CDuce Programming Language
User's manual

Language Version 0.3.2+3

Table of Contents :

1 Compiler/interpreter/toplevel

1.1 Command-line

According to the command line arguments, the **cduce** command behaves either as an interactive toplevel, an interpreter, a compiler, or a loader.

- ```
  cduce [OPTIONS ...]  [--arg ARGUMENT ...]
  ```

  The command operates as an interactive toplevel. See the Toplevel section below.

- ```
  cduce [OPTIONS ...] [ script.cd | --stdin ] [--arg ARGUMENT ...]
  ```

  ```
  cduce [OPTIONS ...] --script script.cd [ ARGUMENT ...]
  ```

  The command runs the script **script.cd**.

- ```
  cduce [OPTIONS ...] --compile script.cd [--arg ARGUMENT ...]
  ```

  The command compiles the script **script.cd** and produces **script.cdo**. If the OCaml/CDuce interface is available and enabled, the compilers looks for a corresponding OCaml interface **script.cmi**. See the page for more information.

- ```
  cduce [OPTIONS ...] --run [ script.cd ... ] [--arg ARGUMENT ...]
  ```

  The command runs one or several pre-compiled scripts.

The arguments that follow the **--arg** option are the scripts' command line. They can be accessed within CDuce using the **argv** operator (of type **[] -> [String*]**).

The options and arguments are:

- **`--verbose`** (for **`--compile`** mode only). Display the type of values in the compiled unit.
- **`--obj-dir`** ***`directory`*** (for **`--compile`** mode only). Specify where to put the **`.cdo`** file (default: same directory as the source file).
- **`--I`** ***`directory`*** Add a directory to the search path for **`.cdo`** and **`.cmi`** files.
- **`--stdin`**. Read CDuce script from standard input.
- **`--no`** ***`feature`***. Disable one of the built-in optional features. The list of feature and their symbolic name can be obtained with the **`-v`** option. Can be used for instance to turn the Expat parser off, in order to use PXP, if both have been included at compile time.
- **`-v`**, **`--version`**. Show version information and built-in optional features, and exit.
- **`--license`**. Show license information and exit.
- **`--help`**. Show usage information about the command line.
- **`--noquery-optim`**. Do not optimize queries.

1.2 Scripting

CDuce can be used for writing scripts. As usual it suffices to start the script file by **`#!`*`install_dir`*`/cduce`** to call in a batch way the CDuce interpreter. The **`--script`** option can be used to avoid **`--arg`** when calling the script. Here is an example of a script file that prints all the titles of the filters of an Evolution mail client.

```
#!/usr/local/bin/cduce --script

type Filter = <filteroptions>[<ruleset> [(<rule>[<title>String _*])+]];;

let src : Latin1 =
  match argv [] with
    | [ f ] -> f
    | _ -> raise "Invalid command line"
in
let filter : Filter =
      match load_xml src with
       | x&Filter -> x
       | _ -> raise "Not a filter document"
in
print_xml(<filters>([filter]/<ruleset>_/<rule>_/<title>_)) ;;
```

1.3 Phrases

CDuce programs are sequences of phrases, which can be juxtaposed or separated by **`;;`**. There are several kinds of phrases:

- Types declarations **`type`** ***`T`*** **`=`** ***`t`***. Adjacent types declarations are mutually recursive, e.g.:

```
type T = <a>[ S* ]
type S = <b>[ T ]
```

- Function declarations **`let f ...`**. Adjacent function declarations are mutually recursive, e.g.:

```
let f (x : Int) : Int = g x
let g (x : Int) : Int  = x + 1
```

- Global bindings **`let p = e`** or **`let p : t = e`**.
- Evaluation statements (an expression to evaluate).
- Textual inclusion **`include "other_cduce_script.cd"`**; note that cycle of inclusion are detected and automatically broken. Filename are relative to the directory of the current file (or the current directory in the toplevel).
- Global namespace binding **`namespace p = "..."`** and global namespace default **`namespace "..."`** (see).
- Schema declaration **`schema name = "..."`** (see XML Schema).
- Alias for an external unit **`using alias = "unit"`** or **`using alias = unit`**: gives an alternative name for a pre-compiled unit. Values and types from **`unit.cdo`** can be referred to either as **`alias.ident`** or as **`unit.ident`**.

1.4 Toplevel

If no CDuce file is given on the command line, the interpreter behaves as an interactive toplevel.

Toplevel phrases are processed after each **`;;`**. Mutually recursive declarations of types or functions must be contained in a single adjacent sequence of phrases (without **`;;`** inbetween).

You can quit the toplevel with the toplevel directive **`#quit`** but also with either **`Ctrl-C`** or **`Ctrl-D`**. Another option is to use the built-in **`exit`**.

The toplevel directive **`#help`** prints an help message about the available toplevel directives.

The toplevel directive **`#env`** prints the current environment: the set of defined global types and values, and also the current sets of prefix-to-namespace bindings used for parsing (as defined by the user) and for pretty-printing (as computed by CDuce itself).

The two toplevel directives **`#silent`** and **`#verbose`** can be used to turn down and up toplevel outputs (results of typing and evaluation).

The toplevel directive **`#reinit_ns`** reinit the table of prefix-to-namespace bindings used for pretty-printing values and types with namespaces (see).

The toplevel directive **`#print_type`** shows a representationo of a CDuce type (including types imported from XML Schema documents).

The toplevel directive `#builtins` prints the name of embedded OCaml values (see).

The toplevel has no line editing facilities. You can use an external wrapper such as *ledit*.

1.5 Lexical entities

The **identifiers** (for variables, types, recursive patterns) are qualified names, in the sense of *XML Namespaces*. The chapter explains how to declare namespace prefixes in CDuce. Identifiers are resolved as XML attributes (which means that the default namespace does not apply).

The dot must be protected by a backslash in identifiers, to avoid ambiguity with the dot notation.

The dot notation serves several purposes:

- to refer to values and types declared in a separate CDuce compilation unit;
- to refer to values from OCaml compilation unit (see);
- to refer to schema components (see);
- to select a field from a record expression.

CDuce supports two style of **comments**: `(* ... *)` and `/* ... */`. The first style allows the programmer to put a piece a code apart. Nesting is allowed, and strings within simple or double quotes are not searched for the end-marker `*)`. In particular, simple quotes (and apostrophes) have to be balanced inside a `(* ... *)` comment. The other style `/* ... */` is more adapted to textual comments. They cannot be nested and quotes are not treated specially inside the comment.

2 Types and patterns

2.1 Types and patterns

In CDuce, a type denotes a set of values, and a pattern extracts sub-values from a value. Syntactically, types and patterns are very close. Indeed, any type can be seen as a pattern (which accepts any value and extracts nothing), and a pattern without any capture variable is nothing but a type.

Moreover, values also share a common syntax with types and patterns. This is motivated by the fact that basic and constructed values (that is, any values without functional values inside) are themselves singleton types. For instance `(1,2)` is both a value, a type and a pattern. As a type, it can be interpreted as a singleton type, or as a pair type made of two singleton types. As a pattern, it can be interpreted as a type constraint, or as a pair pattern of two type constraints.

In this page, we present all the types and patterns that CDuce recognizes. It is also the occasion to present the CDuce values themselves, the corresponding expression constructions, and fundamental operations on them.

2.2 Capture variables and default patterns

A value identifier inside a pattern behaves as a capture variable: it accepts and bind any value.

Another form of capture variable is the default value pattern `( x := c )` where *x* is a capture variable (that is, an identifier), and *c* is a scalar constant. The semantics of this pattern is to bind the capture variable to the constant, disregarding the matched value (and accepting any value).

Such a pattern is useful in conjunction with the first match policy (see below) to define "default cases". For instance, the pattern `((x & Int) | (x := 0), (y & Int) | (y := 0))` accepts any pair and bind `x` to the left component if it is an integer (and `0` otherwise), and similarly for `y` with the right component of the pair.

2.3 Boolean connectives

CDuce recognize the full set of boolean connectives, whose interpretation is purely set-theoretic.

- **`Empty`** denotes the empty type (no value).
- **`Any`** and `_` denote the universal type (all the values); the preferred notation is **`Any`** for types and `_` for patterns, but they are strictly equivalent.
- **`&`** is the conjunction boolean connective. The type ***`t1 & t2`*** has all the values that belongs to ***`t1`*** and to ***`t2`***. Similarly, the pattern ***`p1 & p2`*** accepts all the values accepted by both sub-patterns; a capture variable cannot appear on both side of this pattern.
- **`|`** is the disjunction boolean connective. The type ***`t1 | t2`*** has all the values that belongs either to ***`t1`*** or to ***`t2`***. Similarly, the pattern ***`p1 | p2`*** accepts all the values accepted by any of the two sub-patterns; if both match, the first match policy applies, and ***`p1`*** dictates how to capture sub-values. The two sub-patterns must have the same set of capture variables.
- **`\`** is the difference boolean connective. The left hand-side can be a type or a pattern, but the right-hand side is necessarily a type (no capture variable).

2.4 Recursive types and patterns

A set of mutually recursive types can be defined by toplevel type declarations, as in:

```
type T1 = <a>[ T2* ]
type T2 = <b>[ T1 T1 ]
```

It is also possible to use the syntax ***`T where T1 = t1 and ... and Tn = tn`*** where ***`T`*** and the ***`Ti`*** are type identifiers and the ***`ti`*** are type expressions. The same notation works for recursive patterns (for which there is no toplevel declarations).

There is an important restriction concerning recursive types: any cycle must cross a *type constructor* (pairs, records, XML elements, arrows). Boolean connectives do *not* count as type constructors! The code sample above is a correct definition. The one below is invalid, because there is an unguarded cycle between **`T`** and **`S`**.

```
type T = S | (S,S)  (* INVALID! *)
type S = T          (* INVALID! *)
```

2.5 Scalar types

CDuce has three kind of atomic (scalar) values: integers, characters, and atoms. To each kind corresponds a family of types.

