OCaml

Pragmatic Functional Programming

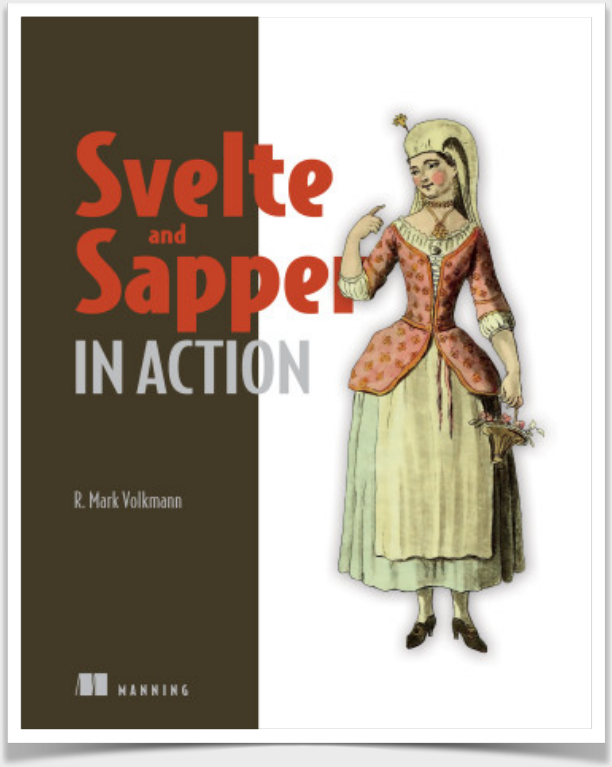
R. Mark Volkmann
Object Computing, Inc.
https://objectcomputing.com
mark@objectcomputing.com
@mark_volkmann

OBJECT COMPUTING
YOUR OUTCOMES ENGINEERED

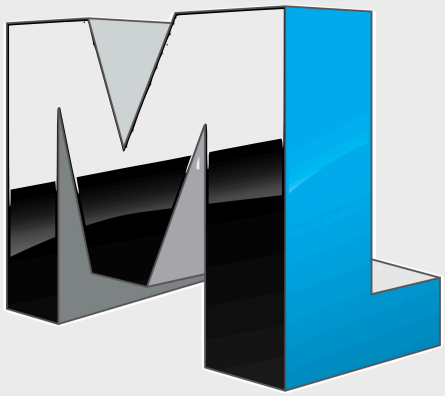
Slides at **https://github.com/mvolkmann/talks/**

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 upcoming Pragmatic Bookshelf book
  “htmx - fresh approach to applying web fundamentals”



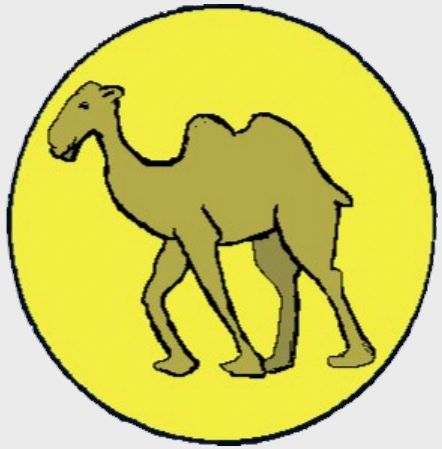
ML Programming Language



- Short for “Meta Language”
- Functional programming language designed by Robin Milner and others at University of Edinburgh
- Uses Hindley-Milner type system
- Influenced many other languages
  - 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
  - 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

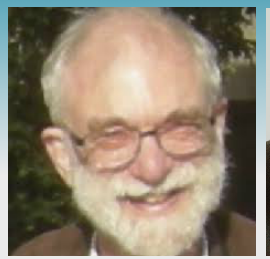
has more compilation targets including assembly, C, JavaScript, and WebAssembly

- Source files have `.ml` extension which stands for "meta language"
- Performance is generally about 50% that of C

Notable Features

- Strong/static type checking
- Incredible type inference
- Automatic garbage collection
- Function application operators
- 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 interfaces from implementations
- 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

- OCaml uses a 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

function definitions

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

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 Messager** (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.
- **Tarides** uses OCaml and contributes to the OCaml compiler, platform, and ecosystem

