OCaml, Batteries Included
OCaml Meeting 2009

David Teller\textsuperscript{1}, Edgar Friendly, Stefano Zacchiroli\textsuperscript{2}, Gabriel Scherer, Jérémie Dimino . . .

\textsuperscript{1}LIFO, Université d’Orléans ⇒ MLState
\textsuperscript{2}PPS, Université Paris 7

February 4th, 2009
Before we start
Introduction

Manipulating data
  From loops to enumerations
  Other data structures

I/O
  From Channels to I/O
  Composability
  Extensibility

Text
  From string to...
  Features

Conclusion
What makes a good language?

Safety
Expressivity
Composability
Syntax
Simplicity
Fun factor
## What makes a good language?

- **Safety**
- **Expressivity**
- **Composability**
- **Syntax**
- **Simplicity**
- **Fun factor**

- OCaml

### OCaml, Batteries Included

David Teller et al.

### Introduction

- **Manipulating data**
  - From loops to enumerations
- **Other data structures**
- **I/O**
  - From Channels to I/O
  - Composability
  - Extensibility
- **Text**
  - From string to...
  - Features
- **Conclusion**
What makes a good language?

- Safety
  - OCaml
  - √
- Expressivity
  - OCaml
  - √
- Composability
- Syntax
- Simplicity
- Fun factor
What makes a good language?

- Safety: OCaml ✓
- Expressivity: OCaml ✓
- Composability: OCaml ✓
- Syntax
- Simplicity
- Fun factor
What makes a good language?

- Safety: OCaml ✓
- Expressivity: OCaml ✓
- Composability: OCaml ✓
- Syntax: OCaml ✓
- Simplicity
- Fun factor
What makes a good language?

- Safety: OCaml
- Expressivity: OCaml
- Composability: OCaml
- Syntax: OCaml
- Simplicity: OCaml
- Fun factor
What makes a good language?

- Safety  OCaml  ✓
- Expressivity  OCaml  ✓
- Composability  OCaml  ✓
- Syntax  OCaml  ✓
- Simplicity  OCaml  ✓
- Fun factor  OCaml  ✓
Popular languages

What about:

Java
C#
VB
Python
JS
## Popular languages

<table>
<thead>
<tr>
<th>What about:</th>
<th>Safe</th>
<th>Ok</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
<td>ok</td>
<td></td>
</tr>
<tr>
<td>C#</td>
<td></td>
<td>ok</td>
<td></td>
</tr>
<tr>
<td>VB</td>
<td></td>
<td>bad</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td>bad</td>
<td></td>
</tr>
<tr>
<td>JS</td>
<td></td>
<td>bad</td>
<td></td>
</tr>
</tbody>
</table>
## Popular languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Safe</th>
<th>Expressive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>ok</td>
<td>ugly</td>
</tr>
<tr>
<td>C#</td>
<td>ok</td>
<td>bad → ok</td>
</tr>
<tr>
<td>VB</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Python</td>
<td>bad</td>
<td>ok</td>
</tr>
<tr>
<td>JS</td>
<td>bad</td>
<td>bad → ok</td>
</tr>
</tbody>
</table>
## Popular languages

<table>
<thead>
<tr>
<th></th>
<th>Safe</th>
<th>Expressive</th>
<th>Composable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>ok</td>
<td>ugly</td>
<td>bad</td>
</tr>
<tr>
<td>C#</td>
<td>ok</td>
<td>bad → ok</td>
<td>ok</td>
</tr>
<tr>
<td>VB</td>
<td>bad</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Python</td>
<td>bad</td>
<td>ok</td>
<td>good</td>
</tr>
<tr>
<td>JS</td>
<td>bad</td>
<td>bad → ok</td>
<td>ok</td>
</tr>
<tr>
<td>Language</td>
<td>Safe</td>
<td>Expressive</td>
<td>Composable</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>Java</td>
<td>ok</td>
<td>ugly</td>
<td>bad</td>
</tr>
<tr>
<td>C#</td>
<td>ok</td>
<td>bad → ok</td>
<td>ok</td>
</tr>
<tr>
<td>VB</td>
<td>bad</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Python</td>
<td>bad</td>
<td>ok</td>
<td>good</td>
</tr>
<tr>
<td>JS</td>
<td>bad</td>
<td>bad → ok</td>
<td>ok</td>
</tr>
</tbody>
</table>
## Popular languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Safe</th>
<th>Expressive</th>
<th>Composable</th>
<th>Syntax</th>
<th>Fun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>ok</td>
<td>ugly</td>
<td>bad</td>
<td>ugly</td>
<td>ugly</td>
</tr>
<tr>
<td>C#</td>
<td>ok</td>
<td>bad → ok</td>
<td>ok</td>
<td>bad</td>
<td>ok</td>
</tr>
<tr>
<td>VB</td>
<td>bad</td>
<td>bad</td>
<td>bad</td>
<td>ugly</td>
<td>ugly</td>
</tr>
<tr>
<td>Python</td>
<td>bad</td>
<td>ok</td>
<td>good</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>JS</td>
<td>bad</td>
<td>bad → ok</td>
<td>ok</td>
<td>bad</td>
<td>good</td>
</tr>
</tbody>
</table>
## Popular languages

