



# OCaml

## Pragmatic Functional Programming

R. Mark Volkmann

Object Computing, Inc.



<https://objectcomputing.com>



[mark@objectcomputing.com](mailto:mark@objectcomputing.com)



[@mark\\_volkmann](https://twitter.com/mark_volkmann)



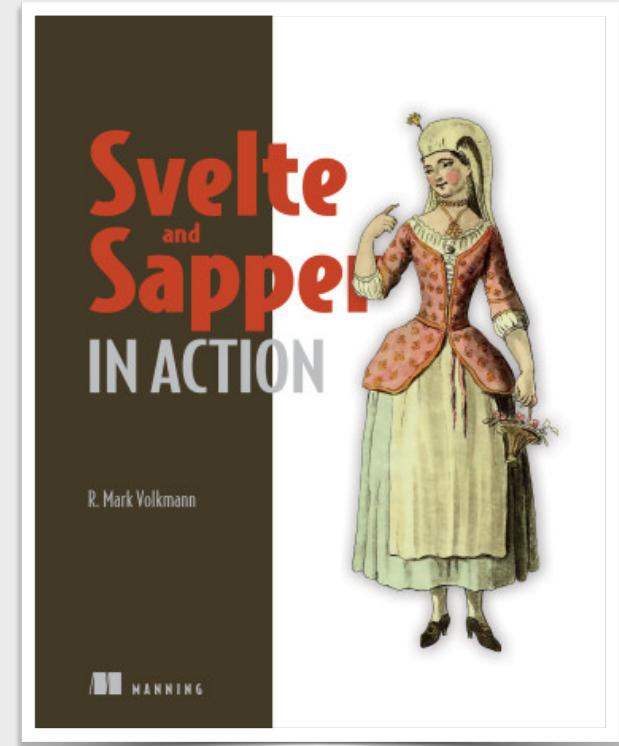
**OBJECT COMPUTING**  
YOUR OUTCOMES ENGINEERED

Slides at <https://github.com/mvolkmann/talks/>



# About Me

- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 43 years of professional software development experience
- Writer and speaker
- Blog at <https://mvolkmann.github.io/blog/>
- Author of Manning book “Svelte ... in Action” and Pragmatic Bookshelf book “htmx - fresh approach to applying web fundamentals”



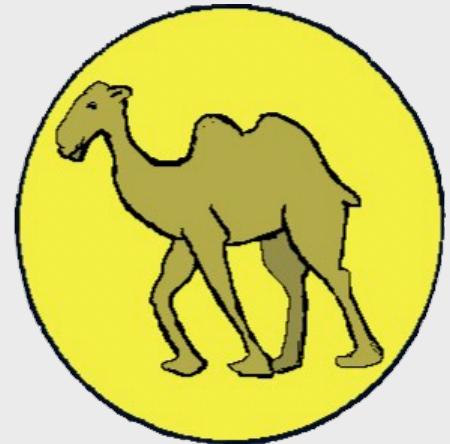
# ML Programming Language

- Short for “Meta Language”
- Designed by Robin Milner and others at University of Edinburgh
- Many other languages were influenced by this
  - Standard ML, Caml, Clojure, Elm, Erlang, F#, Haskell, OCaml, Rust, Scala
  - F# is heavily inspired by OCaml and runs on .NET platform
- F# is to C# as Clojure and Scala are to Java
  - these are ML-inspired languages that interoperate with an underlying non-ML language



# Caml Programming Language

- Short for “Categorical Abstract Machine Language”
- Released in 1985
- Predecessor of OCaml



# OCaml Programming Language

- “An industrial-strength functional programming language with an emphasis on expressiveness and safety”
- Released in 1996 - same year as Java
- Name is short for “Objective Caml”
- Adds support for object-oriented programming (not often used) and more expressive type system
- Comes with interpreter, compiler to bytecode, and compiler to native executables
  - compilers are implemented in OCaml
- Source files have `.ml` extension which stands for “meta language”
- Performance is generally about 50% that of C

