Functional Programming

WS 2007/08

Christian Sternagel[1] (VO + PS)
Friedrich Neurauter[2] (PS)
Harald Zankl[3] (PS)

Computational Logic
Institute of Computer Science

University of Innsbruck

9 November 2007

[1] christian.sternagel@uibk.ac.at
[2] friedrich.neurauter@uibk.ac.at
[3] harald.zankl@uibk.ac.at

Overview

Overview

Exercises

The first test has been moved to November 30

L-Strings

- strings not functional in OCaml
- therefore use module Strng

Strings as character lists

```
type t = char list
val center : int -> t -> t
val join : 'a list -> 'a list list -> 'a list
val left_justify : int -> t -> t
val of_int : int -> t
val of_string : string -> t
val print : t -> unit
val right_justify : int -> t -> t
val to_string : t -> string
val toplevel_printer : t -> unit
```

Setting Up the Interpreter

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

Example

```
Lst
Picture
Strng
```

`w03.mltop`

Overview

What Are Trees?

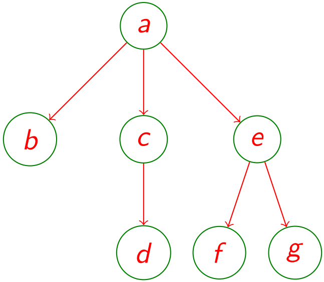
Definition (Tree)

(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

Example

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



Trees in OCaml

Type

empty tree (over Empty)

```
type 'a tree = Empty | Node of 'a * 'a tree list;;
```

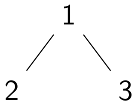
node with content (under Node)

Example

Empty

Node (1, [Node (2, [])])

1



Node (1, [])

Node (1, [Node (2, []); Node (3, [])])

Overview

Restricting the Branching-Factor

Definition (Binary tree)

restrict number of successors (maximal 2)

Type

```
type 'a btree = Empty | Node of 'a btree * 'a * 'a btree;;
```

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
;;
```

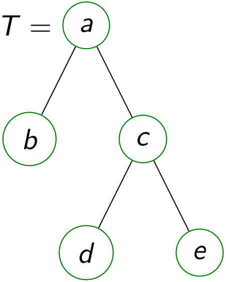
Definition (Height)

height of a tree equals length of longest path from root to some leaf plus 1

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

Example

- size $T = 5$
- height $T = 3$

$T =$ 

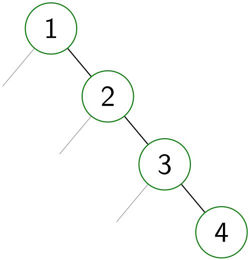
Creating Trees of Lists

The easy way

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

Example

of_list [1;2;3;4] →

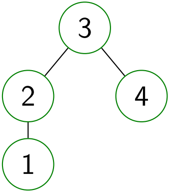


Creating Trees of Lists (cont'd)

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

make [1;2;3;4] $\to^{+}$ 

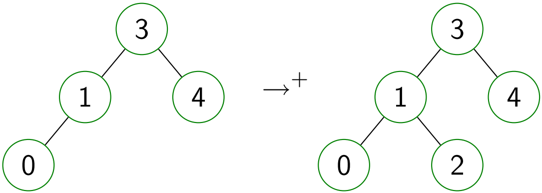
Creating Trees of Lists (cont'd)

Ordered insertion

```
let rec insert c v = function
  | Empty -> Node (Empty, v, Empty)
  | Node (l, w, r) ->
    if c v w <= 0 then Node (insert c v l, w, r) else Node (l, w, insert c v r)
;;
```

Example

insert compare 2



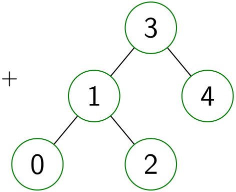
Creating Trees of Lists (cont'd)

Search trees