<table>
<thead>
<tr>
<th></th>
<th>Safe</th>
<th>Expressive</th>
<th>Composable</th>
<th>Syntax</th>
<th>Fun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>ok</td>
<td>ugly</td>
<td>bad</td>
<td>ugly</td>
<td>ugly</td>
</tr>
<tr>
<td>C#</td>
<td>ok</td>
<td>bad → ok</td>
<td>ok</td>
<td>bad</td>
<td>ok</td>
</tr>
<tr>
<td>VB</td>
<td>bad</td>
<td>bad</td>
<td>bad</td>
<td>ugly</td>
<td>ugly</td>
</tr>
<tr>
<td>Python</td>
<td>bad</td>
<td>ok</td>
<td>good</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>JS</td>
<td>bad</td>
<td>bad → ok</td>
<td>ok</td>
<td>bad</td>
<td>good</td>
</tr>
</tbody>
</table>

Good ⇔ Popular
Popular languages

What about:

<table>
<thead>
<tr>
<th>Language</th>
<th>Safe</th>
<th>Expressive</th>
<th>Composable</th>
<th>Syntax</th>
<th>Fun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>ok</td>
<td>ugly</td>
<td>bad</td>
<td>ugly</td>
<td>ugly</td>
</tr>
<tr>
<td>C#</td>
<td>ok</td>
<td>bad → ok</td>
<td>ok</td>
<td>bad</td>
<td>ok</td>
</tr>
<tr>
<td>VB</td>
<td>bad</td>
<td>bad</td>
<td>bad</td>
<td>ugly</td>
<td>ugly</td>
</tr>
<tr>
<td>Python</td>
<td>bad</td>
<td>ok</td>
<td>good</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>JS</td>
<td>bad</td>
<td>bad → ok</td>
<td>ok</td>
<td>bad</td>
<td>good</td>
</tr>
</tbody>
</table>

Good ≟ Popular
OCaml ∉ Popular (yet)
What makes a successful language?

Maybe something like:

Well-suited library (sometimes the only available library)
What makes a successful language?

Maybe something like:

**Well-suited library** (sometimes the only available library)

**Consistent/composable library** (only one string type, only one iteration type, only one exception hierarchy...)

**Extensibility** (new kinds of streams may be created, virtual file system...)

**Tutorials** (which should be trivial to find)
What makes a successful language?

Maybe something like:

Well-suited library (sometimes the only available library)

Consistent/composable library (only one string type, only one iteration type, only one exception hierarchy...)

Extensibility (new kinds of streams may be created, virtual file system...)

Tutorials (which should be trivial to find)

either

Fun factor

or

Public relations (either a company or open-source buzz)
And in OCaml?

OCaml, Batteries Included
David Teller et al.

Introduction
Manipulating data
From loops to enumerations
Other data structures
I/O
From Channels to I/O
Composability
Extensibility
Text
From string to...
Features
Conclusion
And in OCaml?

Well-suited library Low-level library in a high-level language. Minimal library sufficient for testing, not necessarily for development.
And in OCaml?

**Well-suited library**  Low-level library in a high-level language. Minimal library sufficient for testing, not necessarily for development.

**Consistent/composable library**  How do I convert a stream to a list? How do I map a stream or an I/O or a hashtable? How should I represent Unicode strings?
And in OCaml?

**Well-suited library**  Low-level library in a high-level language. Minimal library sufficient for testing, not necessarily for development.

**Consistent/composable library**  How do I convert a stream to a list? How do I map a stream or an I/O or a hashtable? How should I represent Unicode strings?

**Extensibility**  Camlp4/Camlp5 ok. But how do I add new kinds of I/O? How do I create new streams?
And in OCaml?

**Well-suited library**  Low-level library in a high-level language. Minimal library sufficient for testing, not necessarily for development.

**Consistent/composable library**  How do I convert a stream to a list? How do I map a stream or an I/O or a hashtable? How should I represent Unicode strings?

**Extensibility**  Camlp4/Camlp5 ok. But how do I add new kinds of I/O? How do I create new streams?

**Tutorials**  Whole-language tutorials ok. Task-specific languages not so ok. Plus newbies need to find them.
And in OCaml?