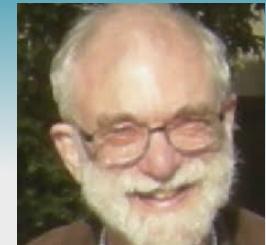


more compilation targets  
including assembly, C,  
JavaScript, and WebAssembly

# Notable Features

- Strong type inference and static type checking
- Automatic garbage collection
- Pattern matching with many ways to match
- Variant types  
(like enums with associated data)
- Polymorphic types  
(OCaml's version of generics)
- Very terse syntax for defining and calling named functions
- Functional programming (not as pure as Haskell, but more pragmatic)
- Automatic function currying
- Limited mutability (immutable by default, but can opt-in to some mutability)
- Module system that separates interface from implementation
- Fast compiler
- Great performance compared to other non-systems programming languages
- Can call C functions
- Ocaml PAckage Manager (OPAM)
- Dune build system
- `utop` REPL

# Type Inference



J. Roger Hindley



Robin Milner

- Uses variant of Hindley–Milner type system
  - type inference algorithm for statically typed, functional programming languages
  - infers the most general types of most expressions  
**without requiring explicit type annotations**
  - can specify types for documentation
  - LSP support in editors shows types on hover
  - partially enabled by making all operators operate on a specific type
    - for example,
      - + adds `int` values,
      - +. adds `float` values,
      - ^ concatenates `string` values

```
let add_ints a b = a + b
let add_floats a b = a +. b
let concat_strings a b = a ^ b
```

function definitions

# Who Uses OCaml?

- **Ahrefs** uses OCaml in its backend systems and data processing pipelines for Search Engine Optimization (SEO) tools and data analysis.
- **Bloomberg** created BuckleScript which compiles OCaml code to JavaScript. In 2022, BuckleScript was renamed to ReScript.
- **Citrix** uses OCaml in the Hypervisor software.
- **Coq** is an interactive theorem prover implemented in OCaml.
- **Docker** uses OCaml in their desktop software for Windows and macOS.
- **Facebook** uses OCaml for many things including
  - **Hack programming language** (extends PHP with static types)
  - **Facebook Messenger** (the web version)
  - **Flow** static type system for JavaScript
- **Infer** static analyzer for Java, C, C++, and Objective-C
- **Haxe** is a high-level cross-platform programming language that can be compiled to run on many platforms. Its compiler is implemented in OCaml.
- **Jane Street** uses OCaml for all their financial software, including algorithmic trading. They are one of the largest users and supporters of OCaml.
- **LexiFi** uses OCaml in their financial software for derivatives pricing and risk management.
- **T3** uses OCaml for algorithmic trading, quantitative analysis, risk management, and other financial software.



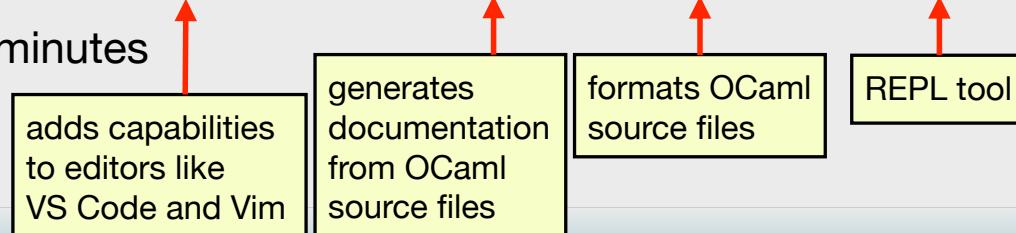
# Derivations

- **Reason** (was ReasonML)
  - alternative OCaml syntax and toolchain developed by Facebook
  - provides a more JavaScript-like syntax while retaining full compatibility with the OCaml language and its libraries
  - also supports JSX
- **ReScript** (was BuckleScript)
  - “a robustly typed language that compiles to efficient and human-readable JavaScript”
  - forked from Reason and not compatible with OCaml
- **Melange**
  - set of tools that work with OCaml and Reason code to generate and interoperate with JavaScript
  - can generate React components

