

# Atdgen reference manual

## release 1.3.1

Martin Jambon  
© 2010–2011 MyLife

September 20, 2019

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Command-line usage</b>	<b>3</b>
2.1	Command-line help . . . . .	3
2.2	Atdgen-biniou example . . . . .	6
2.3	Atdgen-json example . . . . .	9
2.4	Validator example . . . . .	12
<b>3</b>	<b>Default type mapping</b>	<b>13</b>
<b>4</b>	<b>ATD Annotations</b>	<b>14</b>
4.1	Section <code>biniou</code> . . . . .	14
4.1.1	Field <code>biniou.repr</code> . . . . .	14
4.2	Section <code>json</code> . . . . .	16
4.2.1	Field <code>json.name</code> . . . . .	16
4.2.2	Field <code>json.repr</code> . . . . .	16
4.3	Section <code>ocaml</code> . . . . .	17
4.3.1	Field <code>ocaml.predef</code> . . . . .	17
4.3.2	Field <code>ocaml.mutable</code> . . . . .	18
4.3.3	Field <code>ocaml.default</code> . . . . .	18

<i>CONTENTS</i>	2
4.3.4 Field <code>ocaml.from</code> . . . . .	19
4.3.5 Field <code>ocaml.module</code> . . . . .	19
4.3.6 Field <code>ocaml.t</code> . . . . .	20
4.3.7 Field <code>ocaml.field_prefix</code> . . . . .	21
4.3.8 Field <code>ocaml.name</code> . . . . .	21
4.3.9 Field <code>ocaml.repr</code> . . . . .	22
4.3.10 Field <code>ocaml.validator</code> . . . . .	24
4.4 Section <code>ocaml_binious</code> . . . . .	24
4.5 Section <code>ocaml_json</code> . . . . .	25
4.6 Section <code>doc</code> . . . . .	26
4.6.1 Field <code>doc.text</code> . . . . .	26
<b>5 Library</b>	<b>34</b>

## 1 Introduction

Atdgen is a command-line program that takes as input type definitions in the ATD syntax and produces OCaml code suitable for data serialization and deserialization.

Two data formats are currently supported, these are biniou and JSON. Atdgen-biniou and Atdgen-json will refer to Atdgen used in one context or the other.

Atdgen was designed with efficiency and durability in mind. Software authors are encouraged to use Atdgen directly and to write tools that may reuse part of Atdgen's source code.

Atdgen uses the following packages that were developed in conjunction with Atdgen:

- `atd`: parser for the syntax of type definitions
- `biniou`: parser and printer for biniou, a binary extensible data format
- `yjson`: parser and printer for JSON, a widespread text-based data format

Atdgen does not use Camlp4.

## 2 Command-line usage

### 2.1 Command-line help

```
$ atdgen -help
```

Generate OCaml code offering:

- \* OCaml type definitions translated from ATD file (-t)
- \* serializers and deserializers for Biniou (-b)
- \* serializers and deserializers for JSON (-j)
- \* record-creating functions supporting default fields (-v)
- \* user-specified data validators (-v)

Recommended usage: `./atdgen (-t|-b|-j|-v|-dep|-list) example.atd`

-t

Produce files `example_t.mli` and `example_t.ml`  
containing OCaml type definitions derived from `example.atd`.

-b

Produce files `example_b.mli` and `example_b.ml`  
containing OCaml serializers and deserializers for the Biniou  
data format from the specifications in `example.atd`.

-j

Produce files `example_j.mli` and `example_j.ml` containing OCaml serializers and deserializers for the JSON data format from the specifications in `example.atd`.

`-v` Produce files `example_v.mli` and `example_v.ml` containing OCaml functions for creating records and validators from the specifications in `example.atd`.

`-dep` Output Make-compatible dependencies for all possible products of `atdgen -t, -b, -j` and `-v`, and exit.

`-list` Output a space-separated list of all possible products of `atdgen -t, -b, -j` and `-v`, and exit.

`-o [ PREFIX | - ]` Use this prefix for the generated files, e.g. `'foo/bar'` for `foo/bar.ml` and `foo/bar.mli`.  
`'-'` designates stdout and produces code of the form  
`struct ... end : sig ... end`

`-biniou` [deprecated in favor of `-t` and `-b`]  
Produce serializers and deserializers for Biniou including OCaml type definitions (default).

`-json` [deprecated in favor of `-t` and `-j`]  
Produce serializers and deserializers for JSON including OCaml type definitions.

`-j-std` Convert tuples and variants into standard JSON and refuse to print NaN and infinities (implying `-json` mode unless another mode is specified).

`-std-json` [deprecated in favor of `-j-std`]  
Same as `-j-std`.

`-j-pp <func>` OCaml function of type `(string -> string)` applied on the input of each `*_of_string` function generated by `atdgen` (JSON mode). This is originally intended for UTF-8 validation of the input which is not performed by `atdgen`.

`-j-defaults` Output JSON record fields even if their value is known to be the default.

`-j-strict-fields` Call `!Ag_util.Json.unknown_field_handler` for every unknown JSON field found in the input instead of simply skipping them.  
The initial behavior is to raise an exception.