- **Integers**. CDuce integers are arbitrarily large. An integer literal is a sequence of decimal digits, plus an optional leading unary minus (`-`) character.
  - `Int`: all the integers.
  - *`i--j`* (where *`i`* and *`j`* are integer literals, or `*` for infinity): integer interval. E.g.: `100--*`, `*--0`(*) (note that `*` stands both for plus and minus infinity).
  - *`i`* (where *`i`* is an integer literal): integer singleton type.
- **Floats**. CDuce provider minimal features for floats. The only way to construct a value of type `Float` is by the function `float_of : String -> Float`
- **Characters**. CDuce manipulates Unicode characters. A character literal is enclosed in single quotes, e.g. `'a', 'b', 'c'`. The single quote and the backslash character must be escaped by a backslash: `'\''`, `'\\'`. The double quote can also be escaped, but this is not mandatory. The usual `'\n', '\t', '\r'` are recognized. Arbitrary Unicode codepoints can be written in decimal `'\`*`i`*`;'` (*`i`* is an decimal integer; note that the code is ended by a semicolon) or in hexadecimal `'\x`*`i`*`;'`. Any other occurrence of a backslash character is prohibited.
  - `Char`: all the Unicode character set.
  - *`c--d`* (where *`d`* and *`d`* are character literals): interval of Unicode character set. E.g.: `'a'--'z'`.
  - *`c`* (where *`c`* is an integer literal): character singleton type.
  - `Byte`: all the Latin1 character set (equivalent to `'\0;'--'\255;'`).
- **Atoms**. Atoms are symbolic elements. They are used in particular to denote XML tag names, and also to simulate ML sum type constructors and exceptions names. An atomic is written `` ``*`xxx`* `` where *`xxx`* follows the rules for CDuce identifiers. E.g.: `` `yes, `No, `my-name ``. The atom `` `nil `` is used to denote empty sequences.
  - `Atom`: all the atoms.
  - *`a`* (where *`a`* is an atom literal): atom singleton type.
  - `Bool`: the two atoms `` `true `` and `` `false ``.
  - See also: .

2.6 Pairs

Pairs is a fundamental notion in CDuce, as they constitute a building block for sequence. Even if syntactic sugar somewhat hides pairs when you use sequences, it is good to know the existence of pairs.

A pair expression is written *`(e1,e2)`* where *`e1`* and *`e2`* are expressions.

Similarly, pair types and patterns are written *`(t1,t2)`* where *`t1`* and *`t2`* are types or patterns. E.g.: `(Int,Char)`.

When a capture variable *`x`* appears on both side of a pair pattern *`p = (p1,p2)`*, the semantics is the following one: when a value match *`p`*, if *`x`* is bound to *`v1`* by *`p1`* and to *`v2`* by *`p2`*, then *`x`* is bound to the pair *`(v1,v2)`* by *`p`*.

(*) Tuples are syntactic sugar for pairs. For instance, `(1,2,3,4)` denotes `(1,(2,(3,4)))`. You should be careful when putting a parenthesis around a type of the form *`-i`*, indeed, `*--i` would be parsed as a comment. You have to put a whitespace after the left parenthesis.

`(1,(2,(3,4)))`.

2.7 Sequences

Values and expressions

Sequences are fundamental in CDuce. They represents the content of XML elements, and also character strings. Actually, they are only syntactic sugar over pairs.

Sequences expressions are written inside square brackets; element are simply separated by whitespaces: `[ e1 e2 ... en ]`. Such an expression is syntactic sugar for: `(e1,(e2, ... (en,`nil) ...))`. E.g.: `[1 2 3 4]`.

The binary operator `@` denotes sequence concatenation. E.g.: `[ 1 2 3 ] @ [ 4 5 6 ]` evaluates to `[ 1 2 3 4 5 6 ]`.

It is possible to specify a terminator different from `` `nil ``; for instance `[ 1 2 3 4 ; q ]` denotes `(1,(2,(3,(4,q))))`, and is equivalent to `[ 1 2 3 4 ] @ q`.

Inside the square brackets of a sequence expression, it is possible to have elements of the form `! e` (which is not an expression by itself), where `e` is an expression which should evaluate to a sequence. The semantics is to "open" `e`. For instance: `[ 1 2 ![ 3 4 ] 5 ]` evaluates to `[ 1 2 3 4 5 ]`. Consequently, the concatenation of two sequences `e1 @ e2` can also be written `[ !e1 !e2 ]` or `[ !e1 ; e2 ]`.

Types and patterns

In CDuce, a sequence can be heterogeneous: the element can all have different types. Types and patterns for sequences are specified by regular expressions over types or patterns. The syntax is `[ R ]` where `R` is a regular expression, which can be:

- A type or a pattern, which correspond to a single element in the sequence (in particular, `[ _ ]` represents sequences of length 1, *not* arbitrary sequences).
- A juxtaposition of regular expression `R1 R2` which represents concatenation.
- A postfix repetition operator; the greedy operators are `R?`, `R+`, `R*`, and the ungreedy operators are: `R??`, `R+?`, `R*?`. For types, there is no distinction in semantics between greedy and ungreedy.
- A sequence capture variable `x::R` (only for patterns, of course). The semantics is to capture in `x` the subsequence matched by `R`. The same sequence capture variable can appear several times inside a regular expression, including under repetition operators; in that case, all the corresponding subsequences are concatenated together. Two instances of the same sequence capture variable cannot be nested, as in `[x :: (1 x :: Int)]`. Note the difference between `[ x::Int ]` and `[ (x & Int) ]`. Both accept sequences made of a single

integer, but the first one binds **x** to a sequence (of a single integer), whereas the second one binds it to the integer itself.

- Grouping **(*R*)**. E.g.: **[x::(Int Int) y]**.
- Tail predicate **/*p***. The type/pattern ***p*** applies to the current tail of the sequence (the subsequence starting at the current position). E.g.: **[(Int /(x:=1) | /(x:=2)) _*]** will bind **x** to **1** if the sequence starts with an integer and **2** otherwise.
- Repetition ***R* ** *n*** where ***n*** is a positive integer constant, which is just a shorthand for the concatenation of ***n*** copies of ***R***.

Sequence types and patterns also accepts the **[...; ...]** notation. This is a convenient way to discard the tail of a sequence in a pattern, e.g.: **[x::Int* ; _]**, which is equivalent to **[x::Int* _*]**.

It is possible to use the **@** operator (sequence concatenation) on types, including in recursive definitions. E.g.:

```
type t = [ <a>(t @ t) ? ]    (* [s?] where s=<a>[ s? s? ] *)

type x = [ Int* ]
type y = x @ [ Char* ]       (* [ Int* Char* ] *)

type t = [Int] @ t | []      (* [ Int* ] *)
```

however when used in recursive definitions **@** but must be right linear so for instance the following definition are not allowed:

```
type t = t @ [Int] | []      (* ERROR: Ill-formed concatenation loop *)
type t = t @ t               (* ERROR: Ill-formed concatenation loop *)
```

2.8 Strings

In CDuce, character strings are nothing but sequences of characters. The type **String** is pre-defined as **[Char*]**. This allows to use the full power of regular expression pattern matching with strings.

Inside a regular expression type or pattern, it is possible to use **PCDATA** instead of **Char*** (note that both are not types on their own, they only make sense inside square brackets, contrary to **String**).

The type **Latin1** is the subtype of **String** defined as **[Byte*]**; it denotes strings that can be represented in the ISO-8859-1 encoding, that is, strings made only of characters from the Latin1 character set.

Several consecutive characters literal in a sequence can be merged together between two single quotes: **['abc']** instead of **['a' 'b' 'c']**. Also it is possible to avoid square brackets by using double quotes: **"abc"**. The same

escaping rules applies inside double quotes, except that single quotes may be escaped (but must not), and double quotes must be.

2.9 Records

Records are set of finite (name,value) bindings. They are used in particular to represent XML attribute sets. Names are actually Qualified Names (see).

The syntax of a record expression is **`{ l1=e1; ...; ln=en }`** where the ***`li`*** are label names (same lexical conventions as for identifiers), and the ***`vi`*** are expressions. When an expression ***`ei`*** is simply a variable whose name match the field label ***`li`***, it is possible to omit it. E.g.: **`{ x; y = 10; z }`** is equivalent to **`{ x = x; y = 10; z = z }`**. The semi-colons between fields are optional.

They are two kinds of record types. Open record types are written **`{ l1=t1; ...; ln=tn; .. }`**, and closed record types are written **`{ l1 = t1; ...; ln = tn }`**. Both denote all the record values where the labels ***`li`*** are present and the associated values are in the corresponding type. The distinction is that that open type allow extra fields, whereas the closed type gives a strict enumeration of the possible fields. The semi-colon between fields is optional.

Additionally, both for open and close record types, it is possible to specify optional fields by using **`=?`** instead of **`=`** between a label and a type. For instance, **`{ x =? Int; y = Bool }`** represents records with a **`y`** field of type **`Bool`**, and an optional field **`y`** (that when it is present, has type **`Int`**), and no other field.

The syntax is the same for patterns. Note that capture variables cannot appear in an optional field. A common idiom is to bind default values to replace missing optinal fields: **`({ x = a } | (a := 1)) & { y = b }`**. A special syntax makes this idiom more convenient: **`{ x = a else (a:=1); y = b }`**.

As for record expressions, when the pattern is simply a capture variable whose name match the field label, it is possible to omit it. E.g.: **`{ x; y = b; z }`** is equivalent to **`{ x = x; y = b; z = z }`**.