Reason and ReScript are both alternatives to JavaScript that compile to JavaScript

# Installing

- Install OCaml package manager “opam”
  - for Linux and macOS
    - `bash -c "sh <(curl -fsSL https://raw.githubusercontent.com/ocaml/opam/master/shell/install.sh)"`
    - for Windows see <https://ocaml.org/install>
- Enter **opam init**
  - runs for around 5 minutes
- To install tools for development
  - **opam install ocaml-lsp-server odoc ocamlformat utop**
  - runs for around 4 minutes



# utop REPL

- Short for “Universal TOPIlevel”
- Launch by entering **utop**
- Enter OCaml expressions
- Only evaluated when terminated by `; ;`
  - allows entering multiple expressions separated by a single colon and allows them to span multiple lines
- Load an installed module with `#require "{module-name}"; ;`
- Load definitions from a file with `#use "{file-path}"; ;`
- Exit with `ctrl-d` or `#quit; ;`

```
Welcome to utop version 2.14.0 (using OCaml version 5.1.1)

Type #utop_help for help about using utop.

-( 09:14:23 )-< command 0 > { counter: 0 }-
utop # let square x = x * x;;
val square : int -> int = <fun>
-( 09:14:23 )-< command 1 > { counter: 0 }-
utop # square 3;;
- : int = 9
-( 09:14:30 )-< command 2 > { counter: 0 }-
utop # [REDACTED]
Arg Array ArrayLabels Assert_failure Atomic Bigarray Bool Buffe
```

# Using VS Code



- Install “OCaml Platform” extension from OCaml Labs
- For code formatting, create file `.ocamlformat` in each project root directory

- ex.

```
profile = default      first two lines  
version = 0.26.1       are required  
break-infix = fit-or-vertical  
if-then-else = fit-or-vertical  
parse-docstrings = true  
wrap-comments = true
```

# Running a Source File

- Source files that don't depend on other source files or installed modules can be run with the `ocaml` command
  - suppose `hello.ml` contains 

```
let () = print_endline "Hello, World!"
```
  - can run with `ocaml hello.ml`
- Otherwise it's best to use a build tool like `dune`
  - described later

# Comments



- No single-line comments
- Multi-line comments are surrounded by (\* . . . \*)
  - same as in XQuery
- Can nest these

# Primitive Types



- **unit** - one literal value `()`
  - represents having no value
  - return type of functions that don't return anything
- **bool** - 1 byte with literal values true and false
- **char** - 1 byte ASCII, not Unicode
- **int** - 8 or 4 bytes
- **float** - 8 bytes
- **string** - sequence of bytes, not Unicode characters
  - sequence can describe Unicode characters and some tooling assumes this

# Operators

**Arithmetic**

Operator	Description
<code>~-</code>	int negation
<code>+</code>	int addition
<code>-</code>	int subtraction
<code>*</code>	int multiplication
<code>/</code>	int division
<code>~-.</code>	float negation
<code>+. .</code>	float addition
<code>-. .</code>	float subtraction
<code>*. .</code>	float multiplication
<code>/. .</code>	float division
<code>**</code>	float exponentiation
<code>mod</code>	modulo

**Relational and Logical**

Operator	Description
<code>==</code>	physical equality; same address
<code>!=</code>	physical (inequality); different address
<code>=</code>	structural equality; same content
<code>&lt;&gt;</code>	structural inequality; different content
<code>&lt;</code>	less than
<code>&gt;</code>	greater than
<code>&lt;=</code>	less than or equal
<code>&gt;=</code>	greater than equal
<code>&amp;&amp;</code>	boolean and
<code>  </code>	boolean or
<code>not</code>	boolean not

**String**

Operator	Description
<code>^</code>	string concatenation
<code>^^</code>	format string concatenation