Well-suited library  Low-level library in a high-level language. Minimal library sufficient for testing, not necessarily for development.

Consistent/composable library  How do I convert a stream to a list? How do I map a stream or an I/O or a hashtable? How should I represent Unicode strings?

Extensibility  Camlp4/Camlp5 ok. But how do I add new kinds of I/O? How do I create new streams?

Tutorials  Whole-language tutorials ok. Task-specific languages not so ok. Plus newbies need to find them.

Fun factor  Oh, yeah. Despite competition with Haskell.
And in OCaml?

Well-suited library  Low-level library in a high-level language. Minimal library sufficient for testing, not necessarily for development.

Consistent/composable library  How do I convert a stream to a list? How do I map a stream or an I/O or a hashtable? How should I represent Unicode strings?

Extensibility  Camlp4/Camlp5 ok. But how do I add new kinds of I/O? How do I create new streams?

Tutorials  Whole-language tutorials ok. Task-specific languages not so ok. Plus newbies need to find them.

Fun factor  Oh, yeah. Despite competition with Haskell.

Public relations  Insufficient. Despite competition with Haskell.
What can we improve?

Well-suited library  Build a high-level library.
What can we improve?

Well-suited library  Build a high-level library.
Consistent/composable library  Add base abstractions & data structures, uniformize interfaces.
What can we improve?

**Well-suited library**  Build a high-level library.

**Consistent/composable library**  Add base abstractions & data structures, uniformize interfaces.

**Extensibility**  Liberate the base data structures!
What can we improve?

Well-suited library  Build a high-level library.
Consistent/composable library  Add base abstractions & data structures, uniformize interfaces.
Extensibility  Liberate the base data structures!
Tutorials  Improve documentation.
What can we improve?

Well-suited library  Build a high-level library.
Consistent/composable library  Add base abstractions & data structures, uniformize interfaces.
Extensibility  Liberate the base data structures!
Tutorials  Improve documentation.
Fun factor  Can always be improved. Cabal?
What can we improve?

Well-suited library  Build a high-level library.
Consistent/composable library  Add base abstractions & data structures, uniformize interfaces.
Extensibility  Liberate the base data structures!
Tutorials  Improve documentation.
Fun factor  Can always be improved. Cabal?
Public relations  OCaml Developer Days, OCamlCore.org, books, teaching, etc.
Introducing

OCaml Batteries Included

Manipulating data
From loops to enumeration
Other data structures

I/O
From Channels to I/O
Composability
Extensibility

Text
From string to...
Features

Conclusion
Our objectives

A distribution of OCaml with

- Newbie-oriented documentation.
- More comfortable syntax.
- Consistent and high-level API.
- Extensible data structures.
Our objectives

A distribution of OCaml with

- Newbie-oriented documentation.
- More comfortable syntax.
- Consistent and high-level API.
- Extensible data structures.
- More fun!
How?

**API** Existing libraries + uniformization “glue layer”.

**Language** Syntax extensions, auto-loaded.

**Toolchain** Existing tools + transparent shell scripts.

**Documentation** Largely rewritten + new doclet.

**How?**

**API** Existing libraries + uniformization “glue layer”.

**Language** Syntax extensions, auto-loaded.

**Toolchain** Existing tools + transparent shell scripts.

**Documentation** Largely rewritten + new doclet.

**How?**

**API** Existing libraries + uniformization “glue layer”.

**Language** Syntax extensions, auto-loaded.

**Toolchain** Existing tools + transparent shell scripts.

**Documentation** Largely rewritten + new doclet.
How?

**API** Existing libraries + uniformization “glue layer”.

**Language** Syntax extensions, auto-loaded.

**Toolchain** Existing tools + transparent shell scripts.

**Documentation** Largely rewritten + new doclet.

Improve the user experience, don’t reinvent the wheel!

Don’t turn OCaml into Java!
How?

API   Existing libraries + uniformization “glue layer”.
Language Syntax extensions, auto-loaded.
Toolchain Existing tools + transparent shell scripts.
Documentation Largely rewritten + new doclet.

Improve the user experience, don’t reinvent the wheel!
Don’t turn OCaml into Java!
Built on top of the Base library and ExtLib.
Introduction

**Manipulating data**
- From loops to enumerations
- Other data structures

**I/O**
- From Channels to I/O
- Composability
- Extensibility

**Text**
- From string to...
- Features

Conclusion
Step 1

**Objective** Simplify and uniformize data structure access.

**Objective** Decrease need for multi-paradigm for simple tasks.
What is this for?

OCaml has

built-in specialized loops for, while
data structure-based loops List.iter, List.fold_left, List.fold_right, List.map...
built-in general loops let rec
OCaml has

built-in specialized loops for, while (very specialized, very optimized)

data structure-based loops List.iter, List.fold_left, List.fold_right, List.map...

built-in general loops let rec
What is this for?