The **`+`** operator (record concatenation, with priority given to the right argument in case of overlapping) is available on record types and patterns. This operator can be used to make a close record type/pattern open, or to add fields:

```
type t = { a=Int b=Char }
type s = t + {..}                (* { a=Int b=Char .. }
type u = s + { c=Float }         (* { a=Int b=Char c=Float .. } *)
type v = t + { c=Float }         (* { a=Int b=Char c=Float } *)
```

2.10 XML elements

In CDuce, the general of an XML element is **`<(tag) (attr)>content`** where **`tag`**, **`attr`** and **`content`** are three expressions. Usually, **`tag`** is a tag literal **`` `xxx ``**, and in this case, instead of writing **`<(`tag)>`**, you can write: **`<tag>`**. Similarly, when **`attr`** is a record literal, you can omit the surrounding **`({...})`**, and also the semicolon between attributes, E.g: **`<a href="http://..." dir="ltr">[]`**.

The syntax for XML elements types and patterns follows closely the syntax for expressions: **`<(tag) (attr)>content`** where **`tag`**, **`attr`** and **`content`** are three types or patterns. As for expressions, it is possible to simplify the notations for tags and attributes. For instance, **`<(`a) ({ href=String })>[]`** can be written: **`<a href=String>[]`**.

The following sample shows several way to write XML types.

```
type A = <a x=String y=String ..>[ A* ]
type B = <(`x | `y) ..>[ ]
type C = <c x = String; y = String>[ ]
type U = { x = String y =? String ..}
type V = [ W* ]
type W = <v (U)>V
```

2.11 Functions

CDuce is an higher-order functional languages: functions are first-class citizen values, and can be passed as argument or returned as result, stored in data structure, etc...

A functional type has the form **`t -> s`** where **`t`** and **`s`** are types. Intuitively, this type corresponds to functions that accept (at least) any argument of type **`t`**, and for such an argument, returns a value of type **`s`**. For instance, the type **`(Int,Int) -> Int & (Char,Char) -> Char`** denotes functions that maps any pair of integer to an integer, and any pair of characters to a character.

The explanation above gives the intuition behind the interpretation of functional types. It is sufficient to understand which subtyping relations and equivalences hold between (boolean combination) of functional types. For instance, **`Int -> Int & Char -> Char`** is a subtype of **`(Int|Char) -> (Int|Char)`** because with the intuition above, a function of the first type, when given a value of type **`Int|Char`** returns a value of type **`Int`** or of type **`Char`** (depending on the argument).

Formally, the type **`t -> s`** denotes CDuce abstractions **`fun (t1 -> s1; ...; tn -> sn)...`** such that **`t1 -> s1 & ... & tn -> sn`** is a subtype of **`t -> s`**.

Functional types have no counterpart in patterns.

2.12 References

References are mutable memory cells. CDuce has no built-in reference type. Instead, references are implemented in an object-oriented way. The type `ref T` denotes references of values of type *T*. It is only syntactic sugar for the type `{ get = [] -> T ; set = T -> [] }`.

2.13 OCaml abstract types

The notation `!t` is used by the CDuce/OCaml interface to denote the OCaml abstract type `t`.

2.14 Complete syntax

Below we give the complete syntax of type and pattern, the former being patterns without capture variables

3 Expressions

3.1 Value constructors expressions

The page presents the different kind of values: scalar constant (integers, characters, atoms), structured values (pairs, records, sequences, XML elements), and functional values (abstractions). Value themselves are expressions, and the value constructors for structured values operate also on expressions.

This page presents the other kinds of expressions in the language.

3.2 Pattern matching

A fundamental operation in CDuce is pattern matching:

```
match e with
 | p1 -> e1
...
 | pn -> en
```

The first vertical bar **|** can be omitted. The semantics is to try to match the result of the evaluation of ***e*** successively with each pattern ***pi***. The first matching pattern triggers the corresponding expression in the right hand side, which can use the variables bound by the pattern. Note that a first match policy, as for the disjunction patterns.

The static type system ensures that the pattern matching is exhaustive: the type computed for ***e*** must be a subtype of the union of the types accepted by all the patterns.

Local definition is a lighter notation for a pattern matching with a single branch:

```
let p = e1 in e2
```

is equivalent to:

```
match e1 with p -> e2
```

Note that the pattern ***p*** need not be a simple capture variable.

3.3 Functions

Abstraction

The general form for a function expression is:

```
fun f (t1 -> s1; ...; tn -> sn)
 | p1 -> e1
...
 | pn -> en
```

The first line is the *interface* of the function, and the remaining is the *body*, which is a form of pattern matching (the first vertical bar **|** can thus be omitted).

The identifier ***f*** is optional; it is useful to define a recursive function (the body of the function can use this identifier to refer to the function itself).

The interface of the function specifies some constraints on the behavior of the function. Namely, when the function receive an argument of type, say ***ti***, the result (if any) must be of type ***si***. The type system ensures this property by type-checking the body once for each constraint.

The function operate by pattern-matching the argument (which is a value) exactly as for standard pattern matching. Actually, it is always possible to add a line **`x -> match x with`** between the interface and the body without changing the semantics.

When there is a single constraint in the interface, there is an alternative notation, which is lighter for several arguments (that is, when the argument is a tuple):

```
fun f (p1 : t1, ..., pn : tn) : s = e
```

(note the blank spaces around the colons which are mandatory when the pattern is a variable (*)) which is strictly equivalent to:

(*) The reason why the blank spaces are mandatory with variables is that the XML recommendation allows colons to occur in variables ("names" in XML terminology: see section on), so the blanks disambiguate the variables. Actually only the blank on the right hand side is necessary: CDuce accepts **`fun f (x1 :t1, ..., xn :tn):s = e`**, as well (see also *this paragraph* on **`let`** declarations in the tutorial).

```
fun f ((t1,...,tn) -> s) (p1,...,pn) -> e
```

It is also possible to define currified functions with this syntax:

```
fun f (p1 : t1, ..., pn : tn) (q1 : s1, ..., qm : sm) ... : s = e
```

which is strictly equivalent to:

```
fun f ((t1,...,tn) -> (s1,...,sm) -> ... -> s)
 (p1,...,pn) ->
  fun ((s1,...,sm) -> ... -> s)
   (q1,...,qm) ->
     ...
     e
```

The standard notation for local binding a function is:

```
let f = fun g (...) ... in ...
```

Here, ***f*** is the "external" name for the function, and ***g*** is the "internal" name (used when the function needs to call itself recursively, for instance). When the two names coincide (or when you don't need an internal name), there are lighter notations:

```
let fun f (...) ... in ...
let f (...) ... in ...
```

Application

The only way to use a function is ultimately to apply it to an argument. The notation is simply a juxtaposition of the function and its argument. E.g.:

```
(fun f (x : Int) : Int = x + 1) 10
```

evaluates to 11. The static type system ensures that applications cannot fail.

Note that even if there is no functional "pattern" in CDuce, it is possible to use in a pattern a type constraint with a functional type, as in:

```
fun (Any -> Int)
 | f & (Int -> Int) -> f 5
 | x & Int -> x
 | _ -> 0
```

3.4 Exceptions

The following construction raises an exception:

```
raise e
```

The result of the evaluation of **e** is the *argument* of the exception.

It is possible to catch an exception with an exception handler:

```
try e with
 | p1 -> e1
...
 | pn -> en
```

Whenever the evaluation of **e** raises an exception, the handler tries to match the argument of the exception with the patterns (following a first-match policy). If no pattern matches, the exception is propagated.

Note that contrary to ML, there is no exception name: the only information carried by the exception is its argument. Consequently, it is the responsibility of the programmer to put enough information in the argument to recognize the correct exceptions. Note also that a branch **(`A,x) -> e** in an exception handler gives no static information about the capture variable **x** (its type is **Any**). **Note:** it is possible that the support for exceptions will change in the future to match ML-like named exceptions.

3.5 Record operators

There are three kinds of operators on records:

- Field projection:

  ```
  e.l
  ```

  where **l** is the name of a label which must be present in the result of the evaluation of **e**. This construction is equivalent to: **match e with { l = x } -> x**. It is necessary to put whitespace between the expression and the dot when the expression is an identifier.
- Record concatenation:

  ```
  e1 + e2
  ```

  The two expressions must evaluate to records, which are merged together. If both have a field with the same name, the one on the right have precedence. Note that the operator **+** is overloaded: it also operates on integers.
- Field suppression:

```
e \ l
```

deletes the field **`l`** in the record resulting from the evaluation of **`e`** whenever it is present.

3.6 Arithmetic operators

Binary arithmetic operators on integers: **`+,-,*,div,mod`**. Note that **`/`** is used for projection and *not* for division.

The operator **`+,-`** and **`*`** are typed using simple interval arithmetic. The operators **`div`** and **`mod`** produce a warning at compile type if the type of there second argument include the integer **`0`**.

The type **`Float`** represents floating point numbers. An operator **`float_of: String -> Float`** is provided to create values of this type. Currently, no other operator are provided for this type (but you can use OCaml functions to work on floats).

3.7 Generic comparisons, if-then-else

Binary comparison operators (returns booleans): **`=,<<,<=,>>,>=`**. Note that **`<`** is used for XML elements and is this not available for comparison.

The semantics of the comparison is not specified when the values contain functions. Otherwise, the comparison gives a total ordering on CDuce values. The result type for all the comparison operators is **`Bool`**, except for equality when the arguments are known statically to be different (their types are disjoint); in this case, the result type is the singleton **`` `false ``**.

The if-then-else construction is standard:

```
if e1 then e2 else e3
```

and is equivalent to:

```
match e1 with `true -> e2 | `false -> e3
```

Note that the else-clause is mandatory.

The infix operators **`||`** and **`&&`** denote respectively the logical or and the logical and. The prefix operator **`not`** denotes the logical negation.

3.8 Upward coercions

It is possible to "forget" that an expression has a precise type, and give it a super-type:

```
(e : t)
```

The type of this expression if ***t***, and ***e*** must provably have this type (it can have a subtype). This "upward coercion" can be combined with the local let binding:

```
let p : t = e in ...
```

which is equivalent to:

```
let p = (e : t) in ...
```

Note that the upward coercion allows earlier detection of type errors, better localization in the program, and more informative messages.

