`atdgen`
BINIOU AND JSON SERIALIZATION FOR OCAML
release 1.1.0

Martin Jambon

December 19, 2010

Contents

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 depends heavily on the following packages:

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

2 Command-line usage

2.1 Command-line help

```
$ atdgen -help
```

```
Generate OCaml serializers and deserializers.
Default serialization format is biniou.
Usage: ./atdgen FILE.atd
  -biniou
          Write serializers and deserializers for Biniou (default).
  -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
  -json
          Write serializers and deserializers for JSON.
  -nfd
          Do not dump OCaml function definitions
  -ntd
          Do not dump OCaml type definitions
  -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
  -open MODULE1,MODULE2,...
          List of modules to open (comma-separated or space-separated)
  -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
  -std-json
          Convert tuples and variants into standard JSON and
          refuse to print NaN and infinities (implying -json).
  -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 -biniou 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;
}
```

produces the interface file `example.mli` and the implementation file `example.ml`.

This is `example.mli`:

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


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

type gender = [ `Female | `Male ]

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
}

(* 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. *)

val create_date :
  year: int ->
  month: int ->
  day: int ->
  unit -> date
  (** Create a record of type {!date}. *)


(* 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 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. *)

val create_profile :
  id: string ->
  email: string ->
  ?email_validated: bool ->
  name: string ->
  ?real_name: string ->
  ?about_me: string list ->
  ?gender: gender ->
  ?date_of_birth: date ->
  unit -> profile
  (** Create a record of type {!profile}. *)
```

2.3 Atdgen-json example

```
$ atdgen -json 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;
}
```

produces the interface file `example.mli` and the implementation file `example.ml`. This is `example.mli`:

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

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

type gender = [ `Female | `Male ]

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
}

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.lexer_state -> Lexing.lexbuf -> date
  (** Input JSON data of type {!date}. *)

val date_of_string :
  string -> date
  (** Deserialize JSON data of type {!date}. *)

val create_date :
  year: int ->
  month: int ->
  day: int ->
  unit -> date
  (** Create a record of type {!date}. *)


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_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.lexer_state -> Lexing.lexbuf -> profile
  (** Input JSON data of type {!profile}. *)

val profile_of_string :
  string -> profile
  (** Deserialize JSON data of type {!profile}. *)

val create_profile :
  id: string ->
  email: string ->
  ?email_validated: bool ->
  name: string ->
  ?real_name: string ->
  ?about_me: string list ->
  ?gender: gender ->
```

```
?date_of_birth: date ->
unit -> profile
(** Create a record of type {!profile}. *)
```

3 Default mapping

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

ATD	OCaml	Biniou	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 instanciating 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 -json -std-json`) 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 -json -std-json`) 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` must be actually defined in another module. Reader and writer functions must be provided by opening one or several modules (`atdgen -json -open Foo,Bar`).

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;
}
```

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 <biniou 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`

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 ]
]
```

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.module`

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.

4.3.5 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;
}
```

4.3.6 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;
}
```

4.3.7 Field `ocaml.repr`

Integers

Position: after `int` type

Values: `char`, `int32`, `int64`

Semantics: specifies an alternate type for representing integers. The default type is `int`, but `char`, `int32` and `int64` can be used instead. These three types 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.

Example:

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

4.4 Section `ocaml_biniou`

Section `ocaml_biniou` takes precedence over section `ocaml` in `-biniou` mode for the following fields:

- `predef` (see 4.3.1)
- `module` (see 4.3.4)

4.5 Section `ocaml_json`

Section `ocaml_json` takes precedence over section `ocaml` in `-json` mode for the following fields:

- `predef` (see 4.3.1)
- `module` (see 4.3.4)

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 -biniou 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 -biniou genealogy.atd` into the following OCaml interface file `genealogy.mli` with ocamldoc-compliant comments:

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


(** Type definitions for family trees *)

(**
  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 = {
  genetic: bool option;
  pregnancy: bool option;
  raised_from_birth: bool option;
  raised: bool option;
  stepchild: bool option;
  adopted: bool option
}

(**
  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 = int

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

(** Gender, definition depending on the context *)
type gender = [ `F (** female *) | `M (** male *) ]

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

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

(* 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. *)

val create_filiation_type :
  ?genetic: bool ->
  ?pregnancy: bool ->
  ?raised_from_birth: bool ->
  ?raised: bool ->
  ?stepchild: bool ->
  ?adopted: bool ->
  unit -> filiation_type
  (** Create a record of type {!filiation_type}. *)


(* 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 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. *)

val create_filiation :
  parent: person_id ->
  child: person_id ->
  filiation_type: filiation_type ->
  unit -> filiation
  (** Create a record of type {!filiation}. *)


(* 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. *)

val create_person :
  person_id: person_id ->
  name: string ->
  ?gender: gender list ->
  ?biological_gender: gender ->
  unit -> person
  (** Create a record of type {!person}. *)


(* 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
      biniou value of type {!tree}. *)

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

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

val create_tree :
  members: person list ->
  filiations: filiation list ->
  unit -> tree
  (** Create a record of type {!tree}. *)
```

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-1.1.0/odoc/index.html`.