CDuce Programming Language Tutorial

Language Version 0.3.2+3

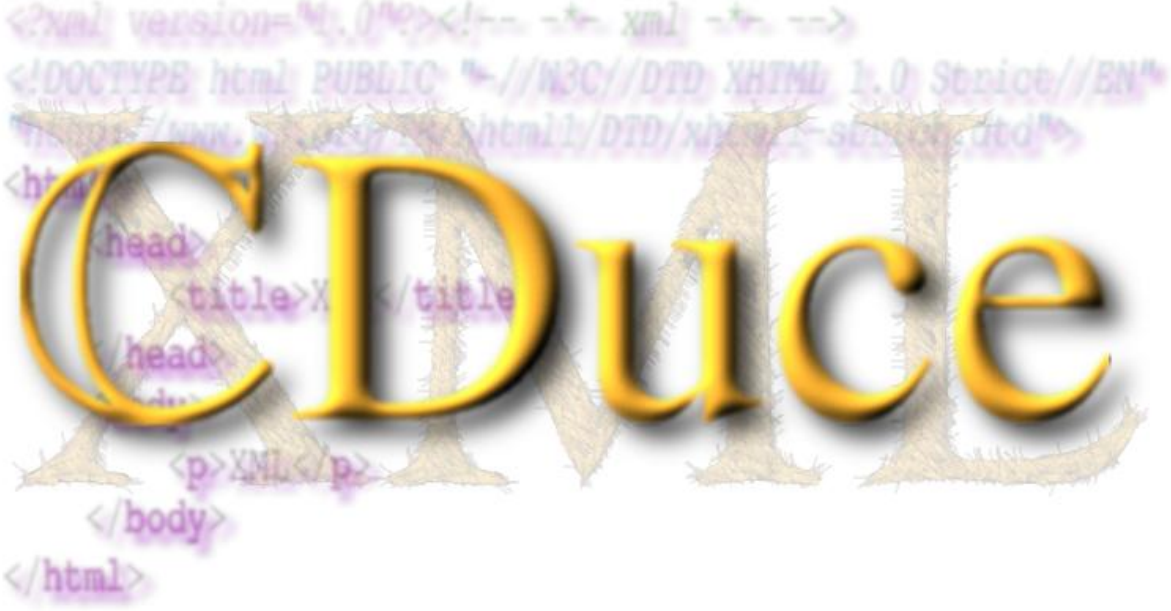

Table of Contents :

1 Getting started

1.1 Key concepts

CDuce is a strongly-typed functional programming language adapted to the manipulation of XML documents. Its syntax is reminiscent of the ML family, but CDuce has a completely different type system.

Let us introduce directly some key concepts:

- **Values** are the objects manipulated by CDuce programs; we can distinguish several kind of values:
  - Basic values: integers, characters.
  - XML documents and fragments: elements, tag names, strings.
  - Constructed values: pairs, records, sequences.
  - Functional values.
- **Types** denote sets of values that share common structural and/or behavioral properties. For instance, `Int` denotes the sets of all integers, and `<a href=String>[]` denotes XML elements with tag `a` that have an attribute `href` (whose content is a string), and with no sub-element.
- **Expressions** are fragments of CDuce programs that *produce* values. For instance, the expression `1 + 3` evaluates to the value `4`. Note that values can be seen either as special cases of expressions, or as the result of evaluating expressions.
- **Patterns** are ``types + capture variables''. They allow to extract from an input value some sub-values, which can then be used in the rest of the program. For instance, the pattern `<a href=x>[]` extracts the value of the `href` attribute and binds it to the *value identifier* `x`.

A first example

```
let x = "Hello, " in
let y = "world!" in
x @ y
```

The expression binds two strings to value identifiers `x` and `y`, and then concatenates them. The general form of the local binding is:

```
let p = e in e'
```

where ***p*** is a pattern and ***e***, ***e'*** are expressions.