OCaml has

built-in specialized loops for, while (very specialized, very optimized)

data structure-based loops List.iter, List.fold_left, List.fold_right, List.map... (requires data structure, not homogeneous among structures)

built-in general loops let rec
What is this for?

OCaml has

built-in specialized loops for, while (very specialized, very optimized)

data structure-based loops List.iter, List.fold_left, List.fold_right, List.map…(requires data structure, not homogeneous among structures)

built-in general loops let rec (general mechanism for implementing loops)
OCaml has

**built-in specialized loops** for, while (very specialized, very optimized)

**data structure-based loops** List.iter, List.fold_left, List.fold_right, List.map... (requires data structure, not homogeneous among structures)

**built-in general loops** let rec (general mechanism for implementing loops)

- Specialized loops are optimizations.
- let rec is (among other things) an extension mechanism.
Overview Lightweight iterators, aka “like Stream.t, but open”.
Enter enumerations

Overview Lightweight iterators, aka “like Stream.t, but open”.

Operations foreach/iter, map, fold, scanl, filter, flatten
Enter enumerations

Overview Lightweight iterators, aka “like Stream.t, but open”.

Operations foreach/iter, map, fold, scanl, filter, flatten

Conversion List.enum/List.backwards/
Array.enum/Array.backwards/
Hashtbl.enum/Hashtbl.keys/Hashtbl.values/
String.enum/String.backwards/...

Construction ( - ), ( -- ), ( ~~~ ), unfold, etc.

Source ExtLib, SDFlow, new stuff
Examples

Exercise Count from 1 to 10.
Examples

Exercise  Count from 1 to 10.
1 -- 10
Examples

**Exercise** Count from 1 to 10.
1 -- 10

**Exercise** Print all elements between 1 to 10.
Examples

Exercise Count from 1 to 10.
1 -- 10

Exercise Print all elements between 1 to 10.
let print_intln x =
    print_int x;
    print_newline ()

foreach (1 -- 10) print_intln
Examples (2)

**Exercise** Print the square numbers of all odd numbers between 1 and 100, by decreasing order.
Examples (2)

**Exercise** Print the square numbers of all odd numbers between 1 and 100, by decreasing order.

```ocaml
let square x = x * x
and odd x = x mod 2 = 1
in
foreach (map square (100 --- 1) // odd)
    print_intln
```
Examples (2)

**Exercise** Print the square numbers of all odd numbers between 1 and 100, by decreasing order.

```ocaml
let square x = x * x
and odd x = x mod 2 = 1
in
foreach (map square (100 -- 1) // odd)
  print_intln
```

Did I mention syntax extensions?

```ocaml
foreach [? x*x | x <- (100 -- 1); x mod 2 = 1]
  print_intln
```
Exercise Keep only the vowels of “OCaml is too cool for school”.

```ocaml
let to_oool = "OCaml is too cool for school"

let vowels = function 'a' | 'e' | 'i' | 'o' | 'u' | 'A' | 'E' | 'I' | 'O' | 'U' - > true
| _ - > false

let vowel x = function 'a' | 'e' | 'i' | 'o' | 'u' | 'A' | 'E' | 'I' | 'O' | 'U' - > true
| _ - > false

let examples = String.of_enum (String.enum to_oool) /
      function 'a' | 'e' | 'i' | 'o' | 'u' | 'A' | 'E' | 'I' | 'O' | 'U' - > true
| _ - > false

let examples2 = String.of_enum (String.enum to_oool) /
      (fun ('a' | 'e' | 'i' | 'o' | 'u' | 'A' | 'E' | 'I' | 'O' | 'U') -> true)
```

Syntax extensions, again:

```ocaml
let examples3 = String.of_enum (String.enum to_oool) /
      fun ('a' | 'e' | 'i' | 'o' | 'u' | 'A' | 'E' | 'I' | 'O' | 'U') -> true
```

```
let examples4 = String.of_enum (String.enum to_oool) /
      fun ('a' | 'e' | 'i' | 'o' | 'u' | 'A' | 'E' | 'I' | 'O' | 'U') -> true
```

Exercise Keep only the vowels of “OCaml is too cool for school”.

```ocaml
define too_cool = "OCaml is too cool for school" in
    String.of_enum(
        (String.enum too_cool) //
        (function 'a'|'e'|'i'|'o'|'u'
            |'A'|'E'|'I'|'O'|'U'  -> true
            | _  -> false)
```
Examples (3)