```
let search_tree c xs = Lst.fold_left (fun x y -> insert c y x) Empty xs;;
```

Example

search_tree compare [3; 1; 0; 4; 2] $\to^+$

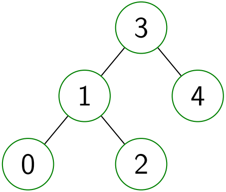


Transforming Trees Into Lists

Flatten

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

Example

flatten  $\to^+$ [0; 1; 2; 3; 4]

A Sorting Algorithm for Lists

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

Overview

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	$\mapsto$	0
e	$\mapsto$	10
x	$\mapsto$	11

6 bits needed (010110)

Some More Useful List Functions

```
 | x :: xs as ys -> if p x then drop_while p xs else ys
;;
let span p xs = (take_while p xs, drop_while p xs);;
let rev xs =
 let rec rev acc = function
  | [] -> acc
  | x :: xs -> rev (x :: acc) xs
 in
 rev [] xs
```

Some More Useful List Functions (cont'd)

```
;;
let rec until p f x = if p x then x else until p f (f x);;
let concat xs = fold append [] xs;;
```

Counting Symbol Frequency

Collate

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

Example

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

Generating a Symbol-Frequency List

Sample

```
let sample xs =
 sort (fun (c, v) (d, w) ->
  compare (v, c) (w, d)) (collate (sort compare xs))
;;
```

Example

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

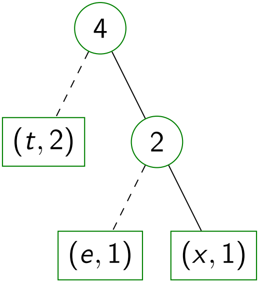
Huffman Trees

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

Type

```
type node = int * char option;;
type htree = node BinTree.t;;
```

Example



Building the Huffman Tree

Step 1

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

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

Example

Lst.map mknode [('e', 1); ('x', 1); ('t', 2)] = [$(e, 1)$; $(x, 1)$; $(t, 2)$]

Building the Huffman Tree (cont'd)

Step 2

- combine first two trees until only one left

```
let insert vt wts =
 let (xts, yts) = Lst.span (fun x -> weight x <= weight vt) wts in
 Lst.append xts (vt :: yts)
;;
let combine = function
 | xt :: yt :: xts ->
  let w = weight xt + weight yt in insert (Node (xt, (w, None), yt)) xts
 | _ -> failwith "Huffman.combine: length has to be greater than 1"
;;
```

Building the Huffman Tree (cont'd)

Step 2 (cont'd)

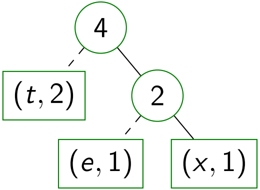
- combine first two trees until only one left

```
let singleton xs = Lst.length xs = 1;;

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

Example

tree [('e', 1); ('x', 1); ('t', 2)] =



Generating a Code-Table

Encoding

- Which code corresponds to a given character?

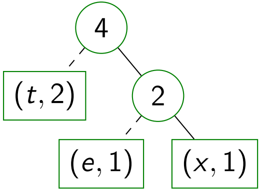
```
let rec table = function
 | Node (Empty, (_, Some c), Empty) -> [(c, [])]
 | Node (l, _, r) ->
   Lst.append
    (Lst.map (fun (c, code) -> (c, 0 :: code)) (table l))
    (Lst.map (fun (c, code) -> (c, 1 :: code)) (table r))
 | _ -> failwith "Huffman.table: the Huffman tree is empty"
;;
let rec lookup xbs v = match xbs with
 | ((x, bs) :: xbs) -> if x = v then bs else lookup xbs v
 | _ -> failwith "Huffman.lookup: not found"
;;
```

Generating a Code-Table (cont'd)

Encoding

- Which code corresponds to a given character?

Example

table 

= [('t', [0]); ('e', [1; 0]); ('x', [1; 1])]

Encoding

- use code-table for compression

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

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 "Huffman.decode: empty tree"
;;
let rec decode t = function
  | [] -> []
  | xs -> let (c, xs) = decode_char (t, xs) in c :: decode t xs
;;
```