> **Note:** A small aside about the examples in this tutorial and their usage. The first program that prints "Hello word" can be tried directly on the on-line prototype: just select and copy it, click on the link to the on-line interpreter in the side bar (we suggest you open it in a new window), paste it in the execution window and run it. The second example instead cannot be run. This is visually signaled by the fact that it contains text in italics. We use italics for meta notation, that is ***e*** and ***e'*** stand for generic expressions, therefore it is useless to run this code (you would just obtain an error signaling that **e** is not bound or that the quote in **e'** is not closed). This is true also in general in what follows: code without italicized text can be copied and pasted in the on-line prototype as they are (of course you must first paste the declarations of the types they use); this is not possible whenever the code contains italicized text.

Patterns are much more than simple variables. They can be used to decompose values. For instance, if the words `Hello` and `world` are in the two elements of a pair, we can capture each of them and concatenate them as follows:

```
let (x,y) = ("Hello, " , "world!") in x @ y
```

Patterns can also check types. So for instance

```
let (x & String, y) = e in x
```

would return a (static) type error if the first projection of ***e*** has not the static type **`String`**.

The form **`let x&t = e in e'`** is used so often that we introduced a special syntax for it:

```
let x : t = e in e'
```

Note the blank spaces around the colons (*). This is because the XML recommendation allows colons to occur in identifiers: see the User's Manual section on *namespaces*. (the same holds true for the functional arrow symbol **`->`** which must

(*) Actually only the first blank is necessary. CDuce accepts **`let x :t = e in e'`**, as well

be surrounded by blanks and by colons in the formal parameters of a function: see *this paragraph* of the User's manual).

1.2 XML documents

CDuce uses its own notation to denote XML documents. In the next table we present an XML document on the left and the same document in CDuce notation on the right (in the rest of this tutorial we we visually distinguish XML code from CDuce one by putting the former in light yellow boxes):

```
<?xml version="1.0"?>
<parentbook>
  <person gender="F">
    <name>Clara</name>
    <children>
      <person gender="M">
        <name>Pål André</name>
        <children/>
      </person>
    </children>
    <email>clara@lri.fr</email>
    <tel>314-1592654</tel>
  </person>
  <person gender="M">
    <name> Bob </name>
    <children>
      <person gender="F">
        <name>Alice</name>
        <children/>
      </person>
      <person gender="M">
        <name>Anne</name>
        <children>
          <person gender="M">
            <name>Charlie</name>
            <children/>
          </person>
        </children>
      </person>
    </children>
    <tel kind="work">271828</tel>
    <tel kind="home">66260</tel>
  </person>
</parentbook>
```

```
let parents : ParentBook =
<parentbook>[
  <person gender="F">[
    <name>"Clara"
    <children>[
      <person gender="M">[
        <name>['Pål ' 'André']
        <children>[]
      ]
    ]
    <email>['clara@lri.fr']
    <tel>"314-1592654"
  ]
  <person gender="M">[
    <name>"Bob"
    <children>[
      <person gender="F">[
        <name>"Alice"
        <children>[]
      ]
      <person gender="M">[
        <name>"Anne"
        <children>[
          <person gender="M">[
            <name>"Charlie"
            <children>[]
          ]
        ]
      ]
    ]
    <tel kind="work">"271828"
    <tel kind="home">"66260"
  ]
]
```

Note the straightforward correspondence between the two notations: instead of using an closing tag, we enclose the content of each element in square brackets. In CDuce square brackets denote sequences, that is, heterogeneous (ordered) lists of blank-separated elements. In CDuce strings are not a primitive data-type but are sequences of characters.

To the purpose of the example we used different notations to denote strings as in CDuce **`"xyz"`**, **`['xyz']`**, **`['x' 'y' 'z']`**, **`[ 'xy' 'z' ]`**, and **`[ 'x' 'yz' ]`** define the same string literal. Note also that the **`"Pål André"`** string is accepted as

CDuce supports Unicode characters.

1.3 Loading XML files

The program on the right hand-side in the previous section starts by binding the variable **parents** to the XML document. It also specifies that parents has the type **ParentBook**: this is optional but it usually allows earlier detection of type errors.