CDuce also have a dynamic type-check construction:

```
(e :? t)
let p :? t = e in ...
```

If the value resulting from the evaluation of ***e*** does not have type ***t***, an exception whose argument (of type **Latin1**) explains the reason of the mismatch is raised.

3.9 Sequences

The concatenation operator is written **@**. There is also a **flatten** operator which takes a sequence of sequences and returns their concatenation.

There are two built-in constructions to iterate over a sequence. Both have a very precise typing which takes into account the position of elements in the input sequence as given by its static type. The **map** construction is:

```
map e with
  | p1 -> e1
...
  | pn -> en
```

Note the syntactic similarity with pattern matching. Actually, **map** is a pattern

matching form, where the branches are applied in turn to each element of the input sequence (the result of the evaluation of ***e***). The semantics is to return a sequence of the same length, where each element in the input sequence is replaced by the result of the matching branch.

Contrary to **map**, the **transform** construction can return a sequence of a different length. This is achieved by letting each branch return a sequence instead of a single element. The syntax is:

```
transform e with
 | p1 -> e1
...
 | pn -> en
```

There is always an implicit default branch **_ -> []** at then end of **transform**, which means that unmatched elements of the input sequence are simply discarded.

Note that **map** can be simulated by **transform** by replacing each expression ***ei*** with **[*ei*]**.

Conversely, **transform** can be simulated by **map** by using the **flatten** operator. Indeed, we can rewrite **transform *e* with ...** as **flatten (map *e* with ... | _ -> [])**.

3.10 XML-specific constructions

Loading XML documents

The **load_xml: Latin1 -> AnyXml** built-in function parses an XML document on the local file system. The argument is the filename. The result type **AnyXml** is defined as:

```
type AnyXml = <(Atom) (Record)>[ (AnyXml|Char)* ]
```

If the support for netclient or curl is available, it is also possible to fetch an XML file from an URL, e.g.: **load_xml "http://..."**. A special scheme **string:** is always supported: the string following the scheme is parsed as it is.

There is also a **load_html: Latin1 -> [Any*]** built-in function to parse in a permissive way HTML documents.

Pretty-printing XML documents

Two built-in functions can be used to produce a string from an XML document:

```
```

```
print_xml: Any -> Latin1
print_xml_utf8: Any -> String
```

They fail if the argument is not an XML document (this isn't checked statically). The first operator **print_xml** prepares the document to be dumped to a ISO-8859-1 encoded XML file: Unicode characters outside Latin1 are escaped accordingly, and the operator fails if the document contains tag or attribute names which cannot be represented in ISO-8859-1. The second operator **print_xml_utf8** always succeed but produces a string suitable for being dumped in an UTF-8 encoded file. See the variants of the **dump_to_file** operator in the section on Input/output.

In both cases, the resulting string does *not* contain the XML prefix "<?xml ...>".

Projection

The projection takes a sequence of XML elements and returns the concatenation of all their children with a given type. The syntax is:

```
e/t
```

which is equivalent to:

```
transform e with <_>[ (x::t | _)* ] -> x
```

For instance, the expression **[<a>[<x>"A" <y>"B"] <b>[<y>"C" <x>"D"]] / <x>_** evaluates to **[<x>"A" <x>"D"]** .

There is another form of projection to extract attributes:

```
e/@l
```

which is equivalent to:

```
transform e with <_ l=l>_ -> l
```

The dot notation can also be used to extract the value of the attribute for one XML element:

```
# <a x=3>[].x;;
- : 3 = 3
```

Iteration over XML trees

Another XML-specific construction is **xtransform** which is a generalization of **transform** to XML trees:

```
xtransform e with
 | p1 -> e1
...
 | pn -> en
```

Here, when an XML elements in the input sequence is not matched by a pattern, the element is copied except that the transformation is applied recursively to its content. Elements in the input sequence which are not matched and are not XML elements are copied verbatim.

3.11 Unicode Strings

Strings are nothing but sequences of characters, but in view of their importance when dealing with XML we introduced the standard double quote notation. So **`[ 'F' 'r' 'a' 'n' 'ç' 'e' ]`** can be written as **`"France"`**. In double quote all the *values* of type **`Char`** can be used: so besides Unicode chars we can also double-quote codepoint-defined characters (**`\x`*`h`*`;`** **`\`*`d`*`;`** where ***`h`*** and ***`d`*** are hexadecimal and decimal integers respectively), and backslash-escaped characters (**`\t`** tab, **`\n`** newline, **`\r`** return, **`\\`** backslash). Instead we cannot use character expressions that are not values. For instance, for characters there is the built-in function **`char_of_int : Int -> Char`** which returns the character corresponding to the given Unicode codepoint (or raises an exception for a non-existent codepoint), and this can only be used with the regular sequence notation, thus **`"France"`**, **`"Fran"@[(char_of_int 231)]@"e"`**, and **`"Fran\231;e"`** are equivalent expressions.

3.12 Converting to and from string

Pretty-printing a value

The built-in function **`string_of: Any -> Latin1`** converts any value to a string, using the same pretty-printing function as the CDuce interpreter itself.

Creating and decomposing atoms from strings

The built-in functions **`split_atom: Atom -> (String,String)`** and **`make_atom: (String,String) -> Atom`** converts between atoms and pair of strings (namespace,local name).

Creating integers from strings

The operator **`int_of`** converts a string to an integer. The string is read in decimal (by default) or in hexadecimal (if it begins with **`0x`** or **`0X`**), octal (if it begins with **`0o`** or **`0O`**), or binary (if it begins with **`0b`** or **`0B`**). It fails if the string is not a decimal representation of an integer or if in the case of hexadecimal, octal, and binary representation the

integer cannot be contained in 64 bits. There is a type-checking warning when the argument cannot be proved to be of type `[ '-'? '0'--'9'+ ] | ['-'? 'O'('b'|'B') '0'--'1'+ ] | ['-'? 'O'('o'|'O') '0'--'7'+ ] | ['-'? 'O'('x'|'X') ('0'--'9'|'a'--'f'|'A'--'F')+]` .

Creating strings from integers

Besides the built-in function `string_of: Any -> Latin1`, it is also possible to create characters, hence strings, from their codepoints: either by enclosing their code within a backslash (`\x` for hexadecimal code) and a semicolon, or by applying the built-in function `char_of_int : Int -> Char`.

3.13 Input-output

Displaying a string

To print a string to standard output, you can use one of the built-in function `print: Latin1 -> []` or `print_utf8: String -> []`.

Loading files

There are two built-in functions available to load a file into a CDuce string:

```
load_file: Latin1 -> Latin1
load_file_utf8: Latin1 -> String
```

The first one loads an ISO-8859-1 encoded file, whereas the second one loads a UTF-8 encoded file.

If the support for netclient or curl is available, it is also possible to fetch a file from an URL, e.g.: `load_file "http://..."`.

Dumping to files

There are two operators available to dump a CDuce string to a file:

```
dump_to_file e1 e2
dump_to_file_utf8 e1 e2
```

The first one creates an ISO-8859-1 encoded file (it fails when the CDuce string contains non Latin1 characters), whereas the second one creates a UTF-8 encoded file. In both cases, the first argument is the filename and the second one is the string to dump.

3.14 System

Running external commands

The predefined function **system** executes an external command (passed to **/bin/sh**) and returns its standard output and standard error channels and its exit code. The type for **system** is:

```
Latin1 -> { stdout = Latin1; stderr = Latin1;
            status = (`exited,Int) | (`stopped,Int) | (`signaled,Int) |}
```

Terminating the program

The predefined function **exit: 0--255 -> Empty** terminates the current process. The argument is the exit code.

Accessing the environment

The built-in function **getenv: Latin1 -> Latin1** queries the system environment for an environment variable. If the argument does not refer to an existing variable, the function raises the exception **`Not_found**.

Command line arguments

The built-in function **argv: [] -> [String*]** returns the sequence of command line arguments given to the current program.

3.15 Namespaces

It is possible in expression position to define a local prefix-namespace binding or to set a local default namespace.

```
namespace p = "..." in e
namespace "..." in e
```

See for more details.

3.16 Imperative features

The construction **ref *T* e** is used to build a *reference* initialized with the result of the expression ***e***; later, the reference can receive any value of type ***T***. The reference is actually a value of type **{ get = [] -> T ; set = T -> [] }**.

Two syntactic sugar constructions are provided to facilitate the use of references:

```
!e          ===  e.get []          Dereferencing
e1 := e2    ===  e1.set e2         Assignment
```

An expression of type **[]** is often considered as a command and followed by another expression. The sequencing operator gives a syntax for that:

```
e1 ; e2   ===  let [] = e1 in e2   Sequencing
```

3.17 Queries

CDuce is endowed with a **select_from_where** syntax to perform some SQL-like queries. The general form of select expressions is

```
select e from
   p1 in e1,
   p2 in e2,
        :
   pn in en
where c
```

where ***e*** is an expression, ***c*** a boolean expression, the ***pi***'s are patterns, and the ***ei***'s are sequence expressions.

It works exactly as a standard SQL select expression, with the difference that relations (that is sequences of tuples) after the **in** keyword can here be generic sequences, and before the **in** generic patterns instead of just capture variables can be used. So the result is the sequence of all values obtained by calculating ***e*** in the sequence of environments in which the free variables of ***e*** are bounded by iteratively matching each pattern ***pi*** with every element of the sequence ***ei***, provided that the condition ***c*** is satisfied. In other words, the first element of the result is obtained by calculating ***e*** in the environment obtained by matching ***p1*** against the first element of ***e1***, ***p2*** against the first element of ***e2***, ... , and ***pn*** against the first element of ***en***; the second element of the result is obtained by calculating ***e*** in the environment obtained by matching ***p1*** against the first element of ***e1***, ***p2*** against the first element of ***e2***, ..., and ***pn*** against the *second* element of ***en***, ... ; and so on.