**Other**

Operator	Description
<code>!</code>	gets ref value (dereferences)
<code>:=</code>	sets ref value (assigns)
<code>@</code>	list concatenation
<code>@@</code>	function application
<code> &gt;</code>	reverse function application (aka pipe forward)

# Functions

- Noiseless syntax
  - `a b c` calls function `a` and passes it arguments `b` and `c`
  - `let a b c = ...` defines function `a` with parameters `b` and `c`
- Automatic currying
  - calling a function with fewer arguments than it has parameters returns a new function that has the remaining parameters

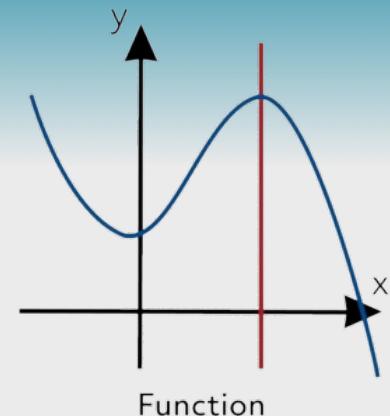
```
open Printf  
let add a b = a + b  
let add5 = add 5  
let () =  
  printf "sum = %d\n" (add 2 3); (* 5 *)  
  printf "sum = %d\n" (add5 2) (* 7 *)
```

let definitions

Function definitions must have at least one parameter and function calls must have at least one argument, even if they are the unit value () .

```
let hello () = print_endline "Hello"  
let () = hello ()
```

recursive functions are described later



# Named Arguments



- Can provide argument labels and parameter names
- Can be optional and have a default value
- When any are optional, must have at least one positional parameter
  - using () satisfies this

```
let greet ?(name = "World") ?suffix:(s = "!") () =
  Printf.printf "Hello, %s%s\n" name s

let () =
  greet ~name:"Mark" ~suffix:"." ();
  greet ~name:"Mark" ();
  greet ()
```

argument label and parameter name are both `name`

argument label is `suffix` and parameter name is `s`

note ~ before argument labels

Hello, Mark.  
Hello, Mark!  
Hello, World!

# Function Application

- @@ and |> operators provide an alternative to surrounding nested function calls with parentheses
- Example

```
let double x = x * 2
let square x = x * x
let () =
  let d = double 2 in
  let s = square d in
  print_int s;

  print_int (square (double 2));

  print_int @@ square @@ double @@ 2;

2 |> double |> square |> print_int
```

uses intermediate variables

all these print 16

|> operators looks like a right-pointing triangles when using a font with ligatures

The diagram illustrates the equivalence between two pieces of F# code. The left side shows a traditional approach using nested function calls and intermediate variables. The right side shows a more concise approach using the |> operator. Red arrows point from the highlighted text in the first code block to the corresponding parts in the second code block.

# Let Expressions vs. |> Operator

```
open Printf

type item = { description : string; price : int }

let tax_rate = 0.085
let cart =
  [
    { description = "eggs"; price = 250 };
    { description = "milk"; price = 350 };
    { description = "bread"; price = 300 };
  ]

let () =
  let subtotal = List.fold_left (fun acc item -> acc + item.price) 0 cart in
  let float_subtotal = float_of_int subtotal /. 100. in
  let tax = float_subtotal *. tax_rate in
  let total = float_subtotal +. tax in
  printf "Total: $%.2f\n" total;

  let total =
    cart
    |> List.fold_left (fun acc item -> acc + item.price) 0
    |> float_of_int
    |> ( *. ) 0.01
    |> ( *. ) (1. +. tax_rate)
  in
  printf "Total: $%.2f\n" total
```

let definitions

let expressions

same as above using reverse function application operator

# Expressions

- Can be a
  - literal - ex. `true`, `3`, `3.14`, or `"hello"`
  - variable - ex. `x`
  - let expression - ex. `let x = 3 in` or `let double x = x * 2 in`
  - keyword expression (ex. `if` ... or `for` ...)
  - function call - ex. `double x`
  - sequence of the above separated by semicolons
    - only last can have a value other than unit
- `ignore` function takes any value and returns `()`

```
ignore (add_dog "Comet" "Whippet");
add_dog "Oscar" "German Shorthaired Pointer" |> ignore;
print_endline "finished adding dogs"
```