`-j-custom-fields FUNCTION`

Call the given function of type (string -> unit)  
for every unknown JSON field found in the input  
instead of simply skipping them.  
See also -j-strict-fields.

-validate  
[deprecated in favor of -t and -v]  
Produce data validators from <ocaml validator="x"> annotations  
where x is a user-written validator to be applied on a specific  
node.  
This is typically used in conjunction with -extend because  
user-written validators depend on the type definitions.

-extend MODULE  
Assume that all type definitions are provided by the specified  
module unless otherwise annotated. Type aliases are created  
for each type, e.g.  
    type t = Module.t

-open MODULE1,MODULE2,...  
List of modules to open (comma-separated or space-separated)

-nfd  
Do not dump OCaml function definitions

-ntd  
Do not dump OCaml type definitions

-pos-fname FILENAME  
Source file name to use for error messages  
(default: input file name)

-pos-lnum LINENUM  
Source line number of the first line of the input (default: 1)

-rec  
Keep OCaml type definitions mutually recursive

-o-name-overlap  
Accept records and classic (non-polymorphic) variants with identical  
field or constructor names in the same module. Overlapping names are  
supported in OCaml since version 4.01.

Duplicate name checking will be skipped, and type annotations will  
be included in the implementation to disambiguate names.  
This is the default if atdgen was compiled for OCaml >= 4.01.0

-o-no-name-overlap  
Disallow records and classic (non-polymorphic) variants  
with identical field or constructor names in the same module.  
This is the default if atdgen was compiled for OCaml < 4.01.0

-version  
Print the version identifier of atdgen and exit.

-help Display this list of options

--help Display this list of options

## 2.2 Atdgen-biniou example

```
$ atdgen -t example.atd
$ atdgen -b example.atd
```

Input file `example.atd`:

```
type profile = {
  id : string;
  email : string;
  ~email_validated : bool;
  name : string;
  ?real_name : string option;
  ~about_me : string list;
  ?gender : gender option;
  ?date_of_birth : date option;
}

type gender = [ Female | Male ]

type date = {
  year : int;
  month : int;
  day : int;
}
```

is used to produce files `example_t.mli`, `example_t.ml`, `example_b.mli` and `example_b.ml`. This is `example_b.mli`:

```
(* Auto-generated from "example.atd" *)

type gender = Example_t.gender

type date = Example_t.date = { year: int; month: int; day: int }

type profile = Example_t.profile = {
  id: string;
  email: string;
  email_validated: bool;
  name: string;
  real_name: string option;
  about_me: string list;
  gender: gender option;
  date_of_birth: date option
```

```

}

(* Writers for type gender *)

val gender_tag : Bi_io.node_tag
(** Tag used by the writers for type {!gender}.
    Readers may support more than just this tag. *)

val write_untagged_gender :
  Bi_outbuf.t -> gender -> unit
(** Output an untagged biniou value of type {!gender}. *)

val write_gender :
  Bi_outbuf.t -> gender -> unit
(** Output a biniou value of type {!gender}. *)

val string_of_gender :
  ?len:int -> gender -> string
(** Serialize a value of type {!gender} into
    a biniou string. *)

(* Readers for type gender *)

val get_gender_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> gender)
(** Return a function that reads an untagged
    biniou value of type {!gender}. *)

val read_gender :
  Bi_inbuf.t -> gender
(** Input a tagged biniou value of type {!gender}. *)

val gender_of_string :
  ?pos:int -> string -> gender
(** Deserialize a biniou value of type {!gender}.
    @param pos specifies the position where
    reading starts. Default: 0. *)

(* Writers for type date *)

val date_tag : Bi_io.node_tag
(** Tag used by the writers for type {!date}.
    Readers may support more than just this tag. *)

val write_untagged_date :
  Bi_outbuf.t -> date -> unit

```

```

(** Output an untagged biniou value of type {!date}. *)

val write_date :
  Bi_outbuf.t -> date -> unit
  (** Output a biniou value of type {!date}. *)

val string_of_date :
  ?len:int -> date -> string
  (** Serialize a value of type {!date} into
    a biniou string. *)

(* Readers for type date *)

val get_date_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> date)
  (** Return a function that reads an untagged
    biniou value of type {!date}. *)

val read_date :
  Bi_inbuf.t -> date
  (** Input a tagged biniou value of type {!date}. *)

val date_of_string :
  ?pos:int -> string -> date
  (** Deserialize a biniou value of type {!date}.
    @param pos specifies the position where
    reading starts. Default: 0. *)

(* Writers for type profile *)

val profile_tag : Bi_io.node_tag
  (** Tag used by the writers for type {!profile}.
    Readers may support more than just this tag. *)

val write_untagged_profile :
  Bi_outbuf.t -> profile -> unit
  (** Output an untagged biniou value of type {!profile}. *)

val write_profile :
  Bi_outbuf.t -> profile -> unit
  (** Output a biniou value of type {!profile}. *)

val string_of_profile :
  ?len:int -> profile -> string
  (** Serialize a value of type {!profile} into
    a biniou string. *)

```



```

(* Readers for type profile *)

val get_profile_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> profile)
  (** Return a function that reads an untagged
      biniou value of type {!profile}. *)

val read_profile :
  Bi_inbuf.t -> profile
  (** Input a tagged biniou value of type {!profile}. *)

val profile_of_string :
  ?pos:int -> string -> profile
  (** Deserialize a biniou value of type {!profile}.
      @param pos specifies the position where
      reading starts. Default: 0. *)

```

Module `Example_t` (files `example_t.mli` and `example_t.ml`) contains all OCaml type definitions that can be used independently from Biniou or JSON.

