Foundations of Computer Science

Lists of pairs and pairs of lists

Dr. Robert Harle & Dr. Jeremy Yallop

2020–2021

Building a List of Pairs

$$\left.\begin{array}{l}[x_1;\ x_2;\ \ldots;\ x_n;\ ]\\ [y_1;\ y_2;\ \ldots;\ y_n;\ ]\end{array}\right\} \mapsto [(x_1,\ y_1);\ (x_2,\ y_2);\ \ldots;\ (x_n,\ y_n);\ ]$$

```
let rec zip xs ys =
  match xs, ys with
  | (x::xs, y::ys) -> (x, y) :: zip xs ys
  | _ -> []
```

Building a List of Pairs

```
let rec zip xs ys =
  match xs, ys with
  | (x::xs, y::ys) -> (x, y) :: zip xs ys
  | _ -> []
```

The **wildcard pattern** (_) matches anything.

For example, _ will match: ([], (y :: ys))

The patterns are **tested in order**

In this match, _ will not match: (x :: xs, (y :: ys))

```
 In[1]: zip [1;2;3;4] ['a';'b';'c']
Out[1]: - : (int * char) list = [(1,'a'); (2,'b'); (3,'c')]
```

Building a List of Pairs

```
let rec zip xs ys =
  match xs, ys with
  | (x::xs, y::ys) -> (x, y) :: zip xs ys
  | _ -> []
```

The **wildcard pattern** (_) matches anything.

For example, _ will match: ([], (y :: ys))

The patterns are **tested in order**

In this match, _ will not match: (x :: xs, (y :: ys))

```
In[1]: zip [1;2;3;4] ['a';'b';'c']

Out[1]: - : (int * char) list = [(1,'a'); (2,'b'); (3,'c')]
```

Building a List of Pairs

```
let rec zip xs ys =
  match xs, ys with
  | (x::xs, y::ys) -> (x, y) :: zip xs ys
  | _ -> []
```

The **wildcard pattern** (_) matches anything.

For example, _ will match: ([], (y :: ys))

The patterns are **tested in order**

In this match, _ will not match: (x :: xs, (y :: ys))

```
 In[1]: zip [1;2;3;4] ['a';'b';'c']
Out[1]: - : (int * char) list = [(1,'a'); (2,'b'); (3,'c')]
```

Building a List of Pairs

The `zip` function builds a list-of-pairs from two lists

```
val zip : 'a list -> 'b list -> ('a * 'b) list
```

The `unzip` function builds a pair-of-lists from a list-of-pairs

```
val unzip : ('a * 'b) list -> ('a list * 'b list)
```

Syntax: Declarations and Local Bindings

`let` in **declarations** (familiar)

```
let p = e
```

`let` in **expressions** (new)

```
let p = e1 in e2
```

Binds the value of e1 to p within expression e2

Useful within a function

Can perform intermediate computations with function arguments

Building a Pair of Results

Defining unzip with a **local binding**:

```
 In[2]: let rec unzip = function   local binding
        | [] -> ([], [])
        | (x, y)::ps -> let xs, ys = unzip ps in
             expression → (x::xs, y::ys)

Out[2]: val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>

 In[3]: unzip [(1,'a');(2,'b')]

Out[3]: - : int list * char list
          = ([1; 2], ['a'; 'b'])
```

The `let` construct binds `xs` and `ys` to the results of the recursive call.

Building a Pair of Results

Defining unzip with a **local binding**:

```
In[2]:  let rec unzip = function
        | [] -> ([], [])
        | (x, y)::ps -> let xs, ys = unzip ps in
                        (x::xs, y::ys)

Out[2]: val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>

In[3]:  unzip [(1,'a');(2,'b')]

Out[3]: - : int list * char list
          = ([1; 2], ['a'; 'b'])
```

local binding

expression →

The let construct binds xs and ys to the results of the recursive call.

Building a Pair of Results

Defining unzip with a **local binding**:

```
 In[2]:  let rec unzip = function  local binding
         | [] -> ([], [])
         | (x, y)::ps -> let xs, ys = unzip ps in
               expression → (x::xs, y::ys)

Out[2]:  val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>

 In[3]:  unzip [(1,'a');(2,'b')]

Out[3]:  - : int list * char list
           = ([1; 2], ['a'; 'b'])
```

The `let` construct binds `xs` and `ys` to the results of the recursive call.