assume `add_dog`  
returns a dog record

# Modules ...

- A `.ml` source file defines a module
- Module name comes from file name, but with first letter uppercased
  - `demo.ml` -> module `Demo`
- `.ml` files can contain the following:
  - `open` statements that make the names in another module available without a module name prefix
  - `include` statements that include values defined in another module
  - `type` aliases
  - `exception` definitions
  - `let` definitions that bind a name to a constant or function
  - `module` definitions that define submodules

not text from  
another source file

most common

# ... Modules

- Often describe a data type, ways to create instances, and operations on them
  - examples in standard library include `Array`, `Char`, `Hashtbl`, `List`, `Map`, `Option`, `Queue`, `Result`, `Set`, `Stack`, and `String`
  - standard library modules that do not describe a data type include `Printf`, `Random`, `Sys`, and `Unix`
- Modules that define a data type
  - do not define methods that are called on instances
  - instead define functions to which an instance and other arguments are passed
  - example

```
let numbers = [4; 1; 9; 7; 2]
let doubled = List.map (fun x -> x * 2) numbers
```

# Defining & Using a Custom Module

- Can define with `.ml` file (implementation) and optional `.mli` file (interface)
- When `.mli` file is present, only values it describes are exposed
  - other values defined in `.ml` file are private

```
type point = float * float           lib/math.mli

(**  
Computes the distance from one point to another.  
@param p1 the first point  
@param p2 the second point  
@return the distance between them  
*)  
val distance : point -> point -> float
```

generate HTML documentation from this using odoc tool

```
type point = float * float           lib/math.ml

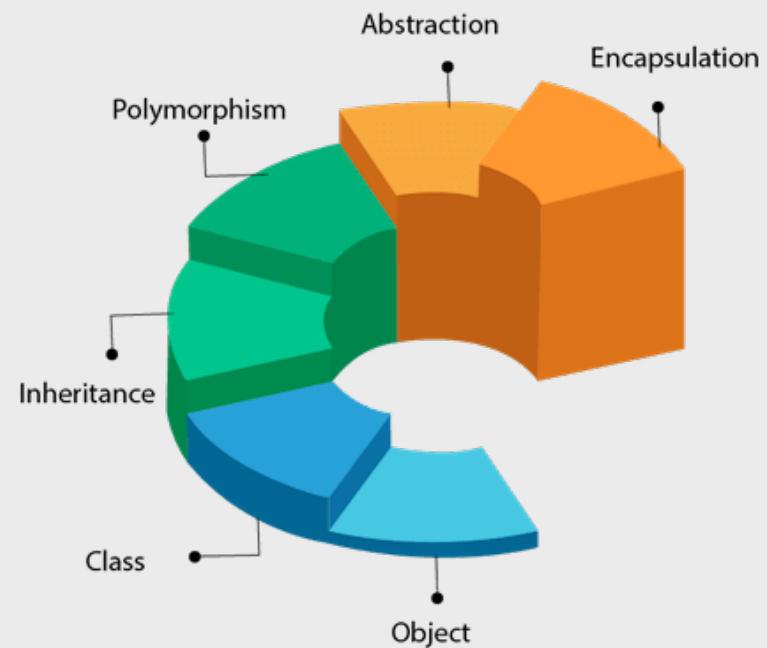
(* private function *)  
let square x = x *. x

(* public function *)  
let distance (x1, y1) (x2, y2) =  
  let dx = x2 -. x1 in  
  let dy = y2 -. y1 in  
  sqrt ((square dx) +. (square dy))
```