For convenience, these definitions are also made available from the `Example_b` module whose interface is shown above. Any type name, record field name or variant constructor can be referred to using either module. For example, the OCaml expressions `((x : Example_t.date) : Example_b.date)` and `x.Example_t.year = x.Example_b.year` are both valid.

## 2.3 Atdgen-json example

```

$ atdgen -t example.atd
$ atdgen -j example.atd

```

Input file `example.atd`:

```

type profile = {
  id : string;
  email : string;
  ~email_validated : bool;
  name : string;
  ?real_name : string option;
  ~about_me : string list;
  ?gender : gender option;
  ?date_of_birth : date option;
}

```

```
type gender = [ Female | Male ]
```

```
type date = {
  year : int;
  month : int;
  day : int;
}
```

is used to produce files `example_t.mli`, `example_t.ml`, `example_j.mli` and `example_j.ml`. This is `example_j.mli`:

```
(* Auto-generated from "example.atd" *)
```

```
type gender = Example_t.gender
```

```
type date = Example_t.date = { year: int; month: int; day: int }
```

```
type profile = Example_t.profile = {
  id: string;
  email: string;
  email_validated: bool;
  name: string;
  real_name: string option;
  about_me: string list;
  gender: gender option;
  date_of_birth: date option
}
```

```
val write_gender :
  Bi_outbuf.t -> gender -> unit
  (** Output a JSON value of type {!gender}. *)
```

```
val string_of_gender :
  ?len:int -> gender -> string
  (** Serialize a value of type {!gender}
      into a JSON string.
      @param len specifies the initial length
      of the buffer used internally.
      Default: 1024. *)
```

```
val read_gender :
  Yojson.Safe.lexer_state -> Lexing.lexbuf -> gender
  (** Input JSON data of type {!gender}. *)
```

```
val gender_of_string :  
  string -> gender  
  (** Deserialize JSON data of type {!gender}. *)  
  
val write_date :  
  Bi_outbuf.t -> date -> unit  
  (** Output a JSON value of type {!date}. *)  
  
val string_of_date :  
  ?len:int -> date -> string  
  (** Serialize a value of type {!date}  
    into a JSON string.  
    @param len specifies the initial length  
    of the buffer used internally.  
    Default: 1024. *)  
  
val read_date :  
  Yojson.Safe.lexeme -> Lexing.lexbuf -> date  
  (** Input JSON data of type {!date}. *)  
  
val date_of_string :  
  string -> date  
  (** Deserialize JSON data of type {!date}. *)  
  
val write_profile :  
  Bi_outbuf.t -> profile -> unit  
  (** Output a JSON value of type {!profile}. *)  
  
val string_of_profile :  
  ?len:int -> profile -> string  
  (** Serialize a value of type {!profile}  
    into a JSON string.  
    @param len specifies the initial length  
    of the buffer used internally.  
    Default: 1024. *)  
  
val read_profile :  
  Yojson.Safe.lexeme -> Lexing.lexbuf -> profile  
  (** Input JSON data of type {!profile}. *)  
  
val profile_of_string :  
  string -> profile  
  (** Deserialize JSON data of type {!profile}. *)
```

Module `Example_t` (files `example_t.mli` and `example_t.ml`) contains all OCaml

type definitions that can be used independently from Biniou or JSON.

For convenience, these definitions are also made available from the `Example_j` module whose interface is shown above. Any type name, record field name or variant constructor can be referred to using either module. For example, the OCaml expressions `((x : Example_t.date) : Example_j.date)` and `x.Example_t.year = x.Example_j.year` are both valid.

## 2.4 Validator example

```
$ atdgen -t example.atd
$ atdgen -v example.atd
```

Input file `example.atd`:

```
type month = int <ocaml validator="fun x -> x >= 1 && x <= 12">
type day = int <ocaml validator="fun x -> x >= 1 && x <= 31">

type date = {
  year : int;
  month : month;
  day : day;
}
<ocaml validator="Date_util.validate_date">
```

is used to produce files `example_t.mli`, `example_t.ml`, `example_v.mli` and `example_v.ml`. This is `example_v.ml`, showing how the user-specified validators are used:

```
(* Auto-generated from "example.atd" *)

type month = Example_t.month

type day = Example_t.day

type date = Example_t.date = { year: int; month: month; day: day }

let validate_month = (
  fun x -> x >= 1 && x <= 12
)
let validate_day = (
  fun x -> x >= 1 && x <= 31
)
let validate_date : _ -> date -> _ = (
```

```

fun path x ->
  match ( Date_util.validate_date ) path x with
  | Some _ as err -> err
  | None ->
    match
      (
        validate_month
      ) ('Field "month" :: path) x.month
    with
    | Some _ as err -> err
    | None ->
      (
        validate_day
      ) ('Field "day" :: path) x.day
  )
let create_date
  ~year
  ~month
  ~day
  () : date =
{
  year = year;
  month = month;
  day = day;
}

```

### 3 Default type mapping

The following table summarizes the default mapping between ATD types and OCaml, binious and JSON data types. For each language more representations are available and are detailed in the next section of this manual.

