COSC 2041

Sebastiaan Joosten

Lecture Sept 11: Datatypes

University of Minnesota
Driven to Discover®

Outline

- Creating lists
- Using lists
- Recursion over lists

Lists in ocaml

- The empty list [] is a list of type ‘a
- If x is an element of type ‘a and xs is a list of type ‘a, then (x :: xs) is a list of type ‘a
- There are no other lists of type ‘a.

- Example:
  (1::(2::(3::[]))) is a list of type int

- No worries, you can just write [1; 2; 3] too

Lists in ocaml

```
utop # (1::(2::(3::[])));;
- : int list = [1; 2; 3]
utop # [];;
- : 'a list = []
utop # [3;4;5] @ [7;8;9];;
- : int list = [3; 4; 5; 7; 8; 9]
utop # ["hello";"world"];;
- : string list = ["hello"; “world"]
utop # ["hello";"world"] @ [1;2;3];;
Error: This expression has type int but an
     expression was expected of type string
```

Lists are of one type only…

```
utop # ["hello";"world";1;2;3];;
Error: This expression has type int but
an expression was expected of type string
```

Common question: but what if I need to mix multiple types?
Answer 1: cannot do that
Answer 2: we’ll see how to do it later

Lists so far…

- We’ve seen how to make lists using :: or @
- This is pretty similar to other languages
- What about functions that use a list as input?
  - I think ocaml is much nicer here

Match statements

- A match statement looks at the structure of a list. Recall that:
  - The empty list [] is a list of type ‘a
  - **If x is an element of type ‘a and xs is a list of type ‘a, then (x :: xs) is a list of type ‘a**
  - There are no other lists of type ‘a.

```
utop # let my_list = [2;3;4];;
val my_list : int list = [2; 3; 4]
utop # match my_list with
         (x :: xs) -> string_of_int x ^ " and then some"
         | [] -> "empty list";;
- : string = "2 and then some"
```

Match statements

- A match statement looks at the structure of a list. Recall that:
  - **The empty list [] is a list of type ‘a**
  - If x is an element of type ‘a and xs is a list of type ‘a, then (x :: xs) is a list of type ‘a
  - There are no other lists of type ‘a.

```
utop # let my_list = [];;
val my_list : int list = []
utop # match my_list with
         (x :: xs) -> string_of_int x ^ " and then some"
         | [] -> "empty list";;
- : string = "empty list"
```

Match statements: binding variables

```
match my_list with
          (x :: xs) -> string_of_int x
```

- This binds the variables x and xs locally
- … much like “let f x xs = …” would bind x and xs
- of course, since x and xs come from the same list, there is a type constraint
  (if x is int, then xs must be int list)

Recursion over lists

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h :: t -> h + sum t
```

```
val sum : int list -> int = <fun>
```

```
let rec length lst =
  match lst with
  | [] -> 0
  | h :: t -> 1 + length t
```

```
val length : 'a list -> int = <fun>
```

Putting it all together

- Let’s write a function that:
  - takes a list of non-negative numbers and an integer ‘budget’
  - returns the longest prefix of the input list such that its sum does not exceed ‘budget’
- stay_in_budget :
  int list -> int -> int list

Putting it all together

- Let’s write a function that:
  - takes a list of non-negative numbers and an integer ‘budget’
  - returns the longest prefix of the input list such that its sum does not exceed ‘budget’
- stay_in_budget :
  int list -> int -> int list

```
let rec
stay_in_budget lst
budget =
```

Putting it all together

- Let’s write a function that:
  - takes a list of non-negative numbers and an integer ‘budget’
  - returns the longest prefix of the input list such that its sum does not exceed ‘budget’
- stay_in_budget :
  int list -> int -> int list

```
let rec
stay_in_budget lst
budget =
  match lst with
```

Putting it all together

- Let’s write a function that:
  - takes a list of non-negative numbers and a non-negative integer ‘budget’
  - returns the longest prefix of the input list such that its sum does not exceed ‘budget’
- stay_in_budget :
  int list -> int -> int list

```
let rec
stay_in_budget lst
budget =
  match lst with
  | [] ->
  | h::t ->
```

Putting it all together

- Let’s write a function that:
  - takes a list of non-negative numbers and a non-negative integer ‘budget’
  - returns the longest prefix of the input list such that its sum does not exceed ‘budget’
- stay_in_budget :
  int list -> int -> int list

```
let rec
stay_in_budget lst
budget =
  match lst with
  | [] -> []
  | h::t ->
```

Putting it all together

- Let’s write a function that:
  - takes a list of non-negative numbers and a non-negative integer ‘budget’
  - returns the longest prefix of the input list such that its sum does not exceed ‘budget’
- stay_in_budget :
  int list -> int -> int list