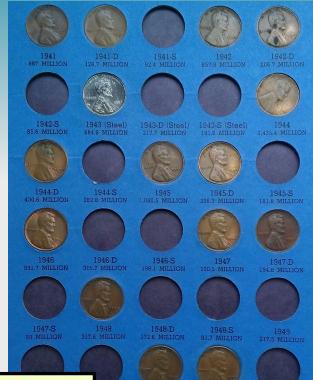
```
let () =                                bin/main.ml  
  let p1 = (0., 0.) in  
  let p2 = (3., 4.) in  
  let d = Module_demo.Math.distance p1 p2 in  
  assert (d = 5.)
```

# Object Oriented Programming

- While OCaml supports objects and classes, those are rarely used, so we won't cover them here



# Collection Types



- **tuple** - immutable, ordered values whose types can differ
- **list** - immutable, ordered values of same type in a linked list
- **association list** - immutable list of tuple pairs ← 

first value of each pair is a key and second is associated value
- **array** - mutable, ordered values of same type with fixed length
- **Set** - immutable, ordered values of same type with no duplicates
- **record** - immutable collection of named fields
- **Map** - immutable collection of key/value pairs
- **Hashtbl** - mutable collection of key/value pairs ← 

only built-in collection type that doesn't have a fixed length
- and more

# Literal Syntaxes

- **tuple**

```
(true, 7, 3.14, "Hello")
```

Tuples use commas, but others use semicolons.

- **list**

```
[1; 3; 7]
```

- **association list**

```
[("red", "FF0000"); ("green", "00FF00"); ("blue", "0000FF")]
```

- **array**

```
[| 1; 3; 7 |]
```

- **record**

```
{ name = "Mark"; number = 19; active = true }
```

# Pattern Matching ...

- Similar to a combination of JavaScript destructuring and **switch** statement

```
open Printf

type player = { name : string; number : int; active : bool }

let player = { name = "Wayne Gretzky"; number = 99; active = false }

let () =
    let { name; number; active } = player in
    if active then
        printf "%s wears number %d.\n" name number
    else
        printf "%s is no longer active.\n" name;

    match player with
    | { name; active } when not active -> printf "%s is no longer active.\n" name
    | { name; number } -> printf "%s wears number %d.\n" name number
```

It's okay for a type and a variable to have the same name.

# OCaml Patterns Can Match

- constant such as 7 or "summer"
- range of characters such as 'a' .. 'f'
- guard using `when` keyword such as `n when 7 <= n && n <= 9`
- variant type constructor such as `None` or `Some x`
- tuple such as `(_, "summer", temperature)`
  - means we don't care about the first element, the second element must be "summer", and we want to capture the third element
- list such as `[]`, `["summer"; other]`, or `first :: second :: rest`
- array such as `[||]` or `[|"summer"; other|]`
- record such as `{name; age = a}`
- multiple match expressions such as `7 | 8 | 9`
- variable to match anything and bind the value to it
- catch-all `_` which doesn't bind the value

# Recursion

- Recursive functions must be defined with `let rec`

```
open Printf

let rec list_sum l =
  match l with
  | [] -> 0
  | hd :: tl -> hd + list_sum tl

let rec list_sum = function
  | [] -> 0
  | hd :: tl -> hd + list_sum tl; shorter way to write previous function

let numbers = [ 1; 2; 3; 4 ]

let () =
  let sum1 = list_sum numbers in
  printf "sum1 = %d\n" sum1;

  let sum2 = List.fold_left ( + ) 0 numbers in
  printf "sum2 = %d\n" sum2; can use provided fold_left function instead
```

# Algebraic Data Types

- **Product types**  
describe a **conjunction**

- can be thought of as  
“this AND this AND this”
- examples in OCaml include  
**tuples** and **records**
  - tuples describe a cartesian product of values
  - example, the tuple type `float * float` describes the combination of every possible float value (x coordinate) with every possible float value (y coordinate)

- **Sum types**  
describe a **disjunction**

- can be thought of as  
“this OR this OR this”
- examples in OCaml include  
**variant types**
  - elements in a `list` are represented as a variant whose value can be the empty list `[]` or a cons cell created with the `::` operator that represents a value and a list tail
  - similarly for tree elements