Formally, the semantics of the select expression above is defined as:

```
transform e1 with p1 ->
   transform e2 with p2 ->
        ...
      transform en with pn ->
         if c then  [e] else []
```

A `select` expression works like a set of nested `transform` expressions. The advantage of using select rather than transform is that queries are automatically optimized by applying classical logic SQL optimization techniques (this automatic optimization can be disabled).

The built-in optimizer is free to move boolean conditions around to evaluate them as soon as possible. A warning is issued if a condition does not depend on any of the variables captured by the patterns.

4 XML Namespaces

4.1 Overview

CDuce fully implements the W3C *XML Namespaces* Recommendation. Atom names (hence XML element tags) and record labels (hence XML attribute names) are logically composed of a namespace URI and a local part. Syntactically, they are written as *qualified names*, conforming to the QName production of the Recommendation:

```
QName     ::= (Prefix ':')? LocalPart
Prefix    ::= NCName
LocalPart ::= NCName
```

The prefix in a QName must be bound to a namespace URI. In XML, the bindings from prefixes to namespace URIs are introduction through special **`xmlns:prefix`** attributes. In CDuce, instead, there are explicit namespace binders. For instance, the following XML documents

```
<p:a q:c="3" xmlns:p="http://a.com" xmlns:q="http://b.com"/>
```

can be written in CDuce:

```
namespace p = "http://a.com" in
namespace q = "http://b.com" in
<p:a q:c="3">[]
```

This element can be bound to a variable **`x`** by a **`let`** binding as follows:

```
let x =
  namespace p = "http://a.com" in
  namespace q = "http://b.com" in
  <p:a q:c="3">[]
```

In which case the namespace declarations are local to the scope of the let. Alternatively, it is possible to use global prefix bindings:

```
namespace p = "http://a.com"
namespace q = "http://b.com"
let x = <p:a q:c="3">[]
```

Similarly, CDuce supports namespace *defaulting*. This is introduced by a local or global **`namespace "..."`** construction. As in the XML, default namespace applies only to tags (atoms), not attributes (record labels). For instance, in the expression **`namespace "A" in <x y="3">[]`**, the namespace for the element tag is "A", and the attribute has no namespace.

The toplevel directive **`#env`** causes CDuce to print, among others, the current set of global bindings.

4.2 Types for atoms

The type **`Atom`** represents all the atoms, in all the namespaces. An underscore in tag position (as in **`<_>[]`**) stands for this type.

Each atom constitutes a subtype of **`Atom`**. In addition to these singelton types, there are the ``any in namespace'' subtypes, written: **`p:*`** where **`p`** is a namespace prefix; this type has all the atoms in the namespace denoted by **`p`**. The token **`.:*`** represents all the atoms in the current default namespace.

When used as atoms and not tags, the singleton types and ``any in namespace'' types must be prefixed by a backquote, as for atom values: **`` `p:x, `p:*, `.:* ``**.

4.3 Printing XML documents

The **`print_xml`** and **`print_xml_utf8`** operators produce a string representation of an XML document. They have to assign prefixes to namespace. In the current implementation, CDuce produces XML documents with no default namespace and only toplevel prefix bindings (that is, **`xmlns:p="..."`** attributes are only produced for the root element). Prefix names are chosen using several heuristics. First, CDuce tries using the prefixes bound in the scope of the **`print_xml`** operator. When this is not possible, it uses global ``hints'': each time a prefix binding is encountered (in the CDuce program or in loaded XML documents), it creates a global hint for the namespace. Finally, it generates fresh prefixes of the form **`ns`*`n`*** where ***`n`*** is an integer. For instance, consider the expression:

```
print_xml (namespace "A" in <a>[])
```

As there is no available name the prefix URI "A", CDuce generates a fresh prefix and produces the following XML documents:

```
<ns1:a xmlns:ns1="A"/>
```

Now consider this expression:

```
print_xml (namespace p = "A" in <p:a>[])
```

CDuce produces:

```
<p:a xmlns:p="A"/>
```

In this case, the prefix binding for the namespace "A" is not in the scope of **`print_xml`**, but the name **`p`** is available as a global hint. Finally, consider:

```
namespace q = "A" in print_xml (namespace p = "A" in <p:a>[])
```

Here, the prefix **`q`** is available in the scope of the **`print_xml`**. So it is used in priority:

```
<q:a xmlns:q="A"/>
```

As a final example, consider the following expression:

```
print_xml (namespace p ="A" in <p:a>[ (namespace p = "B" in <p:a>[]) ])
```

A single name **`p`** is available for both namespaces "A" and "B". CDuce choses to assign it to "A", and it generates a fresh name for "B", so as to produce:

```
<p:a xmlns:ns1="B" xmlns:p="A"><ns1:a/></p:a>
```

Note that the fresh names are ``local" to an application of **`print_xml`**. Several application of **`print_xml`** will re-use the same names **`ns1, ns2`**, ...

4.4 Pretty-printing of XML values and types

The CDuce interpreter and toplevel uses an algorithm similar to the one mentioned in the previous section to pretty-print CDuce values and types that involve namespace.

The main difference is that it does *not* use by default the current set of prefix bindings. The rationale is that this set can change and this would make it difficult to understand the output of CDuce. So only global hints are used to produce prefixes. Once a prefix has been allocated, it is not re-used for another namespace. The toplevel directive **`#env`** causes CDuce to print, amongst other, the table of prefixes

used for pretty-printing. It is possible to reinitialize this table with the directive **#reinit_ns**. This directive also set the current set if prefix bindings as a primary source of hints for assigning prefixes for pretty-printing in the future.

4.5 Accessing namespace bindings

CDuce encourages a processing model where namespace prefixes are just considered as macros (for namespaces) which are resolved by the (CDuce or XML) parser. However, some XML specifications requires the application to keep for each XML element the set of locally visible bindings from prefixes to namespaces. CDuce provides some support for that.

Even if this is not reflected in the type system, CDuce can optionally attach to any XML element a table of namespace bindings. The following built-in functions allows the programmer to explictly access this information:

```
type Namespaces = [ (String,String)* ]
namespaces: AnyXml -> Namespaces
set_namespaces: Namespaces -> AnyXml -> AnyXml
```

The **namespaces** function raises an exception when its argument has no namespace information attached.

When XML elements are generated, either as literals in the CDuce code or by **load_xml**, it is possible to tell CDuce to remember in-scope namespace bindings. This can be done with the following construction:

```
namespace on in e
```

The XML elements built within ***e*** (including by calling **load_xml**) will be annotated. There is a similar **namespace off** construction to turn off this mecanism in a sub-expression, and bothe constructions can be used at top-level.

```
# namespace cduce = "CDUCE";;
# namespaces <cduce:a>[];;
Uncaught CDuce exception: [ `Invalid_argument 'namespaces' ]

# namespace on;;
# namespaces <cduce:a>[];;
- : Namespaces = [ [ "xsd" 'http://www.w3.org/2001/XMLSchema' ]
                   [ "xsi" 'http://www.w3.org/2001/XMLSchema-instance' ]
                   [ "cduce" 'CDUCE' ]
                   ]
# namespaces (load_xml "string:<a xmlns='xxx'/>");;
- : Namespaces = [ [ "" 'xxx' ] ]
```

The default binding for the prefix **xml** never appear in the result of **namespaces**.

The `xtransform` iterator does not change the attached namespace information for XML elements which are just traversed. The generic comparison operator cannot distinguish two XML elements which only differ by the attached namespace information.

4.6 Miscellaneous

Contrary to the W3C *Namespaces in XML 1.1* Candidate Recommendation, a CDuce declaration `namespace p = ""` does *not* undeclare the prefix `p`. Instead, it binds it to the null namespace (that is, a QName using this prefix is interpreted as having no namespace).

5 XML Schema

5.1 Overview

CDuce partially supports *XML Schema* Recommendations (*Primer*, *Structures*, *Datatypes*). Using this CDuce feature it is possible to manipulate XML documents whose leaves are typed values like integers, dates, binary data, and so on.

CDuce supports XML Schema by implementing the following features:

- XML Schema components import
- XML Schema validation
- XML Schema instances output

This manual page describes how to use these features in CDuce, all the documents used in the examples are available in the manual section: XML Schema sample documents.

> **Note:** The support for XML Schema does not currently interact well with separate compilation. When a CDuce unit `script.cd` which uses an XML Schema is compiled, the resulting `script.cdo` object refers to the XML Schema by name. That is, when these units are run, the XML Schema must still be available from the current directory and must not have been changed since compilation.

5.2 XML Schema components (micro) introduction

An XML Schema document could define four different kinds of component, each of them could be imported in CDuce and used as CDuce types:

- **Type definitions** A type definition defines either a simple type or a complex type. The former could be used to type more precisely the string content of an element. You can think at it as a refinement of #PCDATA. XML Schema provides a set of *predefined simple types* and a way to define new simple types. The latter could be used to constraint the content model and the attributes of an XML element. An XML Schema complex type is strictly more expressive than a DTD element declaration.
- **Element declarations** An element declaration links an attribute name to a complex type. Optionally, if the type is a simple type, it can constraints the set of possible values for the element mandating a fixed value or providing a default value.
- **Attribute group definitions** An attribute group definitions links a set of attribute declarations to a name which can be referenced from other XML Schema components.
- **Model group definitions** A model group definition links a name to a constraint over the complex content of an XML element. The linked name can be referenced from other XML Schema components.

Attribute declaration currently don't produce any CDuce type and can't be used for validation themselves.