```
let rec
stay_in_budget lst
budget =
  match lst with
  | [] -> []
  | h::t -> if
      h <= budget then
                 else
```

Putting it all together

- Let's write a function that:
  – takes a list of non-negative numbers and a non-negative integer 'budget'
  – returns the longest prefix of the input list such that its sum does not exceed 'budget'
- stay_in_budget :
  int list -> int -> int list

```
let rec
stay_in_budget lst
budget =
  match lst with
  | [] -> []
  | h::t -> if
      h <= budget then
                else []
```

Putting it all together

- Let's write a function that:
  - takes a list of non-negative numbers and a non-negative integer 'budget'
  - returns the longest prefix of the input list such that its sum does not exceed 'budget'
- stay_in_budget :
  int list -> int -> int list

```
let rec
stay_in_budget lst
budget =
  match lst with
  | [] -> []
  | h::t -> if
      h <= budget then
    h::(stay_in_budget

          ) else []
```

Putting it all together

- Let’s write a function that:
  - takes a list of non-negative numbers and a non-negative integer ‘budget’
  - returns the longest prefix of the input list such that its sum does not exceed ‘budget’
- stay_in_budget :
  int list -> int -> int list

```
let rec
stay_in_budget lst
budget =
  match lst with
   | [] -> []
   | h::t -> if
       h <= budget then
     h::(stay_in_budget
           t (budget - h)
           ) else []
```

Putting it all together

- Let's write a function that:
  - takes a list of non-negative numbers and a non-negative integer 'budget'
  - returns the longest prefix of the input list such that its sum does not exceed 'budget'
- stay_in_budget :
  int list -> int -> int list

```
let rec
stay_in_budget lst budget =
  match lst with
  | [] -> []
  | h::t -> if
      h <= budget then
    h::stay_in_budget
          t (budget - h)
          else []

utop # stay_in_budget [3;4;5]
8;;
- : int list = [3; 4]
utop # stay_in_budget [3;4;5]
80;;
- : int list = [3; 4; 5]
```

Variants

- We can define our own data-types:

type day = **Sun** | **Mon** | **Tue** | **Wed** | **Thu** | **Fri** | **Sat**

let d = **Tue**

val d : day = Tue

- We can also use those data-types:

```
let is_weekend day =
  match day with
  | Sun | Sat -> true
  | _ -> false
```

Capital letters!

- Note that types, function and variable names start with lower case letters, but ‘Constructors’ start with upper case letters:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
let d = Tue
val d : day = Tue
```

Constructors

```
let is_weekend day =
  match day with
  | Sun | Sat -> true
  | _ -> false
```

Variants

- The syntax for matching variants is very similar to that of lists. This is no accident!
- We'll see how to use Variants to define:
  - lists containing multiple data-types
  - lists themselves!

Quiz time!

Some more recursively generating lists…

```
(** downfrom [n] starts at n and counts down to 1:
    {[downfrom 5 = [5;4;3;2;1] ]} *)
let downfrom n =
```

•

Some more recursively generating lists…

```
(** downfrom [n] starts at n and counts down to 1:
    {[downfrom 5 = [5;4;3;2;1] ]} *)
let downfrom n = if n < 1 then [] else
```

Some more recursively generating lists…

```
(** downfrom [n] starts at n and counts down to 1:
    {[downfrom 5 = [5;4;3;2;1] ]} *)
let downfrom n = if n < 1 then [] else
                 n :: downfrom (n - 1)
```

Some more recursively generating lists…

```
let downfrom n = if n < 1 then [] else
                 n :: downfrom (n - 1)

(** upto [n] gives a list of
  * numbers up to [n] (exclusive):
  * {[ upto 5 = [0;1;2;3;4] ]} *)
let upto n =
```

Some more recursively generating lists…

```
let downfrom n = if n < 1 then [] else
                 n :: downfrom (n - 1)

(** upto [n] gives a list of
  * numbers up to [n] (exclusive):
  * {[ upto 5 = [0;1;2;3;4] ]} *)
let upto n =
    let rec loop i =
      if i > n then []
      else i :: loop (i + 1)
    in loop 0
```

Some more recursion on lists…

```
let rec takeWhilePositive lst =
  match lst with
  | [] -> []
  | hd :: tl ->
  if hd > 0 then hd :: takeWhilePositive tl
  else []
```

Some more recursion on lists…

```
let rec takePositive lst =
  match lst with
  | [] -> []
  | hd :: tl ->
  if hd > 0 then hd :: takePositive tl
  else takePositive tl
```

Some more recursion on lists…

```
let rec dropPositive lst =
  match lst with
  | [] -> []
  | hd :: tl ->
  if hd > 0 then dropPositive tl
  else hd :: dropPositive tl
```

Some more recursion on lists…

```
let rec dropWhilePositive lst =
  match lst with
  | [] -> []
  | hd :: tl ->
  if hd > 0 then tl
  else dropWhilePositive tl
```

Next lecture

`Lists with multiple types in them`