# Variant Types ...

- Similar to enums in other languages, but can have associated data
- Options are called “constructors” or “tags”

```
open Printf

type color = Red | Green | Blue

let () =
  let color = Red in
  match color with
  | Red -> printf "FF0000\n"
  | Green -> printf "00FF00\n"
  | Blue -> printf "0000FF\n"
```

these constructors do not have associated data

# ... Variant Types

```
open Printf

type point = float * float

type shape =
| Circle of { center : point; radius : float }
| Rectangle of { lower_left : point; width : float; height : float }

let area shape =
  match shape with
  | Circle { radius = r } -> Float.pi *. r *. r
  | Rectangle { width = w; height = h } -> w *. h

let center = function
| Circle { center = c } -> c
| Rectangle { lower_left = x, y; width = w; height = h } ->
  (x +. (w /. 2.), y +. (h /. 2.))

let () =
  let c = Circle { center = (0., 0.); radius = 10. } in
  let r = Rectangle { lower_left = (0., 0.); width = 10.; height = 5. } in
  printf "c area = %f\n" (area c);
  printf "r area = %f\n" (area r);
  let x, y = center c in
  printf "c center = (%f, %f)\n" x y;
  let x, y = center r in
  printf "r center = (%f, %f)\n" x y;
```

these constructors  
have associated data

# Option & Result Variant Types

- **option** is a variant type with the constructors **None** and **Some of 'a**
- **result** is a variant type with the constructors **Ok of 'a** and **Error of 'e**
- It's common for functions to return an **option**
  - since there are no nulls in OCaml
  - ex. `List.find_opt (fun s -> s.score >= 90) students` names of functions that return an option often end in \_opt
- Some functions return a **result** instead of raising exceptions

# Exceptions ...



- Builtin type **exn** is an “extensible variant type”
- There are around 30 constructors of this type
- More can be defined with **exception** keyword

• Raise an exception with

```
raise SomeException  
or  
raise (OtherException v1 v2)
```

• Catch exceptions with

```
try some-expression with  
| exception SomeException -> expression  
| exception OtherException -> expression
```

• Some functions raise exceptions

- ex. this can raise a **Not\_found** exception

names of functions that raise  
exceptions often end in **\_exc**

```
List.find (fun s -> s.score >= 90) students
```

finds first matching student

# ... Exceptions



- Convenience functions
  - `failwith string` raises a `Failure` exception
  - `invalid_arg string` raises an `Invalid_argument` exception

# Built-in Mutable Types ...

- **Refs**

- represented by a record with mutable **content** field
- **!** operator gets value of **content** field      **!r** is short for **r.contents**
- **:=** operator modifies **content** field      **r := v** is short for **r.contents <- v**
- **incr** and **decr** functions update an **int ref**

```
let score = ref 0 in
  score := !score + 1;
  incr score;
  decr score;
  printf "score = %d\n" !score
```

- **Array elements**

- all array elements are mutable
- **arr.(index)** gets an element value
- **<-** operator updates an element

```
let numbers = [| 1; 2; 3 |] in
  numbers.(2) <- 4
```

# ... Builtin Mutable Types

- Record fields

```
type player = { name: string; mutable score: int }
let player = { name = "Mark"; score = 0 } in
player.score <- succ player.score
```

- Hashtbl collection

```
let dogs = Hashtbl.create 10 in
Hashtbl.add dogs "Comet" "Whippet";
Hashtbl.replace dogs "Comet" "Greyhound";
Hashtbl.remove dogs "Comet"
```

10 is an estimate for  
the number of key/value  
pairs that will be added

# Standard Library vs. Jane Street



- Jane Street developed replacements for OCaml standard library
- Two layers
  - Base - a minimal replacement
  - Core - extends Base to add more features
- To use these
  - install with `opam install Base` and `opam install Core`
  - open with `open Base` or `open Core` at top of each source file

Use of Jane Street modules  
somewhat splits the  
OCaml community in two.