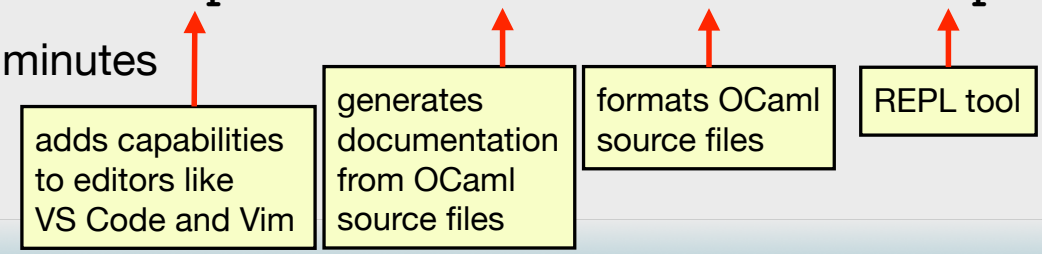
Derivations

All of these compile to JavaScript.

- **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

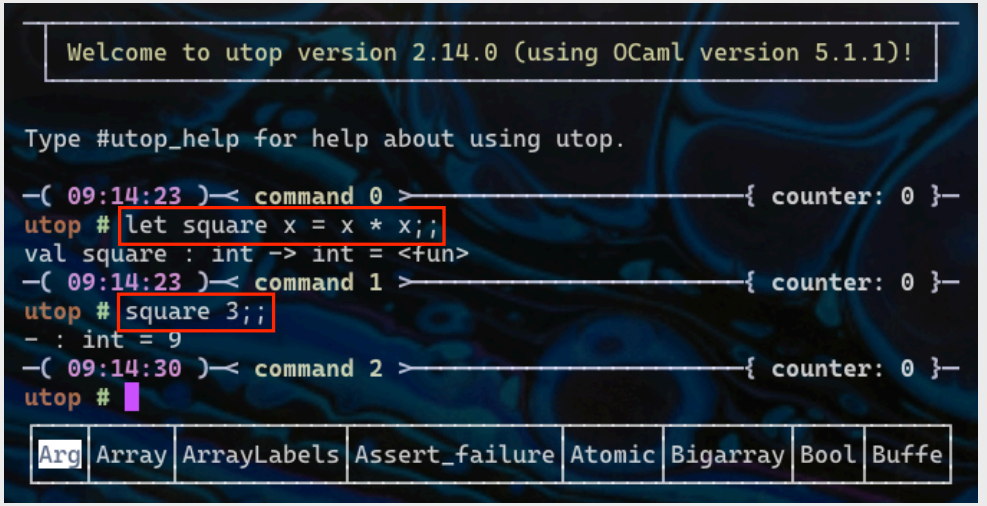
Installing

- 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 TOPlevel”
- 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 definitions from a file with **`#use "{file-path}";;`**
- Load an installed module with **`#require "{module-name}";;`**
- Exit with ctrl-d or **`#quit;;`**

Using VS Code

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

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

first two lines are required

Comments

- No single-line comment syntax
- 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, depending on processor
- **`float`** - 8 bytes
- **`string`** - sequence of bytes, not Unicode characters
  - sequence can describe Unicode characters and some libraries assume this

Spaces around this are not required, but they are preferred.

There is a corresponding standard library module for each of these whose name begins uppercase. These provide functions for operating on values of the type.

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

Ending a source file with **`let () =`** followed by an expression is similar to the main function in other languages.

This ensures that the result of the expression will be the unit value AND makes it clear that the purpose of the expression is the side effects it produces.

Operators

Arithmetic

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

Relational and Logical

Operator	Description
`==`	physical equality; same address
`!=`	physical (inequality); different address
`=`	structural equality; same content
`<>`	structural inequality; different content
`<`	less than
`>`	greater than
`<=`	less than or equal
`>=`	greater than equal
`&&`	boolean and
`||`	boolean or
`not`	boolean not

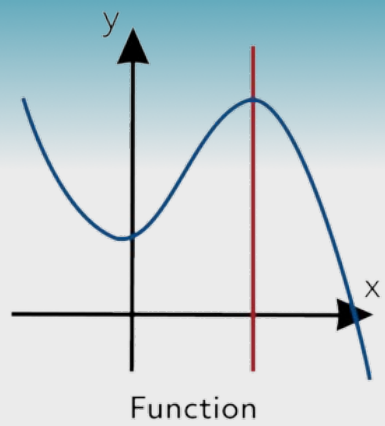
String

Operator	Description
`^`	string concatenation
`^^`	format string concatenation

Other

Operator	Description
`!`	gets ref value (dereferences)
`:=`	sets ref value (assigns)
`@`	list concatenation
`@@`	function application
`|>`	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`
- Must be defined before called
- Automatic currying
  - calling a function with fewer arguments than it has parameters returns a new function that has the remaining parameters