Exercise  Keep only the vowels of “OCaml is too cool for school”.

```ocaml
define too_cool = "OCaml is too cool for school" in
String.of_enum(
  (String.enum too_cool) //
  (function 'a'|'e'|'i'|'o'|'u'
    | 'A'|'E'|'I'|'O'|'U' -> true
    | _ -> false))
```

Syntax extensions, again:

```ocaml
[? String : x | x <- String : too_cool ; vowel x]
where vowel = function 'a'|'e'|'i'|'o'|'u'
    | 'A'|'E'|'I'|'O'|'U' -> true
    | _ -> false
```
Other data structures

- Doubly-linked lists, defunctorized polymorphic maps, multi-maps, dynamic arrays, lazy lists, defunctorized polymorphic sets, ...
Other data structures

- Doubly-linked lists, defunctorized polymorphic maps, multi-maps, dynamic arrays, lazy lists, defunctorized polymorphic sets, ...
- Upgraded lists, arrays, big arrays, hashtables, queues, stacks, maps, sets.
Other data structures

- Doubly-linked lists, defunctored polymorphic maps, multi-maps, dynamic arrays, lazy lists, defunctored polymorphic sets, ...
- Upgraded lists, arrays, big arrays, hashtables, queues, stacks, maps, sets.
- Everything supports Sexplib, printing, enumerations, etc.
Other data structures

- Doubly-linked lists, defunctorized polymorphic maps, multi-maps, dynamic arrays, lazy lists, defunctorized polymorphic sets, ...
- Upgraded lists, arrays, big arrays, hashtables, queues, stacks, maps, sets.
- Everything supports Sexplib, printing, enumerations, etc.
- Most things support comprehension.
Introduction

Manipulating data
   From loops to enumerations
   Other data structures

I/O
   From Channels to I/O
   Composability
   Extensibility

Text
   From string to...
   Features

Conclusion
Channels are closed

Jan 30 00:50:25 <sanguinev> Is there a way to make an output_channel that just accepts output and doesn’t do anything?
Jan 30 00:51:23 <brendan> open_out "'/dev/null" ?
Jan 30 00:54:00 <sanguinev> brendan: I am looking for something that won’t require a file/any specified location.
Jan 30 00:54:47 <Yoric[DT]> Shameless plug: with Batteries, it’s possible.
Jan 30 00:55:14 <Yoric[DT]> (other than that, you could trick it with a pipe, but that’s much more complicated than /dev/null)
Jan 30 00:55:50 <sanguinev> Yoric[DT]: I am hoping for something nice and system agnostic so I can run the same code on linux/unix/mac OSx/cygwin...

...
Channels are closed

Is there a way to make an output_channel that just accepts output and doesn’t do anything?

open_out "\dev/null" ?

brendan: I am looking for something that won’t require a file/any specified location.

Shameless plug: with Batteries, it’s possible.

(Other than that, you could trick it with a pipe, but that’s much more complicated than /dev/null)

Yoric[DT]: I am hoping for something nice and system agnostic so I can run the same code on linux/unix/mac 05x/cygwin...

Also, can’t filter/map/... on channels.
Channels are closed

Jan 30 00:50:25 <sanguinev> Is there a way to make an output_channel that just accepts output and doesn’t do anything?
Jan 30 00:51:23 <brendan> open_out "/dev/null" ?
Jan 30 00:54:00 <sanguinev> brendan: I am looking for something that won’t require a file/any specified location.
Jan 30 00:54:47 <Yoric [DT]> Shameless plug: with Batteries, it’s possible.
Jan 30 00:55:14 <Yoric [DT]> (other than that, you could trick it with a pipe, but that’s much more complicated than /dev/null)
Jan 30 00:55:50 <sanguinev> Yoric [DT]: I am hoping for something nice and system agnostic so I can run the same code on linux/unix/mac OSx/cygwin...

... Also, can’t filter/map/... on channels. Shameless plug #2: channel #ocaml is open, though.
What’s going on?

OCaml’s `in_channel`/`out_channel` are

- operating system-level
- tied to the Unix model
- closed.
Overview  Drop-in replacement for in_channel/out_channel operations.
Enter input/output

Overview  Drop-in replacement for in_channel/out_channel operations.
Operations  All the usual operations, plus i/o filters, position, callbacks, Unicode, auto-flushing...
Overview  Drop-in replacement for `in_channel/out_channel` operations.

Operations  All the usual operations, plus i/o filters, position, callbacks, Unicode, auto-flushing.

Conversion  To/from enumerations, strings, file names, sockets, processes.

Construction  `File.open_in/open_out, wrap_in/wrap_out`.