5.3 XML Schema components import

In order to import XML Schema components in CDuce, you first need to tell CDuce to import an XML Schema document. You can do this using the **schema** keyword to bind an uppercase identifier to a local schema document:

```
# schema Mails = "tests/schema/mails.xsd";;
Registering schema type: attachmentType
Registering schema type: mimeTopLevelType
Registering schema type: mailsType
Registering schema type: mailType
Registering schema type: bodyType
Registering schema type: envelopeType
Registering schema element: header
Registering schema element: Date
Registering schema element: mails
Registering schema attribute group: mimeTypeAttributes
Registering schema model group: attachmentContent
```

The above declaration will (try to) import all schema components included in the schema document mails.xsd as CDuce types. You can reference them using the dot operator, e.g. **S.mails**.

XML Schema permits ambiguity in components name. CDuce chooses to resolve references to Schema components in this order: elements, types, model groups, attribute group.

The result of a schema component reference is an ordinary CDuce type which you

can use as usual in function definitions, pattern matching and so on.

```
let is_valid_mail (Any -> Bool)
  | Mails.mailType -> `true
  | _ -> `false
```

Correctness remark: *while parsing XML Schema documents, CDuce assumes that they're correct with respect to XML Schema recommendations. At minimum they're required to be valid with respect to XML Schema for Schemas. It's recommended that you will check for validity your schemas before importing them in CDuce, strange behaviour is assured otherwise.*

5.4 Toplevel directives

The toplevel directive **`#env`** supports schemas, it lists the currently defined schemas.

The toplevel directive **`#print_type`** supports schemas too, it can be used to print types corresponding to schema components.

```
# #print_type Mails.bodyType;;
[ Char* ]
```

For more information have a look at the manual section about toplevel directives.

5.5 XML Schema # CDuce mapping

- XML Schema **predefined simple types** are mapped to CDuce types directly in the CDuce implementation preserving as most as possible XML Schema constraints. The table below lists the most significant mappings.

XML Schema predefined simple type	CDuce type
`duration`, **`dateTime`**, **`time`**, **`date`**, **`gYear`**, **`gMonth`**, ...	closed record types with some of the following fields (depending on the Schema type): **`year`**, **`month`**, **`day`**, **`hour`**, **`minute`**, **`second`**, **`timezone`**
`boolean`	**`Bool`**
`anySimpleType`, **`string`**, **`base64Binary`**, **`hexBinary`**, **`anyURI`**	**`String`**

integer	**Int**
nonPositiveInteger, **negativeInteger**, **nonNegativeInteger**, **positiveInteger**, **long**, **int**, **short**, **byte**	integer intervals with the appropriate limits
string, **normalizedString**, and the other types derived (directly or indirectly) by restriction from string	String
NMTOKENS, **IDREFS**, **ENTITIES**	**[String*]**
decimal,**float**,**double**	**Float**
(**Not properly supported**)**decimal**, **float**, **double**, **NOTATION**, **QName**	**String**

Simple type definitions are built from the above types following the XML Schema derivation rules.

- XML Schema **complex type definitions** are mapped to CDuce types representing XML elements which can have any tag, but whose attributes and content are constrained to be valid with respect to the original complex type.

  As an example, the following XML Schema complex type (a simplified version of the homonymous **envelopeType** defined in mails.xsd):

```
<xsd:complexType name="envelopeType">
 <xsd:sequence>
  <xsd:element name="From" type="xsd:string"/>
  <xsd:element name="To" type="xsd:string"/>
  <xsd:element name="Date" type="xsd:dateTime"/>
  <xsd:element name="Subject" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

  will be mapped to an XML CDuce type which must have a `From` attribute of type String and four children. Among them the `Date` children must be an XML element containing a record which represents a `dateTime` Schema type.

```
# #print_type Mails.envelopeType;;
<(Any)>[
  <From>String
  <To>String
  <Date>{
    positive = Bool;
    year = Int; month = Int; day = Int;
    hour = Int; minute = Int; second = Int;
    timezone =? { positive = Bool; hour = Int; minute = Int }
  }
  <Subject}>String
```

```
]
```

- XML Schema **element declarations** can bound an XML element either to a complex type or to a simple type. In the former case the conversion is almost identical as what we have seen for complex type conversion. The only difference is that this time element's tag must correspond to the name of the XML element in the schema element declaration, whereas previously it was **Any** type.

  In the latter case (element with simple type content), the corresponding CDuce types is an element type. Its tag must correspond to the name of the XML element in the schema element declaration; its content type its the CDuce translation of the simple type provided in the element declaration.

  For example, the following XML Schema element (corresponding to the homonymous element defined in mails.xsd):

```
<xsd:element name="header">
 <xsd:complexType>
  <xsd:simpleContent>
   <xsd:extension base="xsd:string">
    <xsd:attribute ref="name" use="required" />
   </xsd:extension>
  </xsd:simpleContent>
 </xsd:complexType>
</xsd:element>
```

  will be translated to the following CDuce type:

```
# #print_type Mails.header;;
<header name = String>String
```

  Note that the type of the element content *is not a sequence* unless the translation of the XML Schema types is a sequence itself (as you can notice in the example above). Compare it with the following where the element content is not a sequente, but a single record:

```
# #print_type Mails.Date;;
<Date>{
  positive = Bool;
  year = Int; month = Int; day = Int; hour = Int;
  minute = Int; second = Int;
  timezone =? { positive = Bool; hour = Int; minute = Int }
}
```

  XML Schema wildcards (`xsd:any`) and nullable elements (`xsi:nil`) are supported.

- XML Schema **attribute group definitions** are mapped to record types containing one field for each attribute declarations contained in the group. `use` constraints are respected: optional attributes are mapped to optional fields, required

attributes to required fields. XML Schema attribute wildcards are partly supported; they simply produce open record types instead of closed one, but the actual constraints of the wildcards are discarded.

The following XML Schema attribute group declaration:

```
<xsd:attributeGroup name="mimeTypeAttributes">
 <xsd:attribute name="type" type="mimeTopLevelType" use="required" />
 <xsd:attribute name="subtype" type="xsd:string" use="required" />
</xsd:attributeGroup>
```

will thus be mapped to the following CDuce type:

```
# #print_type Mails.mimeTypeAttributes;;
{  type = [
      'image' | 'text' | 'application' | 'audio' | 'message' |
'multipart' | 'video'
    ];
   subtype = String }
```

- XML Schema **model group definitions** are mapped to CDuce sequence types. `minOccurs` and `maxOccurs` constraints are respected, using CDuce recursive types to represent `unbounded` repetition (i.e. Kleene star).

  `all` constraints, also known as *interleaving constraints*, can't be expressed in the CDuce type system avoiding type sizes explosion. Thus, this kind of content models are normalized and considered, in the type system, as sequence types (the validator will reorder the actual XML documents).

  Mixed content models are supported.

  As an example, the following XML Schema model group definition:

```
<xsd:group name="attachmentContent">
 <xsd:sequence>
  <xsd:element name="mimetype">
   <xsd:complexType>
    <xsd:attributeGroup ref="mimeTypeAttributes" />
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="content" type="xsd:string" minOccurs="0" />
 </xsd:sequence>
</xsd:group>
```

  will be mapped to the following CDuce type:

```
# #print_type Mails.attachmentContent;;
[ X1 <content}>String | X1 ] where
X1 = <mimetype S.mimeTypeAttributes>[  ]
```

5.6 XML Schema validation

The processes of XML Schema validation and assessment check that an XML Schema instance document is valid with respect to an XML Schema document and add missing information such as default values. The CDuce's notion of Schema validation is a bit different.

CDuce permits to have XML values made of arbitrary types, for example you can have XML elements which have integer attributes. Still, this feature is rarely used because the function used to load XML documents (**`load_xml`**) returns XML values which have as leaves values of type PCDATA.

Once you have imported an XML Schema in CDuce, you can use it to validate an XML value returned by **`load_xml`** against an XML Schema component defined in it. The process of validation will basically build a CDuce value which has the type corresponding to the conversion of the XML Schema type of the component used in validation to a CDuce type. The conversion is the same described in the previous secion. Note that is not strictly necessary that the input XML value comes from **`load_xml`** it's enough that it has PCDATA values as leaves.

During validation PCDATA strings are parsed to build CDuce values corresponding to XML Schema simple types and whitespace are handled as specified by XML Schema **`whiteSpace`** facet. For example, validating the **`1234567890`** *PCDATA string* against an **`xsd:integer`** simple type will return the CDuce value **`1234567890`** typed with type **`Int`**. Default values for missing attributes or elements are also added where specified.

You can use the **`validate`** keyword to perform validation in CDuce program. The syntax is as follows:**`validate <expr> with <schema_ref>`** where schema_ref is defined as described in XML Schema components import. Same ambiguity rules will apply here.

More in detail, validation can be applied to different kind of CDuce values depending on the type of Schema component used for validation.

- The typical use of validation is to validate against **element declaration**. In such a case validate should be invoked on an XML CDuce value as in the following example.