ATD	OCaml	Binious	JSON
unit	unit	unit	null
bool	bool	bool	boolean
int	int	svint	number (int)
float	float	float64	number (not int)
string	string	string	string
option	option	numeric variants (tag 0)	None/Some variants
list	list	array	array
shared	no wrapping	shared	not implemented
variants	polymorphic variants	regular variants	variants
record	record	record	object
tuple	tuple	tuple	tuple

Notes:

- The JSON null value serves only as the unit value and is useful in practice only for instantiating parametrized types with “nothing”. Option types have a distinct representation that does not use the null value.
- OCaml floats are written to JSON numbers with either a decimal point or an exponent such that they are distinguishable from ints, even though the JSON standard does not require a distinction between the two.
- The optional values of record fields denoted in ATD by a question mark are unwrapped or omitted in both biniou and JSON.
- JSON option values and JSON variants are represented in standard JSON (`atdgen -j -j-std`) by a single string e.g. `"None"` or a pair in which the first element is the name (constructor) e.g. `["Some", 1234]`. Yojson also provides a specific syntax for variants using edgy brackets: `<"None">`, `<"Some": 1234>`.
- Biniou field names and variant names other than the option types use the hash of the ATD field or variant name and cannot currently be overridden by annotations.
- JSON tuples in standard JSON (`atdgen -j -j-std`) use the array notation e.g. `["ABC", 123]`. Yojson also provides a specific syntax for tuples using parentheses, e.g. `("ABC", 123)`.
- Types defined as `abstract` are defined in another module.

## 4 ATD Annotations

### 4.1 Section biniou

#### 4.1.1 Field `biniou.repr`

##### Integers

*Position:* after `int` type

*Values:* `svint` (default), `uvint`, `int8`, `int16`, `int32`, `int64`

*Semantics:* specifies an alternate type for representing integers. The default type is `svint`. The other integers types provided by biniou are supported by `Atdgen-biniou`. They have to map to the corresponding OCaml types in accordance with the following table:

Biniou type	Supported OCaml type	OCaml value range
svint	int	min_int ... max_int
uvint	int	0 ... max_int, min_int ... -1
int8	char	'\000' ... '\255'
int16	int	0 ... 65535
int32	int32	Int32.min_int ... Int32.max_int
int64	int64	Int64.min_int ... Int64.max_int

In addition to the mapping above, if the OCaml type is `int`, any biniou integer type can be read into OCaml data regardless of the declared biniou type.

*Example:*

```
type t = {
  id : int
    <ocaml repr="int64">
    <biniou repr="int64">;
  data : string list;
}
```

### Floating-point numbers

*Position:* after `float` type

*Values:* `float64` (default), `float32`

*Semantics:* `float32` allows for a shorter serialized representation of floats, using 4 bytes instead of 8, with reduced precision. OCaml floats always use 8 bytes, though.

*Example:*

```
type t = {
  lat : float <biniou repr="float32">;
  lon : float <biniou repr="float32">;
}
```

### Arrays and tables

*Position:* applies to lists of records

*Values:* `array` (default), `table`

*Semantics:* `table` uses biniou's table format instead of a regular array for serializing OCaml data into biniou. Both formats are supported for reading into OCaml data regardless of the annotation. The table format allows

*Example:*

```
type item = {
```

```

    id : int;
    data : string list;
}

type items = item list <binou repr="table">

```

## 4.2 Section json

### 4.2.1 Field json.name

*Position:* after field name or variant name

*Values:* any string making a valid JSON string value

*Semantics:* specifies an alternate object field name or variant name to be used by the JSON representation.

*Example:*

```

type color = [
  Black <json name="black">
  | White <json name="white">
  | Grey <json name="grey">
]

type profile = {
  id <json name="ID"> : int;
  username : string;
  background_color : color;
}

```

A valid JSON object of the `profile` type above is:

```

{
  "ID": 12345678,
  "username": "kimforever",
  "background_color": "black"
}

```

### 4.2.2 Field json.repr

#### Association lists

*Position:* after (string \* \_) list type

*Values:* object



*Semantics:* uses JSON's object notation to represent association lists.

*Example:*

```
type counts = (string * int) list <json repr="object">
```

A valid JSON object of the `counts` type above is:

```
{
  "bob": 3,
  "john": 1408,
  "mary": 450987,
  "peter": 93087
}
```

Without the annotation `<json repr="object">`, the data above would be represented as:

```
[
  [ "bob", 3 ],
  [ "john", 1408 ],
  [ "mary", 450987 ],
  [ "peter", 93087 ]
]
```

## Floats

*Position:* after `float` type

*Values:* `int`

*Semantics:* specifies a float value that must be rounded to the nearest integer and represented in JSON without a decimal point nor an exponent.