... Functions

- Examples

```
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

type of `add` is `int -> int -> int`

```
let add = (+)
```

point-free style

- 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
  - `?` before parameter makes it optional
- When any are optional, must have at least one positional parameter
  - using `()` satisfies this

argument label and parameter name are both **`name`**

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

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

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

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

note ~ before argument labels

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

```
2 ⊳ double ⊳ square ⊳ print_int
```

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
```

defines a record type

`let` definitions

list of records

prices in cents

anonymous function

`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 expressions above separated by semicolons
    - only last can have a value other than unit
- `ignore` function takes any value and returns `()`

variable

function

Technically `;` is a binary operator between two expressions.
The value of the left is discarded and the value on the right becomes the result.

```
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 values in another module available without a module name prefix
  - `include` statements that include values defined in another module
  - `type` declarations
  - `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

generate HTML documentation from this using `odoc` tool

lib/math.mli

```
type point = float * float

(**
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
```

lib/math.ml

```
type point = float * float

(* 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))
```

bin/main.ml

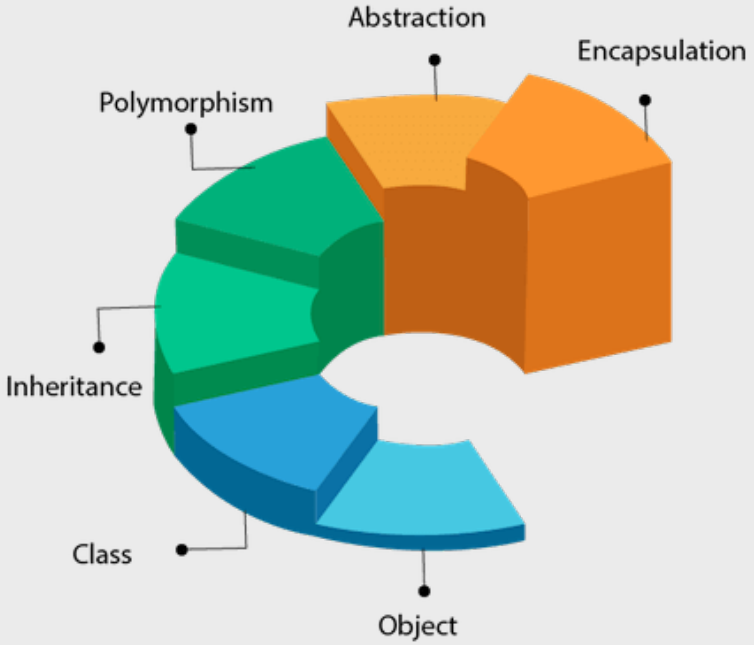
```
let () =
  let p1 = (0., 0.) in
  let p2 = (3., 4.) in
  let d = Module_demo.Math.distance p1 p2 in
  assert (d = 5.)
```

library name

module name

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
- **record** - immutable (by default) collection of named fields
- **Set** - immutable, ordered values of same type with no duplicates
- **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")
  ```
- **list**
  ```
  [1; 3; 7]
  ```
- **association list**
  ```
  [("red", "FF0000"); ("green", "00FF00"); ("blue", "0000FF")]
  ```
- **array**
  ```
  [| 1; 3; 7 |]
  ```
- **record**
  ```
  { name = "Mark"; number = 19; active = true }
  ```

Tuples use commas, but others use semicolons.

Pattern Matching

- Similar to a combination of JavaScript destructuring and **`switch`** statement
- Patterns must be exhaustive

```
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'`
- Variant type constructor such as `None` or `Some x`

more on variant types ahead

- Tuple such as `(_, "summer", temperature)`
  - means we don't care about first element, second element must be `"summer"`, and we want to capture third element value
- 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
- Guard using `when` keyword such as `n when 7 <= n && n <= 9`
- 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

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
```

shorter way to write previous function

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”

```
type color = Red | Green | Blue

let () =
  let color = Red in
  match color with
  | Red -> print_endline "FF0000"
  | Green -> print_endline "00FF00"
  | Blue -> print endline "0000FF"
```

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`

  polymorphic type pronounced “alpha”

  - it’s common for functions to return an `option` since there are no nulls in OCaml
  - names of functions that return an option often end in `_opt`
  - ex. `List.find_opt (fun s -> s.score >= 90) students`

    finds first matching student
- `result` is a variant type with the constructors `Ok of 'a` and `Error of 'e`
  - some functions return a `result` instead of raising exceptions

Exceptions ...