Source  ExtLib, OCamlNet, Camomile, more stuff.
Let's do it with Batteries

Exercise Let's implement `cat` with Batteries.
Let's do it with Batteries

Exercise Let's implement `cat` with Batteries.

```ocaml
open IO, File
foreach (args ())
  (fun s -> copy (open_in s) stdout)
```
Let’s do it with Batteries

**Exercise** Let’s implement `cat` with Batteries.

```ocaml
open IO, File
foreach (args ())
  (fun s -> copy (open_in s) stdout)

or

foreach (args ()) **>
  fun s -> copy (open_in s) stdout
```
Let’s do it with Batteries (2)

**Exercise** Now, let’s implement a version of `cat` which prints line numbers along with line contents.
Exercise Now, let’s implement a version of `cat` which prints line numbers along with line contents.

```ocaml
open IO
foreach (args ()) (fun s ->
    Enum.iteri
    (Printf.printf "%d %s\n")
    (File.lines_of s)
)
```
Let’s do it with Batteries (2)

Exercise Now, let’s implement a version of `cat` which prints line numbers along with line contents.

```ocaml
open IO

let print_lines = fun s ->
  Enum.iteri
  (Printf.printf "%d %s\n"
   (File.lines_of s))

In this last version, a file was automatically opened, read (lazily), split into lines and closed.
```
Going further

Exercise Add gzip-decompression.
Going further

**Exercise** Add gzip-decompression.

```ocaml
open IO
foreach (args ()) (fun s ->
  Enum.iteri
    (Printf.printf "%d %s\n")
    (lines_of (GZip.uncompress (File.open_in s)))
)
```
Going further

**Exercise** Add gzip-decompression.

```ocaml
open IO

foreach (args ())(fun s ->
  Enum.iteri
    (Printf.printf "%d %s\n")
    (lines_of (GZip.uncompress (File.open_in s)))
)
```

**Exercise** Count number of bytes read.

```ocaml
foreach (args ())(fun s ->
  let (inp, pos) = pos_in (File.open_in s) in
  Enum.iteri
    (Printf.printf "%d %s\n")
    (lines_of (GZip.uncompress inp));

  Printf.printf "Read %d bytes\n" (pos ())
)
```
Exercise  Add gzip-decompression.

open IO
foreach (args ()) (fun s ->
    Enum.iteri
    (Printf.printf "%d %s\n")
    (lines_of (GZip.uncompress (File.open_in s)))
)

Exercise  Count number of bytes read.

foreach (args ()) (fun s ->
    let (inp, pos) = pos_in (File.open_in s) in
    Enum.iteri
    (Printf.printf "%d %s\n")
    (lines_of (GZip.uncompress inp));
    Printf.printf "\tRead %d bytes\n" (pos ())
)
Going further

**Exercise** Add gzip-decompression.

```ocaml
open IO
foreach (args ()) (fun s ->
  Enum.iteri
    (Printf.printf "%d %s\n")
    (lines_of (GZip.uncompress (File.open_in s)))
)
```

**Exercise** Count number of bytes read.

```ocaml
foreach (args ()) (fun s ->
  let (inp, pos) = pos_in (File.open_in s) in
  Enum.iteri
    (Printf.printf "%d %s\n")
    (lines_of (GZip.uncompress inp));
  Printf.printf "\tRead %d bytes\n" (pos ())
)
```

etc.
I/O is open

Want to read from a string, a socket, an http connexion, etc?
I/O is open

Want to read from a string, a socket, an http connexion, etc? Writing new inputs/outputs is easy.

```ocaml
val create_in:
    read:(unit -> char) ->
    input:(string -> int -> int -> int) ->
    close:(unit -> unit) ->
    input

val wrap_in:
    read:(unit -> char) ->
    input:(string -> int -> int -> int) ->
    close:(unit -> unit) ->
    underlying:(input list) ->
    input
```
Introduction

Manipulating data
  - From loops to enumerations
  - Other data structures

I/O
  - From Channels to I/O
  - Composability
  - Extensibility

Text
  - From string to...
  - Features

Conclusion
The problem with strings

Strings are mutable, hence:

- difficult to trust
- slow to append.
The problem with strings

Strings are mutable, hence:
  - difficult to trust
  - slow to append.

Strings are arrays of `char`, hence:
  - confuse `characters` and `bytes`
  - have no clear notion of encoding.
Introducing ropes

r"This is a UTF-8 encoded rope"

Overview  Functional UTF-8 encoded text with $O(\ln(n))$ concatenation but slower get.

Limitations  About 700Mb in 32-bit, about 220Gb in 64-bit.
Introducing ropes

r"This is a UTF-8 encoded rope"

Overview  Functional UTF-8 encoded text with $O(ln(n))$ concatenation but slower get.

Limitations  About 700Mb in 32-bit, about 220Gb in 64-bit.