*Example:*

```
type unixtime = float <json repr="int">
```

## 4.3 Section `ocaml`

### 4.3.1 Field `ocaml.predef`

*Position:* left-hand side of a type definition, after the type name

*Values:* `none`, `true` or `false`

*Semantics:* this flag indicates that the corresponding OCaml type definition must be omitted.

*Example:*

```
(* Some third-party OCaml code *)
type message = {
  from : string;
  subject : string;
  body : string;
}

(*
  Our own ATD file used for making message_of_string and
  string_of_message functions.
*)
type message <ocaml predef> = {
  from : string;
  subject : string;
  body : string;
}
```

#### 4.3.2 Field `ocaml.mutable`

*Position:* after a record field name

*Values:* `none`, `true` or `false`

*Semantics:* this flag indicates that the corresponding OCaml record field is mutable.

*Example:*

```
type counter = {
  total <ocaml mutable> : int;
  errors <ocaml mutable> : int;
}
```

translates to the following OCaml definition:

```
type counter = {
  mutable total : int;
  mutable errors : int;
}
```

#### 4.3.3 Field `ocaml.default`

*Position:* after a record field name marked with a `~` symbol or at the beginning of a tuple field.

*Values:* any valid OCaml expression

*Semantics:* specifies an explicit default value for a field of an OCaml record or tuple, allowing that field to be omitted.

*Example:*

```
type color = [ Black | White | Rgb of (int * int * int) ]

type ford_t = {
  year : int;
  ~color <ocaml default="Black"> : color;
}

type point = (int * int * <ocaml default="0"> : int)
```

#### 4.3.4 Field `ocaml.from`

*Position:* left-hand side of a type definition, after the type name

*Values:* OCaml module name without the `_t`, `_b`, `_j` or `_v` suffix. This can be also seen as the name of the original ATD file, without the `.atd` extension and capitalized like an OCaml module name.

*Semantics:* specifies the base name of the OCaml modules where the type and values coming with that type are defined.

It is useful for ATD types defined as `abstract` and for types annotated as predefined using the annotation `<ocaml predef>`. In both cases, the missing definitions must be provided by modules composed of the base name and the standard suffix assumed by Atdgen which is `_t`, `_b`, `_j` or `_v`.

*Example:* First input file `part1.atd`:

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

Second input file `part2.atd` depending on the first one:

```
type point <ocaml from="Part1"> = abstract
type points = point list
```

#### 4.3.5 Field `ocaml.module`

In most cases since Atdgen 1.2.0 `module` annotations are deprecated in favor of `from` annotations previously described.

*Position:* left-hand side of a type definition, after the type name

*Values:* OCaml module name

*Semantics:* specifies the OCaml module where the type and values coming with that type are defined. It is useful for ATD types defined as **abstract** and for types annotated as predefined using the annotation `<ocaml predef>`. In both cases, the missing definitions can be provided either by globally opening an OCaml module with an OCaml directive or by specifying locally the name of the module to use.

The latter approach is recommended because it allows to create type and value aliases in the OCaml module being generated. It results in a complete module signature regardless of the external nature of some items.

*Example:* Input file `example.atd`:

```
type document <ocaml module="Doc"> = abstract

type color <ocaml predef module="Color"> =
  [ Black | White ] <ocaml repr="classic">

type point <ocaml predef module="Point"> = {
  x : float;
  y : float;
}
```

gives the following OCaml type definitions (file `example.mli`):

```
type document = Doc.document

type color = Color.color = Black | White

type point = Point.point = { x: float; y: float }
```

Now for instance `Example.Black` and `Color.Black` can be used interchangeably in other modules.

#### 4.3.6 Field `ocaml.t`

*Position:* left-hand side of a type definition, after the type name. Must be used in conjunction with a `module` field.

*Values:* OCaml type name as found in an external module.

*Semantics:* This option allows to specify the name of an OCaml type defined in an external module.

It is useful when the type needs to be renamed because its original name is already in use or not enough informative. Typically we may want to give the name `foo` to a type originally defined in OCaml as `Foo.t`.

*Example:*

```

type foo <ocaml_biniou module="Foo" t="t"> = abstract
type bar <ocaml_biniou module="Bar" t="t"> = abstract
type t <ocaml_biniou module="Baz"> = abstract

```

allows local type names to be unique and gives the following OCaml type definitions:

```

type foo = Foo.t
type bar = Bar.t
type t = Baz.t

```

#### 4.3.7 Field `ocaml.field_prefix`

*Position:* record type expression

*Values:* any string making a valid prefix for OCaml record field names

*Semantics:* specifies a prefix to be prepended to each field of the OCaml definition of the record. Overridden by alternate field names defined on a per-field basis.

