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

19 October 2007

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

Week 3 - Strings

Overview

Overview

Week 3 - Strings

Lists

Syntax

- [] 'nil', the empty list
- :: 'cons', add element
- [1; 2; 3] syntactic sugar

Functions

- Lst.hd first element
- Lst.tl all but first
- Lst.init create list
- Lst.map apply function to list elements
- Lst.fold combine list elements by function

Modules

Using Files

- implementation files (`.ml`)
- signature files (`.mli`)
- ADTs - abstract data types (e.g., Stck)

Inline

- **module** *Imp* = **struct** ... **end**
- **module type** *Sig* = **sig** ... **end**
- **module** *Module* : *Sig* = *Imp*

Modules (cont'd)

Implementation

- type declarations, function definitions, constants
- '**type** *type* = ... ;;' for types
- '**let** *name* = ... ;;' for values

Signature

- types, values
- '**type** *type* [= ...]' for types (possibly abstract)
- '**val** *name* : *type*' for values

Overview

Week 3 - Strings

Built-In Type string

Syntax

- constructed using double quotes `""`
- concatenation: $(\hat{}) :$ string $->$ string $->$ string

Not functional!
Demonstration

Overview

Week 3 - Strings

Strng: A String Implementation Using Lists

`strng.ml`

- install type abbreviation **type** t = char list;;
- advantage: all list functions can be used for l-strings
- of_string : string −> t
- to_string : t −> string
- of_int : int −> t
- print : t −> unit

Nice Interpreter Output

Toplevel directives

- ▶ always start with # and end with ;;
- ▶ #cd "dir";; change directory
- ▶ #install_printer name;; change output function for certain type
- ▶ #load "file.cmo";; load bytecode
- ▶ #quit;; exit the interpreter
- ▶ #remove_printer name;; remove output function for certain type
- ▶ #trace fun;; trace computation of function
- ▶ #untrace fun;; stop tracing of function
- ▶ #use "file";; execute file content

Nice Interpreter Output (cont'd)

`.ocamlinit`

```
#cd "_build/";;
#install_printer Strng.toplevel_printer;;
#install_printer Picture.toplevel_printer;;
```

Strng: Implementation

```
type t = char list;;

let of_string s =
 let rec of_string i m s =
  if i = m then [] else String.get s i :: of_string (i + 1) m s
 in of_string 0 (String.length s) s
;;
let to_string xs =
 let rec to_string i s = function
  | [] -> s
  | x :: xs -> String.set s i x; to_string (i + 1) s xs
 in to_string 0 (String.create (Lst.length xs)) xs
;;
let of_int i = of_string (string_of_int i);;
let print s = Format.printf "%s\n" (to_string s);;
let toplevel_printer s =
 Format.printf "\"%s\"" (String.escaped (to_string s))
```

Overview

The Picture Analogon

Picture

- ▶ **atomic part**: pixel
- ▶ **height** and **width**
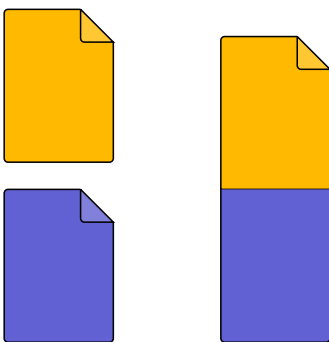- ▶ **white** pixel

L-String

- ▶ **atomic part**: character
- ▶ **rows** and **columns**
- ▶ **blank** character (space)

The Type of Pictures

```
type width = int;;
type height = int;;
type t = (width ∗ height ∗ Strng.t list);;
```
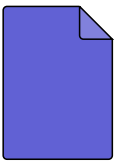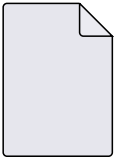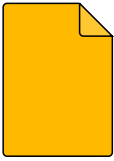
Combining Pictures - Stack Above Each Other


```
let rec above (w1, h1, p1) (w2, h2, p2) =
 if w1 = w2 then
   (w1, h1 + h2, p1 @ p2)
 else
   failwith "Picture.above: different widths"
;;
```

Combining Pictures - Stack Above Each Other (cont'd)

$\vdots$

let stack ps = Lst.fold1 above ps;;

$\vdots$

Fold Lists Containing At Least One Element

Type
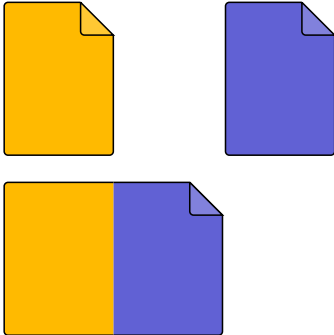Lst.fold1 : ('a $->$ 'a $->$ 'a) $->$ 'a list $->$ 'a

$$\text{Lst.fold1} \circ [x_1; \ldots; x_{n-1}; x_n] = (x_1 \circ (\cdots (x_{n-1} \circ x_n) \cdots))$$

