Table

Encoding

- ▶ Which code corresponds to a given character?

Example



Generating a Code-Table (cont'd)

Encoding

- ▶ Which code corresponds to a given character?

```
let rec table = function
| Node(Empty, (_, Some c), Empty) -> [(c, [])]
| Node(l, _, r)                    ->
  (Lst.map (fun (c, code) -> (c, 0::code)) (table l)) @
  (Lst.map (fun (c, code) -> (c, 1::code)) (table r))
| _ -> failwith "the Huffman tree is empty"
```

Encoding

- ▶ use code-table for compression

```
let encode t text = Lst.concat(Lst.map (lookup t) text)
```

```
let rec lookup xbs v = match xbs with
| ((x,bs)::xbs) -> if x = v then bs else lookup xbs v
| _              -> failwith "not_found"
```

Example

```
encode
  [('t',[0]);('e',[1;0]);('x',[1;1])]
  ['t';'e';'x';'t']
→+ [0;1;0;1;1;0]
```

Decoding

- ▶ use Huffman tree for decompression

```
let rec decode_char = function
| (Node(Empty,(_,Some c),Empty),cs) -> (c,cs)
| (Node(xt,_,_),0::cs)              -> decode_char (xt,cs)
| (Node(_,_,xt),1::cs)             -> decode_char (xt,cs)
| _                                  -> failwith "empty_tree"
```

```
let rec decode t = function
| [] -> []
| xs -> let (c,xs) = decode_char (t,xs) in c::decode t xs
```