*Example:*

```

type point2 = {
  x : int;
  y : int;
} <ocaml field_prefix="p2_">

```

gives the following OCaml type definition:

```

type point2 = {
  p2_x : int;
  p2_y : int;
}

```

#### 4.3.8 Field `ocaml.name`

*Position:* after record field name or variant name

*Values:* any string making a valid OCaml record field name or variant name

*Semantics:* specifies an alternate record field name or variant names to be used in OCaml.

*Example:*

```

type color = [

```

```

    Black <ocaml name="Grey0">
  | White <ocaml name="Grey100">
  | Grey <ocaml name="Grey50">
]

type profile = {
  id <ocaml name="profile_id"> : int;
  username : string;
}

```

gives the following OCaml type definitions:

```

type color = [
  'Grey0
  | 'Grey100
  | 'Grey50
]

type profile = {
  profile_id : int;
  username : string;
}

```

#### 4.3.9 Field `ocaml.repr`

##### Integers

*Position:* after `int` type

*Values:* `char`, `int32`, `int64`, `float`

*Semantics:* specifies an alternate type for representing integers. The default type is `int`, but `char`, `int32`, `int64` or `float` can be used instead.

The three types `char`, `int32` and `int64` are supported by both `Atdgen-biniou` and `Atdgen-json` but `Atdgen-biniou` currently requires that they map to the corresponding fixed-width types provided by the `biniou` format.

The type `float` is only supported in conjunction with JSON and is useful when an OCaml float is used to represent an integral value, such as a time in seconds returned by `Unix.time()`. When converted into JSON, floats are rounded to the nearest integer.

*Example:*

```

type t = {
  id : int
  <ocaml repr="int64">

```

```

    <binio repr="int64">;
    data : string list;
  }

```

### Lists and arrays

*Position:* after a list type

*Values:* `array`

*Semantics:* maps to OCaml's `array` type instead of `list`.

*Example:*

```

type t = {
  id : int;
  data : string list
  <ocaml repr="array">;
}

```

### Sum types

*Position:* after a sum type (denoted by square brackets)

*Values:* `classic`

*Semantics:* maps to OCaml's classic variants instead of polymorphic variants.

*Example:*

```

type fruit = [ Apple | Orange ] <ocaml repr="classic">

```

translates to the following OCaml type definition:

```

type fruit = Apple | Orange

```

### Shared values

*Position:* after a `shared` type

*Values:* `ref`

*Semantics:* wraps the value using OCaml's `ref` type, which is as of Atdgen 1.1.0 the only way of sharing values other than records.

*Example:*

```

type shared_string = string shared <ocaml repr="ref">

```

translates to the following OCaml type definition:

```
type shared_string = string ref
```

#### 4.3.10 Field `ocaml.validator`

*Position:* after any type expression except type variables

*Values:* OCaml function that takes one argument of the given type and returns a bool

*Semantics:* `atdgen -v` produces for each type named *t* a function `validate_t`:

```
val validate_t : t -> bool
```

Such a function returns true if and only if the value and all of its subnodes pass all the validators specified by annotations of the form `<ocaml validator="...">`.

*Example:*

```
type positive = int <ocaml validator="fun x -> x > 0">

type point = {
  x : positive;
  y : positive;
  z : int;
}
<ocaml validator="Point.validate">
(* Some validating function from a user-defined module Point *)
```

The generated `validate_point` function is equivalent to the following:

```
let validate_point p =
  Point.validate p
  && (fun x -> x > 0) p.x
  && (fun x -> x > 0) p.y
```

## 4.4 Section `ocaml_binious`

Section `ocaml_binious` takes precedence over section `ocaml` in Biniou mode (`-b`) for the following fields:

- `predef` (see 4.3.1)
- `module` (see 4.3.5)
- `t` (see 4.3.6)



## 4.5 Section `ocaml_json`

Section `ocaml_json` takes precedence over section `ocaml` in JSON mode (`-j`) for the following fields:

- `predef` (see 4.3.1)
- `module` (see 4.3.5)
- `t` (see 4.3.6)

*Example:*

This example shows how to parse a field into a generic tree of type `Yojson.Safe.json` rather than a value of a specialized OCaml type.

```
type dyn <ocaml_json module="Yojson.Safe" t="json"> = abstract
```

```
type t = { foo: int; bar: dyn }
```

translates to the following OCaml type definitions:

```
type dyn = Yojson.Safe.json
```

```
type t = { foo : int; bar : dyn }
```

Sample OCaml value of type `t`:

```
{
  foo = 12345;
  bar =
    'List [
      'Int 12;
      'String "abc";
      'Assoc [
        "x", 'Float 3.14;
        "y", 'Float 0.0;
        "color", 'List [ 'Float 0.3; 'Float 0.0; 'Float 1.0 ]
      ]
    ]
}
```

Corresponding JSON data as obtained with `string_of_t`:

```
{"foo":12345,"bar":[12,"abc",{ "x":3.14,"y":0.0,"color":[0.3,0.0,1.0]}]}
```

## 4.6 Section doc

Unlike comments, `doc` annotations are meant to be propagated into the generated source code. This is useful for making generated interface files readable without having to consult the original ATD file.

Generated source code comments can comply to a standard format and take advantage of documentation generators such as `javadoc` or `ocamldoc`.