Operations  All the operations of String except mutability.
Introducing ropes

"This is a UTF-8 encoded rope"

Overview  Functional UTF-8 encoded text with $O(ln(n))$ concatenation but slower get.

Limitations  About 700Mb in 32-bit, about 220Gb in 64-bit.

Operations  All the operations of String except mutability.

Conversion  Rope.of_latin1, Rope.of uchar, ...
Introducing ropes

r"This is a UTF-8 encoded rope"

Overview  Functional UTF-8 encoded text with $O(ln(n))$ concatenation but slower get.

Limitations  About 700Mb in 32-bit, about 220Gb in 64-bit.

Operations  All the operations of String except mutability.

Conversion  Rope.of_latin1, Rope.of_uchar, ...

Notes  Allocation optimized (with Camlp4!), immutable implementation.
Introducing string with capabilities

\texttt{ro"... a read-only Latin-1 string";;}
\texttt{wo"... a write-only Latin-1 string";;}
\texttt{rw"... a read-write Latin-1 string";;}

**Overview** Your usual strings, but with phantom types to ensure read-only/write-only/read-write capability.
Introducing string with capabilities

ro"... a read-only Latin-1 string";;
wo"... a write-only Latin-1 string";;
rw"... a read-write Latin-1 string";;

Overview Your usual strings, but with phantom types to ensure read-only/write-only/read-write capability.

Operations All the operations of String.
Introducing string with capabilities

ro"... a read-only Latin-1 string";;
wo"... a write-only Latin-1 string";;
rw"... a read-write Latin-1 string";;

Overview  Your usual strings, but with phantom types to ensure read-only/write-only/read-write capability.

Operations  All the operations of String.

Notes  Optimized allocation for read-only strings.
Exercise From a string s, return the first 5 characters, skip the next 3, then return the next 5 characters, the next 5 characters and the rest of the string.
Exercise From a string \( s \), return the first 5 characters, skip the next 3, then return the next 5 characters, the next 5 characters and the rest of the string.

```ocaml
let hairsplit s =
  open String in
let e = enum s in
  [? List : of_enum (f e) | f <- List :
    [take 5; skip 3 | - take 5; take 5; identity]]
```

Exercise Same thing, but with Unicode.
**Exercise**

From a string `s`, return the first 5 characters, skip the next 3, then return the next 5 characters, the next 5 characters and the rest of the string.

```ocaml
let hairsplit s =
  open String in
let e = enum s in
  [? List : of_enum (f e) | f <- List :
    [take 5; skip 3 | - take 5; take 5; identity]]
```

**Exercise** Same thing, but with Unicode.

```ocaml
let hairsplit s =
  open Rope in
let e = enum s in
  [? List : of_enum (f e) | f <- List :
    [take 5; skip 3 | - take 5; take 5; identity]]
```
**Exercise**

From a string `s`, return the first 5 characters, skip the next 3, then return the next 5 characters, the next 5 characters and the rest of the string.

```ocaml
given a string `s`
let `hairsplit s` =
  open String in
  let `e` = enum `s` in
  [? List : of_enum (f e) | f <- List :
    [take 5; skip 3 |- take 5; take 5; identity]]
```

**Exercise**

Same thing, but with Unicode.

```ocaml
given a rope `s`
let `hairsplit s` =
  open Rope in
  let `e` = enum `s` in
  [? List : of_enum (f e) | f <- List :
    [take 5; skip 3 |- take 5; take 5; identity]]
```
Text features, too

All these data structures support

- iteration, map, folds, filters, replacement, enumeration, construction from enumeration, searching from left to right, from right to left, from a given index, chopping, trimming, stripping, upper/lowercasing, splitting, slicing, splicing, etc.
- printing
- transcoding
- pattern-matching.
Introduction

Manipulating data
  From loops to enumerations
  Other data structures

I/O
  From Channels to I/O
  Composability
  Extensibility

Text
  From string to...
  Features

Conclusion
Enough for one day

Let’s not detail

▶ uniform number modules for functorization
▶ safe integers
▶ enumerable signature
▶ on-line help
▶ documentation by topics
▶ mostly flat module-space
▶ overlay modules for labels or exceptionless error management
▶ the Future module
▶ printing
▶ marshaling
▶ substrings
▶ path management
▶ package management
▶ calling the compilers from a module
▶ …
Status

**Latest version**  Alpha 3 for OCaml 3.10/3.11 being bugsquashed.

**Site**  http://batteries.forge.ocamlcore.org.

**License**  Mostly LGPL + LE, bits in BSD.

**Availability**  Tarball, GODI package.

**Tools**  ocaml, ocamlc, ocamlopt, ocamlcp, ocamlbuild.

**Size**  27,650 loc signatures, 24,407 loc implementations.

**Next version**  Beta 1, expected ca. March.
Status

Latest version  Alpha 3 for OCaml 3.10/3.11 being bugsquashed.