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

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

- Catch exceptions with

```
try
  some-expression
with
| exception SomeException -> expression
| exception OtherException (v1, v2) -> expression
```

... Exceptions

- Names of functions that raise exceptions *sometimes* end in **`_exc`**
  - ex. finds first matching student and can raise a **`Not_found`** exception

```
try
  let star = List.find (fun s -> s.score >= 90) students in
  printf "first star student is %s\n" star.name
with
| Not_found -> printf "no star student found"
```

- 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 **`contents`** field
  - **`:=`** operator modifies **`contents`** field
  - **`incr`** and **`decr`** functions update an **`int ref`**
- **Array elements**
  - all array elements are mutable
  - **`arr.(index)`** gets an element value
  - **`<-`** operator updates an element

`!r` is short for **`r.contents`**

`r := v` is short for **`r.contents <- v`**

No compiler magic here ... **`ref`**, **`incr`**, and **`decr`** are functions you could implement.

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

```
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 added

`"Comet"` is a key

Standard Library vs. Jane Street

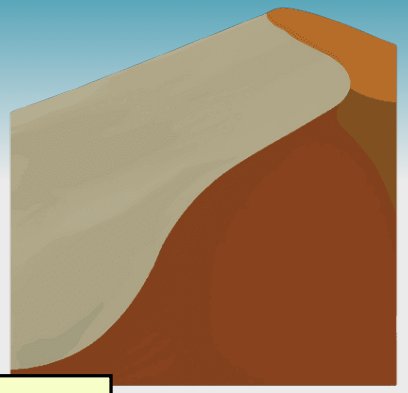
- Jane Street developed replacements for OCaml standard library
- Two layers
  - **`base`** - a minimal replacement
  - **`core`** - extends **`base`** to add more features
- To use these
  - install with **`opam install base`** and **`opam install core`**
  - open with **`open base`** or **`open core`**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

Use of Jane Street modules somewhat splits the OCaml community.

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`**
  - activate 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

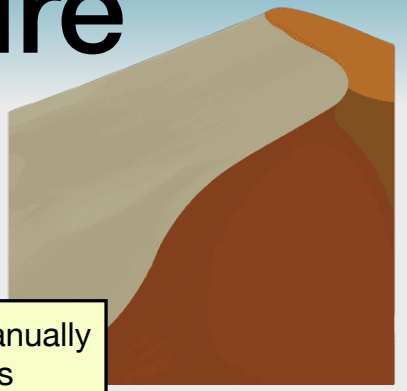


- Most popular build system for OCaml and Reason
- Creates, builds, tests, and runs projects
- To create a project, **`dune init project {name}`**
- To build a project, **`dune build [-w]`**
- To run tests, **`dune test [-w]`**
- To run an executable, **`dune exec {exe-name}`**
  - a project can define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

can also just manually create a few files

`-w` for watch mode

Dune Project File Structure



- *project-directory*
  - **`dune-project`**
  - **`{project_name}.opam`**
  - **`_build`**
  - **`bin`**
    - **`dune`**
    - **`main.ml`** main source file
  - **`lib`**
    - **`dune`**
    - **`.ml`** files that define modules
  - **`test`**
    - **`dune`**
    - **`test_{project_name}.ml`**

marks the top of a Dune project; defines project-wide configurations

generated file that describes metadata and dependencies for OCaml packages

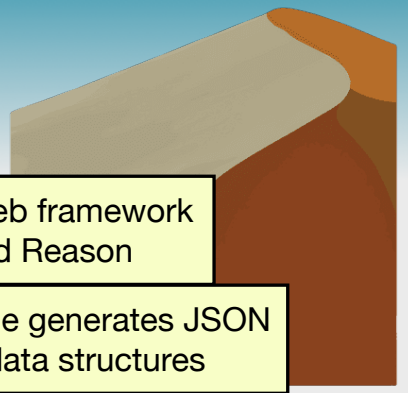
holds generated files

subdirectories **`bin`**, **`lib`**, and **`test`** are just conventions, not required

`dune` files describe the target type for files in the directory (executables, libraries, or tests), libraries used, and preprocessors to be applied

can also just manually create a few files

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})))
```

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

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

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))`**

`pps` stands for PreProcessing Specification

- Example

```
type int_list = int list
[@@deriving show]

let numbers = [4; 1; 9; 7; 2]

let () =
  print_endline (show_int_list numbers)
```

generates **`show_int_list`** function

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
- Advanced features add complexity
  - such as functors and GADTs
- 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

functors are functions that take a module and produce a new module

GADTs (Generalized Algebraic Data Structures) enable more precise and flexible type checking