

Objective Caml version 3.06

```

# type t = {x: int};;
type t = { x : int; }

# let f (p: t) = p.x;;
val f : t -> int = <fun>

# type t = <x: int>;;
type t = < x : int >

# let f (p: t) = p#x;;
val f : t -> int = <fun>

# let g (p: {y: int}) = p.y;;
Characters 6-7:
  let g (p: {y: int}) = p.y;;
      ^
Syntax error: ')' expected, the highlighted '(' might be unmatched

# let g (p: <y: int>) = p#y;;
val g : < y : int > -> int = <fun>

# let h p = p.z;;
Characters 10-13:
  let h p = p.z;;
      ^^^
Unbound record field label z

# let h p = p#z;;
val h : < z : 'a; .. > -> 'a = <fun>

# let h p = p#z + 1;;
val h : < z : int; .. > -> int = <fun>

# let h (p: <z: int; ..>) = p#z + 1;;
val h : < z : int; .. > -> int = <fun>

# class c1 =
object
  method x = 1
end;;
class c1 : object method x : int end

# let p1 = new c1;;
val p1 : c1 = <obj>

# let (q1: t) = new c1;;
val q1 : t = <obj>

# [p1; q1];;
- : c1 list = [<obj>; <obj>]

# p1 = q1;;
- : bool = false

# class c2 =
object
  method x = 2
end;;
class c2 : object method x : int end

# let p2 = new c2;;
val p2 : c2 = <obj>

# let (q2: t) = new c2;;
val q2 : t = <obj>

# let l = [p1; p2; q1; q2];;
val l : c1 list = [<obj>; <obj>; <obj>; <obj>]

# f;;
- : t -> int = <fun>

# List.map f l;;
- : int list = [1; 2; 1; 2]

# class c3 =

```

```

object
  method x = 0
  method y = 1
  method z = 2
end;;
      class c3 : object method x : int method y : int method z : int end

# let p3 = new c3;;
val p3 : c3 = <obj>

# let l' = p3 :: l;;
Characters 15-16:
  let l' = p3 :: l;;
                ^
This expression has type c1 list but is here used with type
  < x : int; y : int; z : int > list
Type c1 = < x : int > is not compatible with type
  < x : int; y : int; z : int >
Only the second object type has a method y

# f p3;;
Characters 2-4:
  f p3;;
    ^^
This expression has type < x : int; y : int; z : int >
but is here used with type t = < x : int >
Only the first object type has a method y

# g p3;;
Characters 2-4:
  g p3;;
    ^^
This expression has type < x : int; y : int; z : int >
but is here used with type < y : int >
Only the first object type has a method x

# h;;
- : < z : int; .. > -> int = <fun>

# h p3;;
- : int = 3

# (p3 :> t);;
- : t = <obj>

# f (p3 :> t);;
- : int = 0

# f (p3 :> c1);;
- : int = 0

# f (p3 :> c2);;
- : int = 0

# f p3;;
Characters 2-4:
  f p3;;
    ^^
This expression has type < x : int; y : int; z : int >
but is here used with type t = < x : int >
Only the first object type has a method y

# (p2 :> c3);;
Characters 1-3:
  (p2 :> c3);;
    ^^
This expression cannot be coerced to type c3 = < x : int; y : int; z : int >;
it has type c2 = < x : int > but is here used with type
  #c3 as 'a = < x : int; y : int; z : int; .. >
Only the second object type has a method y

# type t = {x: int};;
type t = { x : int; }

# type u = {x: int; y: int};;
type u = { x : int; y : int; }

# let p = {x = 1; y = 2};;

```

```

val p : u = {x = 1; y = 2}

# (p :> t);;
Characters 1-2:
  (p :> t);;
  ^
This expression cannot be coerced to type t; it has type u
but is here used with type t

# class produce_c1 =
object
  method produce () = new c1
end;;
class produce_c1 : object method produce : unit -> c1 end

# let prod1 = new produce_c1;;
val prod1 : produce_c1 = <obj>

# class produce_c3 =
object
  method produce () = new c3
end;;
class produce_c3 : object method produce : unit -> c3 end

# let prod3 = new produce_c3;;
val prod3 : produce_c3 = <obj>

# [prod1; prod3];;
Characters 8-13:
  [prod1; prod3];;
  ^^^^^^
This expression has type produce_c3 = < produce : unit -> c3 >
but is here used with type produce_c1 = < produce : unit -> c1 >
Type c3 = < x : int; y : int; z : int > is not compatible with type
c1 = < x : int >
Only the first object type has a method y

# (prod3 :> produce_c1);;
- : produce_c1 = <obj>

# (prod1 :> produce_c3);;
Characters 1-6:
  (prod1 :> produce_c3);;
  ^^^^^^
This expression cannot be coerced to type
produce_c3 = < produce : unit -> c3 >;
it has type produce_c1 = < produce : unit -> c1 > but is here used with type
< produce : unit -> (#c3 as 'a) >
Type c1 = < x : int > is not compatible with type
'a = < x : int; y : int; z : int; .. >
Only the second object type has a method y

# [(prod3 :> produce_c1); prod1];;
- : produce_c1 list = [<obj>; <obj>]

# class consume_c1 =
object
  method consume (x: c1) = ()
end;;
class consume_c1 : object method consume : c1 -> unit end

# class consume_c3 =
object
  method consume (x: c3) = ()
end;;
class consume_c3 : object method consume : c3 -> unit end

# let cons1 = new consume_c1;;
val cons1 : consume_c1 = <obj>

# let cons3 = new consume_c3;;
val cons3 : consume_c3 = <obj>

# (cons3 :> consume_c1);;
Characters 1-6:
  (cons3 :> consume_c1);;
  ^^^^^^
This expression cannot be coerced to type

```

```
consume_c1 = < consume : c1 -> unit >;
it has type consume_c3 = < consume : c3 -> unit > but is here used with type
consume_c1
Type c3 = < x : int; y : int; z : int > is not compatible with type
c1 = < x : int >
Only the first object type has a method y.
This simple coercion was not fully general. Consider using a double coercion.

# (cons1 :> consume_c3);;
Characters 1-6:
(cons1 :> consume_c3);;
^^^^^^
This expression cannot be coerced to type
consume_c3 = < consume : c3 -> unit >;
it has type consume_c1 = < consume : c1 -> unit > but is here used with type
consume_c3
Type c1 = < x : int > is not compatible with type
c3 = < x : int; y : int; z : int >
Only the second object type has a method y.
This simple coercion was not fully general. Consider using a double coercion.

#
```