License  Mostly LGPL + LE, bits in BSD.

Availability  Tarball, GODI package.

Tools  ocaml, ocamlc, ocamlopt, ocamlcp, ocamlbuild.

Size  27,650 loc signatures, 24,407 loc implementations.

Next version  Beta 1, expected ca. March.

Testing needed!
Status

**Latest version**  Alpha 3 for OCaml 3.10/3.11 being bugsquashed.

**Site**  http://batteries.forge.ocamlcore.org.

**License**  Mostly LGPL + LE, bits in BSD.

**Availability**  Tarball, GODI package.

**Tools**  ocaml, ocamlc, ocamlopt, ocamlcp, ocamlbuild.

**Size**  27,650 loc signatures, 24,407 loc implementations.

**Next version**  Beta 1, expected ca. March.

  Testing needed!

**Applications**  Extrapol static analyzer for C.
Batteries for new apps

Larger standard library No more reimplementing lazy lists or standard operators or trivial list functions.
Higher-level library More composability, more extensibility, etc.
Syntactic sugar More readable algorithms.
Batteries for new apps

Larger standard library  No more reimplementing lazy lists or standard operators or trivial list functions.

Higher-level library  More composability, more extensibility, etc.

Syntactic sugar  More readable algorithms.

Fun!
Batteries for newbies

**Documentation** More examples, on-line help.

**Uniformity** Modules follow more rigorous patterns and should be easier to learn.

**Purity** Going imperative is less often necessary.
Batteries for newbies

Documentation More examples, on-line help.
Uniformity Modules follow more rigorous patterns and should be easier to learn.
Purity Going imperative is less often necessary.
Fun!
Batteries for new libraries

Conventions  Standard signatures, obsolete primitives, etc.
Batteries for new libraries

Conventions Standard signatures, obsolete primitives, etc.
Better composition
Fun!
Conventions  Standard signatures, obsolete primitives, etc.
Better composition
Fun!

Essentially, please consider compatibility with Batteries for your next libraries.
Problems to solve

- Huge binary size.
- Toplevel pretty-printers.
- Confusing error messages.
- Operator precedence for ⩽, ⩾.
- One-tarball distribution? Symbiosis?
In the future

- More OCamlNet ($\beta$).
- Preferences ($\beta$).
- Pa-do ($\beta$).
In the future

- More OCamlNet ($\beta$).
- Preferences ($\beta$).
- Pa-do ($\beta$).
- Logging.
- Relooking the documentation with iframes.
- Locales.
- Optimizations (eg strings).
- CoThreads.
In the future

- More OCamlNet (β).
- Preferences (β).
- Pa-do (β).
- Logging.
- Relooking the documentation with iframes.
- Locales.
- Optimizations (eg strings).
- CoThreads.
- oUnit?
- CamlGraph?
- AST/bytecode generation?
- Functional unparsing with Camlp4 support?
- Graphics? Cairo? Some UI toolkit?
- OS integration?
In the future

- More OCamlNet (β).
- Preferences (β).
- Pa-do (β).
- Logging.
- Relooking the documentation with iframes.
- Locales.
- Optimizations (e.g. strings).
- CoThreads.
- oUnit?
- CamlGraph?
- AST/bytecode generation?
- Functional unparsing with Camlp4 support?
- Graphics? Cairo? Some UI toolkit?
- OS integration?
- Don’t hesitate to use our Request for Features tracker.
In the future

- More OCamlNet ($\beta$).
- Preferences ($\beta$).
- Pa-do ($\beta$).
- Logging.
- Relooking the documentation with iframes.
- Locales.
- Optimizations (e.g., strings).
- CoThreads.
- oUnit?
- CamlGraph?
- AST/bytecode generation?
- Functional unparsing with Camlp4 support?
- Graphics? Cairo? Some UI toolkit?
- OS integration?
- Don’t hesitate to use our Request for Features tracker.
- And our bug tracker, of course.
Thank you

Questions?
If you’re... 

- A PhD, a PhD student or a future PhD student.
- Into OCaml, similar languages or Coq.
- Into compilers, concurrency, distributed systems, semantics, proof of programs.
- Into language tools, language front-ends, language design.
- Interested by safe programming for the web.

Contact me/us: MLState may have a job/PhD/internship for you.
If you’re... 

- A PhD, a PhD student or a future PhD student.
- Into OCaml, similar languages or Coq.
- Into compilers, concurrency, distributed systems, semantics, proof of programs.
- Into language tools, language front-ends, language design.
- Interested by safe programming for the web.

Contact me/us: MLState may have a job/PhD/internship for you.