Building a Pair of Results

Defining unzip with a **local binding**:

```
 In[2]:  let rec unzip = function
         | [] -> ([], [])
         | (x, y)::ps -> let xs, ys = unzip ps in
                         (x::xs, y::ys)

Out[2]:  val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>

 In[3]:  unzip [(1,'a');(2,'b')]

Out[3]:  - : int list * char list
           = ([1; 2], ['a'; 'b'])
```

local binding

expression →

The let construct binds xs and ys to the results of the recursive call.

Building a Pair of Results

Defining unzip with a **local binding**:

```
 In[2]: let rec unzip = function  local binding
        | [] -> ([], [])
        | (x, y)::ps -> let xs, ys = unzip ps in
              expression → (x::xs, y::ys)

Out[2]: val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>

 In[3]: unzip [(1,'a');(2,'b')]

Out[3]: - : int list * char list
          = ([1; 2], ['a'; 'b'])
```

The `let` construct binds `xs` and `ys` to the results of the recursive call.

Building a Pair of Results

Defining unzip with an **auxiliary function**:

```
let conspair ((x, y), (xs, ys)) = (x::xs, y::ys)


let rec unzip = function
| [] -> ([], [])
| xy :: pairs -> conspair (xy, unzip pairs)
```

Building a Pair of Results

Defining unzip with an **auxiliary function**:

```
let conspair ((x, y), (xs, ys)) = (x::xs, y::ys)


let rec unzip = function
| [] -> ([], [])
| xy :: pairs -> conspair (xy, unzip pairs)
```

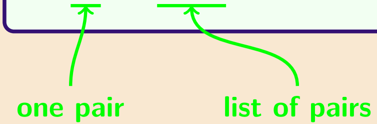
one pair

Building a Pair of Results

Defining unzip with an **auxiliary function**:

```
let conspair ((x, y), (xs, ys)) = (x::xs, y::ys)


let rec unzip = function
| [] -> ([], [])
| xy :: pairs -> conspair (xy, unzip pairs)
```



one pair

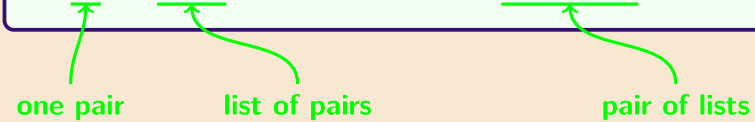
list of pairs

Building a Pair of Results

Defining unzip with an **auxiliary function**:

```
let conspair ((x, y), (xs, ys)) = (x::xs, y::ys)


let rec unzip = function
| [] -> ([], [])
| xy :: pairs -> conspair (xy, unzip pairs)
```



one pair

list of pairs

pair of lists

Building a Pair of Results

Defining unzip with an **accumulator**:

```
let rec revUnzip = function
| ([], xs, ys) -> (xs, ys)
| ((x, y)::ps, xs, ys) -> revUnzip (ps, x::xs, y::ys)
```

Question: How to call revUnzip?

```
revUnzip (pairs, [], [])
```

Question: What's the result of the following?

```
 In[4]: let pairs = [("a", 1); ("b", 2)];;
        revUnzip (pairs, [], [])

Out[4]: - : string list * int list
          = (["b"; "a"], [2; 1])
```

Building a Pair of Results

Defining unzip with an **accumulator**:

```
let rec revUnzip = function
| ([], xs, ys) -> (xs, ys)
| ((x, y)::ps, xs, ys) -> revUnzip (ps, x::xs, y::ys)
```

Question: How to call revUnzip?

```
revUnzip (pairs, [], [])
```

Question: What's the result of the following?

```
In[4]:  let pairs = [("a", 1); ("b", 2)];;
        revUnzip (pairs, [], [])

Out[4]: - : string list * int list
          = (["b"; "a"], [2; 1])
```

Building a Pair of Results

Defining unzip with an **accumulator**:

```
let rec revUnzip = function
| ([], xs, ys) -> (xs, ys)
| ((x, y)::ps, xs, ys) -> revUnzip (ps, x::xs, y::ys)
```

Question: How to call revUnzip?

```
revUnzip (pairs, [], [])
```

Question: What's the result of the following?

```
 In[4]: let pairs = [("a", 1); ("b", 2)];;
        revUnzip (pairs, [], [])

Out[4]: - : string list * int list
          = (["b"; "a"], [2; 1])
```