```
# let xml = <Date>"2003-10-15T15:44:01Z" in
  validate xml with Mails.Date;;
  - : S.Date =
  <Date> {
    time_kind=`dateTime;
    positive=`true;
    year=2003; month=10; day=15;
```

```
    hour=15; minute=44; second=1;
    timezone={ positive=`true; hour=0; minute=0 }
  }
```

The tag of the given element is checked for consistency with the element declaration; attributes and content are checked against the Schema type declared for the element.

- Sometimes you may want to validate an element against an XML Schema **complex type** without having to use element declarations. This case is really similar to the previous one with the difference that the Schema component you should use is a complex type declaration, you can apply such a validation to any XML value. The other important difference is that the tag name of the given value is completely ignored.

  As an example:

```
# let xml = load_xml "envelope.xml" ;;
val xml : Any = <ignored_tag From="fake@microsoft.com">[
                  <From>[ 'user@unknown.domain.org' ]
                  <To>[ 'user@cduce.org' ]
                  <Date>[ '2003-10-15T15:44:01Z' ]
                  <Subject>[ 'I desperately need XML Schema support in
CDuce' ]
                  <header name="Reply-To">[ 'bill@microsoft.com' ]
                  ]
# validate xml with Mails.envelopeType;;
- : S.envelopeType =
    <ignored_tag From="fake@microsoft.com">[
      <From>[ 'user@unknown.domain.org' ]
      <To>[ 'user@cduce.org' ]
      <Date> {
        time_kind=`dateTime;
        positive=`true;
        year=2003; month=10; day=15;
        hour=15; minute=44; second=1;
        timezone={ positive=`true; hour=0; minute=0 }
      }
      <Subject>[ 'I desperately need XML Schema support in CDuce' ]
      <header name="Reply-To">[ 'bill@microsoft.com' ]
    ]
```

- Similarly you may want to validate against a **model group**. In this case you can validate CDuce's sequences against model groups. Given sequences will be considered as content of XML elements.

  As an example:

```
# let xml = load_xml "attachment.xml";;
  val xml : Any =
    <ignored_tag ignored_attribute="foo">[
      <mimetype type="application"; subtype="msword">[ ]
      <content>[ '\n    ### removed by spamoracle ###\n  ' ]
    ]
# let content = match xml with <_>cont -> cont | _ -> raise "failure";;
```

```
  val content : Any = [
    <mimetype type="application"; subtype="msword">[ ]
    <content>[ '\n    ### removed by spamoracle ###\n  ' ]
  ]
# validate content with Mails.attachmentContent;;
- : Mails.attachmentContent =
    [ <mimetype type="application"; subtype="msword">[ ]
      <content>[ '\n    ### removed by spamoracle ###\n  ' ]
    ]
```

- Finally is possible to validate records against **attribute groups**. All required attributes declared in the attribute group should have corresponding fields in the given record. The content of each of them is validate against the simple type defined for the corresponding attribute in the attribute group. Non required fields are added if missing using the corresponding default value (if any).

  As an example:

```
# let record = { type = "image"; subtype = "png" };;
  val record :
    { type = [ 'image' ] subtype = [ 'png' ] } =
      { type="image" subtype="png" }
# validate record with Mails.mimeTypeAttributes ;;
- : { type = [ 'image' | 'text' | ... ] subtype = String } =
      { type="image" subtype="png" }
```

5.7 XML Schema instances output

It is possible to use the normal `print_xml` and `print_xml_utf8` built-in functions to print values resulting from XML Schema validation.

5.8 Unsupported XML Schema features

The support for XML Schema embedded in CDuce does not attempt to cover the full XML Schema specification. In particular, imported schemas are not checked to be valid. You can use for instance this *on-line validator* to check validity of a schema.

Also, some features from the XML Schema specification are not or only partially supported. Here is a non-exhaustive list of limitations:

- Substitution groups.
- Some facets (pattern, totalDigits, fractionDigits).
- `<redefine>` (inclusion of an XML Schema with modifications).
- `xsi:type`.

6 XML Schema sample documents

6.1 Sample XML documents

All the examples you will see in the manual section regarding CDuce's XML Schema support are related to the XML Schema Document **`mails.xsd`** and to the XML Schema Instance **`mails.xml`** reported below.

6.2 mails.xsd

```
<!-- mails.xsd -->

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:element name="mails" type="mailsType" />

 <xsd:complexType name="mailsType">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
   <xsd:element name="mail" type="mailType" />
  </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="mailType">
  <xsd:sequence>
   <xsd:element name="envelope" type="envelopeType" />
   <xsd:element name="body" type="bodyType" />
   <xsd:element name="attachment" type="attachmentType"
                minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute use="required" name="id" type="xsd:integer" />
 </xsd:complexType>

 <xsd:element name="header">
  <xsd:complexType>
   <xsd:simpleContent>
    <xsd:extension base="xsd:string">
     <xsd:attribute ref="name" use="required" />
    </xsd:extension>
   </xsd:simpleContent>
  </xsd:complexType>
```

```
 </xsd:element>

 <xsd:element name="Date" type="xsd:dateTime" />

 <xsd:complexType name="envelopeType">
  <xsd:sequence>
   <xsd:element name="From" type="xsd:string" />
   <xsd:element name="To" type="xsd:string" />
   <xsd:element ref="Date" />
   <xsd:element name="Subject" type="xsd:string" />
   <xsd:element ref="header" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="From" type="xsd:string" use="required" />
 </xsd:complexType>

 <xsd:simpleType name="bodyType">
  <xsd:restriction base="xsd:string" />
 </xsd:simpleType>

 <xsd:complexType name="attachmentType">
  <xsd:group ref="attachmentContent" />
  <xsd:attribute ref="name" use="required" />
 </xsd:complexType>

 <xsd:group name="attachmentContent">
  <xsd:sequence>
   <xsd:element name="mimetype">
    <xsd:complexType>
     <xsd:attributeGroup ref="mimeTypeAttributes" />
    </xsd:complexType>
   </xsd:element>
   <xsd:element name="content" type="xsd:string" minOccurs="0" />
  </xsd:sequence>
 </xsd:group>

 <xsd:attribute name="name" type="xsd:string" />

 <xsd:attributeGroup name="mimeTypeAttributes">
  <xsd:attribute name="type" type="mimeTopLevelType" use="required" />
  <xsd:attribute name="subtype" type="xsd:string" use="required" />
 </xsd:attributeGroup>

 <xsd:simpleType name="mimeTopLevelType">
  <xsd:restriction base="xsd:string">
   <xsd:enumeration value="text" />
   <xsd:enumeration value="multipart" />
   <xsd:enumeration value="application" />
   <xsd:enumeration value="message" />
   <xsd:enumeration value="image" />
   <xsd:enumeration value="audio" />
   <xsd:enumeration value="video" />
  </xsd:restriction>
 </xsd:simpleType>

</xsd:schema>
```

6.3 mails.xml

```
<!-- mails.xml -->

<mails>
  <mail id="0">
```

```
    <envelope From="bill@microsoft.com">
      <From>user@unknown.domain.org</From>
      <To>user@cduce.org</To>
      <Date>2003-10-15T15:44:01Z</Date>
      <Subject>I desperately need XML Schema support in CDuce</Subject>
      <header name="Reply-To">bill@microsoft.com</header>
    </envelope>
    <body>
      As subject says, is it possible to implement it?
    </body>
    <attachment name="signature.doc">
      <mimetype type="application" subtype="msword"/>
      <content>
        ### removed by spamoracle ###
      </content>
    </attachment>
  </mail>
  <mail id="1">
    <envelope From="zack@cs.unibo.it">
      <From>zack@di.ens.fr</From>
      <To>bill@microsoft.com</To>
      <Date>2003-10-15T16:17:39Z</Date>
      <Subject>Re: I desperately need XML Schema support in CDuce</Subject>
    </envelope>
    <body>
      user@unknown.domain.org wrote:
      > As subject says, is possible to implement it?

      Sure, I'm working on it, in a few years^Wdays it will be finished
    </body>
  </mail>
</mails>
```

7 Interfacing CDuce with OCaml

7.1 Introduction

This page describes the CDuce/OCaml interface. This interface allows the programmer to:

- call OCaml functions from a CDuce module;
- export a CDuce model as an OCaml module, by giving it an explicit OCaml signature.

The intended usages for the interface are:

- Piggyback existing OCaml libraries, such as database, network, GUI, data structures;
- Use CDuce as an XML layer (input/output/transformation) for OCaml projects;
- Develop fully mixed OCaml/CDuce projects.

To see how to build CDuce with support for the OCaml interface, see the *INSTALL* file from the CDuce distribution.

7.2 Translating types

The heart of the interface is a mapping from OCaml types to CDuce types. An OCaml type ***t*** is translated to a CDuce type **T(*t*)**, which is meant to be isomorphic to ***t***: there is a canonical function ***t*** # **T(*t*)** from OCaml values of type ***t*** to CDuce values of type **T(*t*)**, and another canonical function **T(*t*)** # ***t***.

- Basic OCaml types **char**, **int**, **string**, **unit** are translated respectively to **Byte = '\0;'--'\255;'**, **-1073741824 -- 1073741823**, **Latin1 = [Byte*]**, **[] = `nil**.
- Tuple types ***t*1 * ... * *t*n** are translated to nested CDuce product types **(T(*t*1),(...,T(*t*n))...)**. A function type ***t* -> *s*** is translated to **T(*t*) -> T(*s*)**. Labels on the argument of the arrow are discarded.

- A list type **`t list`** is translated to an homogeneous sequence type **`[ T(t)* ]`**. An array type **`t array`** has the same translation.
- A option type **`t option`** is translated to the type **`[ T(t)? ]`**.
- A variant type with a declaration **`A1 of t1 | ... | An of tn`** is translated to a type **`(`A1,T(t1)) | ... | (`An,T(tn))`**. If a constructor **`Ai`** has no argument, the resulting term is **`` `Ai ``**, not **`(`Ai,[])`**. Polymorphic variant types are treated similarly.
- A record type with a declaration **`{ l1 : t1; ...; ln : tn }`** is translated to a closed record type **`{ l1 = T(t1); ... ; ln = T(tn) }`**. Mutable fields are just copied.
- Private variant and record types are treated correctly: the interface never tries to generate OCaml values of these types, but it will happily translate them to CDuce values.
- A reference type **`t ref`** is translated to the CDuce reference type **`ref T(t)`**. When converting a Caml reference to CDuce, the operation (set,get) on the resulting reference refers to the original reference. However, when converting a CDuce reference to OCaml, the content of the reference is fetched (set), and a fresh OCaml reference is created (copy semantics).
- The type **`Cduce_lib.Value.t`** is translated to the CDuce type **`Any`**. The corresponding translation functions are the identity. This can be used to avoid multiple copies when translating a complex value back and forth between CDuce and OCaml. The type **`Cduce_lib.Encodings.Utf8.t`** is translated to the CDuce type **`String`**. The type **`Big_int.big_int`** is translated to the CDuce type **`Int`**.
- A *monomorphic* abstract type **`t`** is translated to the CDuce type **`!t`**. This type just acts as a container for values of the abstract type. CDuce never produces a value of this type, and it cannot inspect the content of such a value (apart from checking its type).

The canonical translation is summarized in the following box:

OCaml type *t*	CDuce type T(*t*)
`char`	`Byte = '\0;'--'\255;'`
`int`	`-1073741824 -- 1073741823`
`string`	`Latin1 = [ Byte* ]`
`unit`	`[] = `nil`
`bool`	`Bool = `true | `false`
t1 * ... * *tn*	(T(*t1*),(...,T(*tn*))...)
t -> *s*	T(*t*) -> T(*s*)
t list	[T(*t*)*]
t array	[T(*t*)*]
t option	[T(*t*)?]
A of *t* | B of *s* | C	(`A, T(*t*)) | (`B, T(*s*)) | `C

`[ `A of t | `B of s | `C]`	`(`A, T(t)) | (`B, T(s)) | `C`
`{ x : t; y : s }`	`{ x = T(t); y = T(s) }`
`t ref`	`ref T(t)`
`Cduce_lib.Value.t`	`Any`
`Cduce_lib.Encodings.Utf8.t`	`String`
`Big_int.big_int`	`Int`

Only monomorphic types are handled by the interface. It is allowed to use polymorphic constructors as an intermediate, as long as the final type to be translated is monomorphic. Recursive types, including unguarded ones (option **`-rectypes`** of the OCaml compiler) are accepted. In the following example:

