ocaml+twt 0.90 quick reference

This sheet tries to concisely demonstrate most syntax forms recognized by the ocaml+twt preprocessor. If you need more details, check the examples included with the distribution. All structural whitespace in the following examples is significant: if a line is indented here, it must be indented in your source.

Applications

List.iter

    Printf.printf "\%d\n"

lst

List.map

    function Some x -> x

List.filter

    function
       | Some x -> true
       | None -> false

lst

(if b then (+) else (-))

    x
    y

Sequences

Nothing special:

statement-1
statement-2
...

let

Let looks more like it would in a procedural language:

let x = 1
sequence

let rec f x = sequence
and g x = sequence
and h y = sequence

sequence

let x = match y with
    | pat1 -> csq1
    | pat2 -> csq2

sequence

You can’t put a let and its consequent on one line (e.g. let x = y in f x).

if-then-else

    if condition then expression
    if condition then sequence
    if condition then expression else expression
    if condition then sequence else sequence
    if condition then
        sequence else if condition then
            sequence
        else
            sequence

fun

Nothing special, but you don’t need parentheses if the fun is on its own line:

fun x y -> expression
fun x y -> sequence

Pattern matching

All patterns occurring on their own line must be indented and have pipes:

function Some x -> true | None -> false

match expression with
    | pattern -> expression
    | pattern -> sequence
    | pattern -> expression

A match or function may appear on the same line as a let:

let x = function
    | pattern -> expression
    ...
sequence

Exception handling

Nothing special, given the above forms for pattern matching:

try expression with Exception -> expression

try
sequence
with
    | Exn1 -> expression
    | Exn2 -> sequence
Records, lists, and arrays

The preprocessor ignores anything within curly braces or square brackets, including newlines. Thus, indentation within these operators doesn’t matter, you still have to use ; to separate items, and complicated expressions must be parenthesized.

```ocaml
type point = { x : int; y : int }
type point = {
  x : int;
  y : int
}

if condition then
  [ 1; 2 ]
else
  [ 2; 1 ]

let x = [elem1; elem2]
sequence

let x = [elem1;
  elem2;
  ... ]
sequence

let x = {
  field1 = value1;
  field2 = value2;
  ...
} sequence

Loops

Don’t use done:

for i = 1 to 10 do expression

for i = 1 to 10 do
  sequence

while condition do expression

while condition do
  sequence

Modules

Don’t use end:

module A = struct
  type point = { x : int;
  y : int }

let origin = { x = 0; y = 0 }

let reflect_x { x = a; y = b } =
  { x = 0 - a;
    y = b }

The local module syntax is supported, but the struct must start on its own line. For functors, module Name = functor ...
  ... -> must appear as one line. See the modules.ml example for details. Module signatures are the same, except without end:

module A : sig
  type point = { x : int;
    y : int }

  val origin : point
  val reflect_x : point -> point

Objects

Don’t use end. Method and initializer bodies are any other sequence. See the objects.ml example for details.

class shape =
  object
    method virtual area : unit -> float
  end

class circle =
  object (self)
    inherit shape
    val r = 1.0
    method area () =
      3.14159 *. r *. r
  end

Union types

The syntax rules for pattern matching apply:

type shape =
  | Square
  | Circle
  | Triangle

Combining expressions

Because ocaml+twt is a line-oriented preprocessor, the following general rule applies when combining expressions on the same line:

If an expression spans multiple lines, it must start on its own line.

For example, the following will not work:

```ocaml
let lst2 = List.map
  string_of_float
  lst1
sequence
```

The multi-line application in this example must start on a new line, rather than on the same line as its containing expression. (Alternatively, you could just make it one parenthesized line.) There are a few exceptions to this rule: loops, local module structures, and immediate objects must always start on their own line (even if they are only one line), match

```ocaml
may appear on the same line as a let (even if their patterns are on individual lines), and records, lists, and arrays may span multiple lines as previously described.
```