### 4.6.1 Field `doc.text`

*Position:*

- after the type name on the left-hand side of a type definition
- after the type expression on the right hand of a type definition (but not after any type expression)
- after record field names
- after variant names

*Values:* UTF-8-encoded text using a minimalistic markup language

*Semantics:* The markup language is defined as follows:

- Blank lines separate paragraphs.
- `{ }` can be used to enclose inline verbatim text.
- `{ { } }` can be used to enclose verbatim text where whitespace is preserved.
- The backslash character is used to escape special character sequences. In regular paragraph mode the special sequences are `[\\]`, `[ ]` and `[ ]`. In inline verbatim text, special sequences are `[\\]` and `[ ]`. In verbatim text, special sequences are `[\\]` and `[ ]`.

*Example:* The following is a full example demonstrating the use of `doc` annotations but also shows the full interface file `genealogy.mli` generated using:

```
$ atdgen -b genealogy.atd
```

Input file `genealogy.atd`:

```
<doc text="Type definitions for family trees">
```

```

type tree = {
  members : person list;
  filiations : filiation list;
}

type filiation = {
  parent : person_id;
  child : person_id;
  filiation_type : filiation_type;
}
  <doc text="Connection between parent or primary caretaker and child">

type filiation_type = {
  ?genetic : bool option;
  ?pregnancy : bool option;
  ?raised_from_birth : bool option;
  ?raised : bool option;
  ?stepchild : bool option;
  ?adopted : bool option;
}
  <doc text="
Example of a father who raised his child from birth
but may not be the biological father:

{{{
{
  genetic = None;
  pregnancy = Some false;
  raised_from_birth = Some true;
  raised = Some true;
  stepchild = Some false;
  adopted = Some false;
}
}}}}
">

type person_id
  <doc text="Two persons with the same {{person_id}} must be the same
person. Two persons with different {{person_id}}s
may be the same person if there is not enough evidence to
support it."> = int

type person = {
  person_id : person_id;
  name : string;
  ~gender : gender list;

```

```

?biological_gender
  <doc text="Biological gender actually used for procreating"> :
    gender option;
}

```

```

type gender =
[
| F <doc text="female">
| M <doc text="male">
]
  <doc text="Gender, definition depending on the context">

```

translates using `atdgen -b genealogy.atd` into the following OCaml interface file `genealogy_b.mli` with ocaml-doc-compliant comments:

```

(* Auto-generated from "genealogy.atd" *)

(** Type definitions for family trees *)

(**
  Two persons with the same [person_id] must be the same person. Two persons
  with different [person_id]s may be the same person if there is not enough
  evidence to support it.
*)
type person_id = Genealogy_t.person_id

(** Gender, definition depending on the context *)
type gender = Genealogy_t.gender

type person = Genealogy_t.person = {
  person_id: person_id;
  name: string;
  gender: gender list;
  biological_gender: gender option
  (** Biological gender actually used for procreating *)
}

(**
  Example of a father who raised his child from birth but may not be the
  biological father:

```

```

{v
\{
  genetic = None;
  pregnancy = Some false;

```

```

    raised_from_birth = Some true;
    raised = Some true;
    stepchild = Some false;
    adopted = Some false;
  \}
v}
*)
type filiation_type = Genealogy_t.filiation_type = {
  genetic: bool option;
  pregnancy: bool option;
  raised_from_birth: bool option;
  raised: bool option;
  stepchild: bool option;
  adopted: bool option
}

(** Connection between parent or primary caretaker and child *)
type filiation = Genealogy_t.filiation = {
  parent: person_id;
  child: person_id;
  filiation_type: filiation_type
}

type tree = Genealogy_t.tree = {
  members: person list;
  filiations: filiation list
}

(* Writers for type person_id *)

val person_id_tag : Bi_io.node_tag
(** Tag used by the writers for type {!person_id}.
    Readers may support more than just this tag. *)

val write_untagged_person_id :
  Bi_outbuf.t -> person_id -> unit
(** Output an untagged biniou value of type {!person_id}. *)

val write_person_id :
  Bi_outbuf.t -> person_id -> unit
(** Output a biniou value of type {!person_id}. *)

val string_of_person_id :
  ?len:int -> person_id -> string
(** Serialize a value of type {!person_id} into
    a biniou string. *)

```

```

(* Readers for type person_id *)

val get_person_id_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> person_id)
  (** Return a function that reads an untagged
      biniou value of type {!person_id}. *)

val read_person_id :
  Bi_inbuf.t -> person_id
  (** Input a tagged biniou value of type {!person_id}. *)

val person_id_of_string :
  ?pos:int -> string -> person_id
  (** Deserialize a biniou value of type {!person_id}.
      @param pos specifies the position where
      reading starts. Default: 0. *)

(* Writers for type gender *)

val gender_tag : Bi_io.node_tag
  (** Tag used by the writers for type {!gender}.
      Readers may support more than just this tag. *)

val write_untagged_gender :
  Bi_outbuf.t -> gender -> unit
  (** Output an untagged biniou value of type {!gender}. *)

val write_gender :
  Bi_outbuf.t -> gender -> unit
  (** Output a biniou value of type {!gender}. *)

val string_of_gender :
  ?len:int -> gender -> string
  (** Serialize a value of type {!gender} into
      a biniou string. *)

(* Readers for type gender *)

val get_gender_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> gender)
  (** Return a function that reads an untagged
      biniou value of type {!gender}. *)

val read_gender :
  Bi_inbuf.t -> gender