```
type 'a t = A of int | B of 'a t
type s = int t

type 'a u = A of ('a * 'a) u | B
type v = int u
```

the type **`s`** can be translated, but the type **`v`** can't, because its infinite unfolding is not a regular type.

OCaml object types are not supported.

Note that values are copied in depth (until reaching an abstract type, a function types, etc...). In particular, translating an OCaml cyclic values to CDuce will not terminate (well, with a stack overflow!).

7.3 Calling OCaml from CDuce

If an OCaml value has a type that can be translated, it is possible to use it from CDuce (see the How to compile and link section for more details).

In a CDuce module, you can write **`M.f`** to denote the result of translating the OCaml value **`M.f`** to CDuce.

If the value you want to use has a polymorphic type, you can make the translation work by explicitly instantiating its type variables with CDuce types. The syntax is **`M.f with { t1 ... tn }`** where the **`ti`** are CDuce types. The type variables are listed in the order they appear in a left-to-right reading of the OCaml type. Example:

```
let listmap = List.map with { Int String }
```

will return a function of type **`(Int -> String) -> ([Int*] -> [String*])`**

7.4 Calling CDuce from OCaml

We have seen in the section above how OCaml values can be used from a CDuce module. It is also possible to use CDuce values from OCaml. To do so, you must give an OCaml interface (.mli) for the CDuce module (.cdo). The interface can define arbitrary types, and declare monomorphic values. These values must be defined in the CDuce module with a compatible type (subtype of the translation).

As an example, suppose you have this CDuce module (foo.cd):

```
type s = (`A,int) | `B
let double (x : Latin1) : Latin1 = x @ x
let dump (x : s) : Latin1 = string_of x
```

You can define an OCaml interface for it (foo.mli):

```
type t = A of int | B
val double: string -> string
val dump: t -> string
```

When the foo.cdo module is compiled, CDuce will look for the foo.cmi compiled interface (hence, you must first compile it yourself with OCaml), and generate stub code, so as to define an OCaml module **`Foo`** with the given interface. This module can then be linked together with other "regular" OCaml modules, and used from them.

Notes:

- It is not mandatory to export all the values of the CDuce module in the OCaml interface.
- The types defined in the interface cannot (currently) be used within the CDuce module.

7.5 How to compile and link

Here is the protocol to compile a single CDuce module:

- Create a **`.cmi`** from your OCaml file with **`ocamlc -c foo.mli`**.
- Compile your CDuce file **`cduce --compile foo.cd`**. This command will create a CDuce bytecode file **`foo.cdo`**, which also contains the OCaml glue code to export CDuce values as OCaml ones, and to bind OCaml values used within the CDuce module.
- Compile the OCaml glue code **`ocamlfind ocamlc -c -package cduce -pp cdo2ml -impl foo.cdo`**. The**`cdo2ml`** tool extracts the OCaml glue code from the CDuce bytecode file.

You can then link the resulting OCaml module, maybe with other modules (either regular ones, or wrapping a CDuce module): **`ocamlfind ocamlc -o ... -package cduce -linkpkg foo.cmo ...`**. When the program is run, the CDuce bytecode file **`foo.cdo`** is looked in the *current directory* only, and loaded dynamically (with a checksum test).

It might be preferable to include the CDuce bytecode directly into the OCaml glue code. You can do this by giving **`cdo2ml`** the **`-static`** option: **`ocamlfind ocamlc -c -package cduce -pp "cdo2ml -static" -impl foo.cdo`**. Modules which have been compiled this way don't need the corresponding **`.cdo`** at runtime.

If you choose static linking, you have to use a correct ordering when linking with OCaml. Note that it is possible to mix static and dynamic linking for various CDuce modules in a same program.

Everything works *mutatis mutandis* with the native OCaml compiler ocamlopt.

You might need to pass extra **`-I`** flags to CDuce so that it could find the referenced **`.cmi`** files.

It is possible to run a CDuce module with **`cduce --run foo.cdo`**, but only if it doesn't use OCaml values.

Interested users can look at the output of **`cdo2ml`** to better understand how the interface works.

7.6 Calling OCaml from the toplevel

The tool **`cduce_mktop`** creates custom versions of the CDuce toplevel with built-in support for some OCaml modules / functions.

```
cduce_mktop [-I path | -p package | -l unit ... | -byte ] [target]
[primitive file]
```

The **`target`** argument is the file name of the resulting toplevel. The **`primitive file`** argument points to a file whose contents specify a set of built-in OCaml values to be embedded in the toplevel. Each line must either be a qualified value (like **`List.map`**) or the name of an OCaml unit (like **`List`**). Empty lines and lines starting with a sharp character are ignored.

The **`-byte`** flag forces the creation of the bytecode version of the toplevel (by defaut, the toplevel is produced with ocamlopt).

The leading **`-I`** options enrich the search path for OCaml units. The **`-p`** options serves a similar purpose; their arguments are findlib package names. All these paths

are included in the produced toplevel. The **`-l`** options give the OCaml units to be linked in (e.g. **`x.cmx`** or **`x.cmxa`**)(the **`-p`** option automatically include the units).

In a custom toplevel, the directive **`#builtins`** prints the name of embedded OCaml values.

7.7 Examples

Getting the value of an environment variable

```
let home = Sys.getenv "home";;
```

Ejecting your CD with CDuce

This example demonstrates how to use OCamlSDL library.

```
Sdl.init `None [ `EVERYTHING ];;
let cd = Sdlcdrom.cd_open 0;;
Sdlcdrom.cd_eject cd;;
```

If you put these lines in a file **`cdsdl.cd`**, you can compile and link it with:

```
cduce --compile cdsdl.cd -I `ocamlfind query ocamlsdl`
ocamlfind ocamlc -o cdsdl -pp "cdo2ml -static" -impl cdsdl.cdo   \
  -package cduce,ocamlsdl -linkpkg
```

Accessing MySQL

This example demonstrates how to use ocaml-mysql library.

```
let db = Mysql.connect Mysql.defaults;;

match Mysql.list_dbs db `None [] with
 | (`Some,l) -> print [ 'Databases: ' !(string_of l) '\n' ]
 | `None -> [];;

print [
  'Client info: ' !(Mysql.client_info []) '\n'
  'Host info: ' !(Mysql.host_info db) '\n'
  'Server info: ' !(Mysql.server_info db) '\n'
  'Proto info: ' !(string_of (Mysql.proto_info db)) '\n'
];;
```

If you put these lines in a file **`cdmysql.cd`**, you can compile and link it with:

```
cduce --compile cdmysql.cd -I `ocamlfind query mysql`
ocamlfind ocamlc -o cdmysql -pp "cdo2ml -static" -impl cdmysql.cdo   \
  -package cduce,mysql -linkpkg
```

Evaluating CDuce expressions

This example demonstrates how to dynamically compile and evaluate CDuce programs contained in a string.

```
let pr = Cduce_lib.Value.print_utf8

try
 let l = Cduce_lib.Cduce.eval
  "let fun f (x : Int) : Int = x + 1;;
   let fun g (x : Int) : Int = 2 * x;;
   f;; g;;
   let a = g (f 10);;
  "
 in
 transform l with
  | ((`Some,id),v) ->
        pr [ !id ' = ' !(string_of v) '\n' ]
  | (`None, f & (Int -> Int)) ->
        pr [ !(string_of (f 100)) '\n' ]
  | (`None,v) ->
        pr [ !(string_of v) '\n' ]
with (exn & Latin1) ->
  print [ 'Exception: ' !exn  '\n' ]
```

If you put these lines in a file **`eval.cd`**, you can compile and link it with:

```
cduce --compile eval.cd -I `ocamlfind query cduce`
ocamlfind ocamlc -o eval -pp "cdo2ml -static" -impl eval.cdo   \
  -package cduce -linkpkg
```

Use CDuce to compute the factorial on big integers

```
(* File cdnum.mli: *)

val fact: Big_int.big_int -> Big_int.big_int


(* File cdnum.cd: *)

let aux ((Int,Int) -> Int)
 | (x, 0 | 1) -> x
 | (x, n) -> aux (x * n, n - 1)

let fact (x : Int) : Int = aux (Big_int.unit_big_int, x)
  (* Could write 1 instead of Big_int.unit_big_int. Just for fun. *)
```

8 Table of Contents

This manual is under construction !

Sections:

TODO PAGES TABLE OF CONTENTS