# OPAM

- To install a package, `opam install package-name`
- To uninstall a package, `opam install package-name`
- **Switches** enable using specific versions of OCaml and packages
  - get “default” switch by default
  - create global switches that have names with  
`opam switch create switch-name ocaml-version`
  - list global switches with `open switch list`
  - use a global switch with `opam switch switch-name`
  - local switches are associated with a specific project directory
    - create a local switch by cd’ing to a project directory and entering `opam switch create .`
    - cd’ing to a project directory will activate its local switch

# Dune

- Most popular build system for OCaml and Reason
- Creates, builds, tests, and runs projects
- To create a project, **dune init project {name}** can also just manually  
create a few files
- To build a project, **dune build [-w]**
- To run tests, **dune test [-w]**
- To run project, **dune exec {exe-name}**
  - a project can define more than one executable target

# Dune Project File Structure

- *project-directory*
  - **dune-project** marks the top of a Dune project; defines project-wide configurations
  - **{project\_name}.opam** describes metadata and dependencies for OCaml packages
  - **\_build** holds generated files
  - **bin**
    - **dune**
    - **main.ml** main source file
  - **lib**
    - **dune**
    - **.ml** files that define modules
  - **test**
    - **dune**
    - **test\_{project\_name}.ml**



# dune Files

- Each directory contains a **dune** file
  - uses a LISP-like syntax that describes “stanzas”
  - example

```
(executable
  (public_name dream_demo)
  (name main)
  (flags (:standard -w -32) (:standard -w -69))
  (libraries dream ppx_deriving_yojson.runtime uuidm yojson)
  (preprocess (pps lwt_ppx ppx_deriving.show ppx_yojson_conv)))

(rule
  (targets form.ml dog_row.ml)
  (deps form.eml.html dog_row.eml.html)
  (action (run dream_eml %{deps} --workspace %{workspace_root})))
```

This rule is for processing JSX-like syntax used with the Dream framework.

**dream** is a web framework for OCaml and Reason

**yojson** module generates JSON from OCaml data structures

**uuidm** module generates UUID values

**ppx** stands for PreProcessor eXtension. These operate on the abstract syntax tree of a program.

# PreProcessing eXtensions (PPX)

- OCaml uses these to
  - generate code such as functions that operate on instances of a type
  - add support for new syntax
  - perform static analysis to provide additional compile-time checks or optimizations
- These operate on abstract syntax tree of a program
- Enable with **preprocess** stanza in **dune** files
  - ex. `(preprocess (pps ppx_deriving.show))`
- Example

```
type int_list = int list  
[@@deriving show] ← generates show_int_list function  
  
let numbers = [4; 1; 9; 7; 2]  
  
let () =  
  print_endline (show_int_list numbers)
```

pps stands for  
preprocessing specification

outputs  
[4; 1; 9; 7; 2]

# Resources

- **OCaml home page** - <https://ocaml.org>
- **OCaml Discord server** - <https://discord.gg/cCYQbqN>
- **My blog** - <https://mvolkmann.github.io/blog/> (select OCaml)
- **My example code** - <https://github.com/mvolkmann/ocaml-examples/>
- **Learn X in Y minutes** Where X=OCaml -  
<https://learnxinyminutes.com/docs/ocaml/>
- **“OCaml Programming: Correct + Efficient + Beautiful” book** -  
<https://cs3110.github.io/textbook/> see “YouTube playlist” link
- **“Real World OCaml” book** - <https://dev.realworldocaml.org/>



# Wrap Up

- OCaml is an interesting functional programming language
- Has many of the type features of Haskell, but is more pragmatic in regards to mutating state and other side effects
- OCaml type inference is incredible!
  - specifying types is mainly just for documentation
- Biggest impediment to OCaml adoption is the poor state of library documentation and lack of example code
  - being addressed by adding cookbook-like material to “Learn” section of main web site