```

```

(** Input a tagged biniou value of type {!gender}. *)

val gender_of_string :
  ?pos:int -> string -> gender
  (** Deserialize a biniou value of type {!gender}.
    @param pos specifies the position where
    reading starts. Default: 0. *)

(* Writers for type person *)

val person_tag : Bi_io.node_tag
  (** Tag used by the writers for type {!person}.
    Readers may support more than just this tag. *)

val write_untagged_person :
  Bi_outbuf.t -> person -> unit
  (** Output an untagged biniou value of type {!person}. *)

val write_person :
  Bi_outbuf.t -> person -> unit
  (** Output a biniou value of type {!person}. *)

val string_of_person :
  ?len:int -> person -> string
  (** Serialize a value of type {!person} into
    a biniou string. *)

(* Readers for type person *)

val get_person_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> person)
  (** Return a function that reads an untagged
    biniou value of type {!person}. *)

val read_person :
  Bi_inbuf.t -> person
  (** Input a tagged biniou value of type {!person}. *)

val person_of_string :
  ?pos:int -> string -> person
  (** Deserialize a biniou value of type {!person}.
    @param pos specifies the position where
    reading starts. Default: 0. *)

(* Writers for type filiation_type *)

```

```

val filiation_type_tag : Bi_io.node_tag
  (** Tag used by the writers for type {!filiation_type}.
      Readers may support more than just this tag. *)

val write_untagged_filiation_type :
  Bi_outbuf.t -> filiation_type -> unit
  (** Output an untagged biniou value of type {!filiation_type}. *)

val write_filiation_type :
  Bi_outbuf.t -> filiation_type -> unit
  (** Output a biniou value of type {!filiation_type}. *)

val string_of_filiation_type :
  ?len:int -> filiation_type -> string
  (** Serialize a value of type {!filiation_type} into
      a biniou string. *)

(* Readers for type filiation_type *)

val get_filiation_type_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> filiation_type)
  (** Return a function that reads an untagged
      biniou value of type {!filiation_type}. *)

val read_filiation_type :
  Bi_inbuf.t -> filiation_type
  (** Input a tagged biniou value of type {!filiation_type}. *)

val filiation_type_of_string :
  ?pos:int -> string -> filiation_type
  (** Deserialize a biniou value of type {!filiation_type}.
      @param pos specifies the position where
      reading starts. Default: 0. *)

(* Writers for type filiation *)

val filiation_tag : Bi_io.node_tag
  (** Tag used by the writers for type {!filiation}.
      Readers may support more than just this tag. *)

val write_untagged_filiation :
  Bi_outbuf.t -> filiation -> unit
  (** Output an untagged biniou value of type {!filiation}. *)

val write_filiation :
  Bi_outbuf.t -> filiation -> unit

```



```

(** Output a biniou value of type {!filiation}. *)

val string_of_filiation :
  ?len:int -> filiation -> string
  (** Serialize a value of type {!filiation} into
      a biniou string. *)

(* Readers for type filiation *)

val get_filiation_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> filiation)
  (** Return a function that reads an untagged
      biniou value of type {!filiation}. *)

val read_filiation :
  Bi_inbuf.t -> filiation
  (** Input a tagged biniou value of type {!filiation}. *)

val filiation_of_string :
  ?pos:int -> string -> filiation
  (** Deserialize a biniou value of type {!filiation}.
      @param pos specifies the position where
      reading starts. Default: 0. *)

(* Writers for type tree *)

val tree_tag : Bi_io.node_tag
  (** Tag used by the writers for type {!tree}.
      Readers may support more than just this tag. *)

val write_untagged_tree :
  Bi_outbuf.t -> tree -> unit
  (** Output an untagged biniou value of type {!tree}. *)

val write_tree :
  Bi_outbuf.t -> tree -> unit
  (** Output a biniou value of type {!tree}. *)

val string_of_tree :
  ?len:int -> tree -> string
  (** Serialize a value of type {!tree} into
      a biniou string. *)

(* Readers for type tree *)

val get_tree_reader :

```

```
Bi_io.node_tag -> (Bi_inbuf.t -> tree)
(** Return a function that reads an untagged
    binou value of type {!tree}. *)

val read_tree :
  Bi_inbuf.t -> tree
(** Input a tagged binou value of type {!tree}. *)

val tree_of_string :
  ?pos:int -> string -> tree
(** Deserialize a binou value of type {!tree}.
    @param pos specifies the position where
    reading starts. Default: 0. *)
```

## 5 Library

A library named `atdgen` is installed by the standard installation process. Only a fraction of it is officially supported and documented. The documentation is available online at <http://oss.wink.com/atdgen/atdgen-1.3.1/odoc/index.html>.