Example
$$\text{fold1 } (+) \ [1;\ 2;\ 3] = \qquad 1 + 2 + 3 \qquad = 6$$
$$\text{fold1 } (\char`\^) \ [\text{"Hello"};\ \text{", World!"}] = \text{"Hello"} \char`\^ \text{", World!"} = \text{"Hello, World!"}$$

Combining Pictures - Spread Side By Side

```
let rec beside (w1, h1, p1) (w2, h2, p2) =
  if h1 = h2 then
    (w1 + w2, h1, Lst.zip2_with (@) p1 p2)
  else
    failwith "Picture.beside: different heights"
;;
```
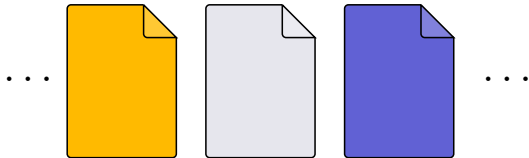
Combine Two Lists Via Function

Type

zip2_with : ('a −> 'b −> 'c) −> 'a list −> 'b list −> 'c list

Lst.zip2_with $\circ$ $[x_1; \ldots; x_m]$ $[y_1; \ldots; y_n] = [x_1 \circ y_1; \ldots; x_{\min\{m,n\}} \circ y_{\min\{m,n\}}]$
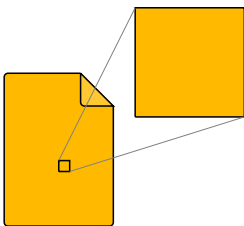
Example

zip2_with ($*$) [1; 2] [3; 4; 5] =   [1 $*$ 3; 2 $*$ 4]   = [3; 8]

zip2_with drop [1; 0] [['a']; ['b']] = [drop 1 ['a']; drop 0 ['b']] = [[]; ['b']]

Combining Pictures - Spread Side By Side (cont'd)

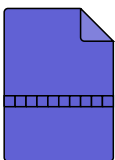**let** spread ps = Lst.fold1 beside ps;;
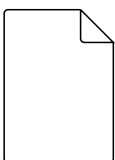
Creating Pictures - Pixels, Rows, and Empty Pictures
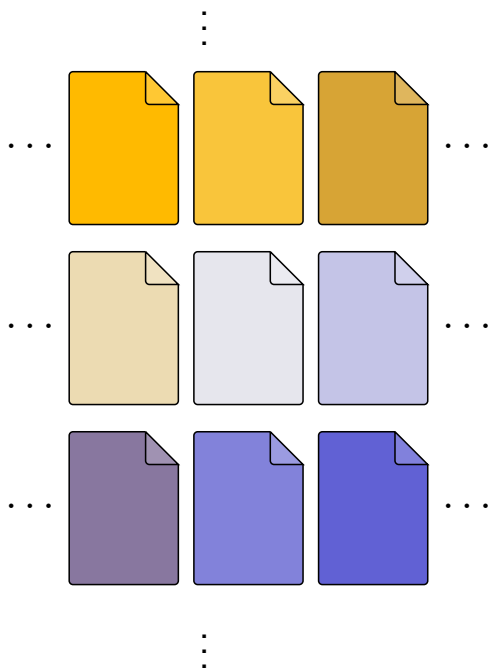
Pixel

let pixel c = (1, 1, [[c]]);;

Row

let row s = spread (Lst.map pixel s);;

Empty

let empty w h =
  let line = Lst.init ' ' w **in**
  let rows = Lst.init line h **in**
  stack (Lst.map row rows)
;;

Combining Pictures - Tile

let tile pss = stack (Lst.map spread pss);;

Margins

Signatures

- stack_with : height −> t list
- spread_with : width −> t list
- tile_with : height −> width −> t list list

Functions

```
let stack_with h ps =
  Lst.fold1 (fun p q −> above (above p (empty (width q) h)) q) ps
;;

let spread_with w ps =
  Lst.fold1 (fun p q −> beside (beside p (empty w (height q))) q) ps
;;

let tile_with w h pss = stack_with h (Lst.map (spread_with w) pss);;
```

Printing Pictures

Idea

- ▶ convert to Strng.t
- ▶ use Strng.print

Realization

- ▶ Picture:

  ```
  let to_strng (_, _, p) = Strng.join ['\n'] p;;
  ```

- ▶ Strng:

  ```
  ;;
  let join s xs =
    Lst.fold1 (fun xs ys -> xs @ s @ ys) xs
  ```