If the file XML on the left hand-side is stored in a file, say, parents.xml then it can be loaded from the file by ***load_xml "parents.xml"*** as the builtin function **load_xml** converts and XML document stored in a file into the CDuce expression representing it. However **load_xml** has type **String->Any**, where **Any** is the type of all values. Therefore if we try to reproduce the same binding as the above by writing the following declaration

```
let parents : ParentBook = load_xml "parents.xml"
```

we would obtain a type error as we were trying to use an expression of type **Any** where an expression of type **ParentBook** is expected. The right way to reproduce the binding above is:

```
let parents : ParentBook =
     match load_xml "parents.xml" with
          x & ParentBook -> x
       |  _ -> raise "parents.xml is not a document of type ParentBook"
```

what this expression does is that before assigning the result of the load_xml expression to the variable **parents** it matches it against the type **ParentBook**. If it succeeds (i.e., if the XML file in the document has type **ParentBook**) then it performs the assignment (the variable **x** is bound to the result of the load_xml expression by the pattern **x&ParentBook**) otherwise it raises an exception.

Of course an exception such as "parents.xml is not a document of type ParentBook" it is not very informative about why the document failed the match an where the error might be. In CDuce it is possible to ask the program to perform this check and raise an informative exception (a string that describes and localize the problem) by using the dynamic type check construction **(*e*:?*t*)** which checks whether the expression ***exp*** has type ***t*** and it either returns the result of ***exp*** or raise an informative exception.

```
let parents  = load_xml "parents.xml" :? ParentBook
```

which perform the same test as the previous program but in case of failure give information to the programmer on the reasons why the type check failed. The dynamic type check can be also used in a let construction as follows

```
let parents :? ParentBook = load_xml "parents.xml"
```

which is completely equivalent to the previous one.

The command **load_xml "parents.xml"** is just an abbreviated form for **load_xml "file://parents.xml"**. If CDuce is compiled with netclient or curl support, then it is also possible to use other URI schemes such as http:// or ftp://. A special scheme string: is always supported: the string following the scheme is parsed as it is. (*) So, for instance, **load_xml "string:*exp*"** parses litteral XML code ***exp*** (it corresponds to XQuery's **{ *exp* }**), while **load_xml ("string:" @ x)** parses the XML code associated to the string variable **x**. Thus the following definition of **x**

```
let x : Any = <person>[ <name>"Alice" <children>[] ]
```

is completely equivalent to this one

```
let x = load_xml "string:<person><name>Alice</name> <children/></person>"
```

1.4 Type declarations

First, we declare some types:

```
type ParentBook = <parentbook>[Person*]
type Person = FPerson | MPerson
type FPerson = <person gender="F">[ Name Children (Tel | Email)*]
type MPerson = <person gender="M">[ Name Children (Tel | Email)*]
type Name = <name>[ PCDATA ]
type Children = <children>[Person*]
type Tel = <tel kind=?"home"|"work">['0'--'9'+ '-'? '0'--'9'+]
type Echar = 'a'--'z' | 'A'--'Z' | '_' | '0'--'9'
type Email= <email>[ Echar+ ('.' Echar+)* '@' Echar+ ('.' Echar+)+ ]
```

The type ParentBook describes XML documents that store information of persons. A tag **<tag attr1=... attr2=... ...>** followed by a sequence type denotes an XML document type. Sequence types classify ordered lists of heterogeneous elements and they are denoted by square brackets that enclose regular expressions over types (note that a regular expression over types *is not* a type, it just describes the content of a sequence type, therefore if it is not enclosed in square brackets it is meaningless). The definitions above state that a ParentBook element is formed by a possibly empty sequence of persons. A person is either of type **FPerson** or **MPerson** according to the value of the gender attribute. An equivalent definition for Person would thus be:

```

```

(*) All these schemes are available for **load_html** and **load_file** as well.

```
<person gender="F"|"M">[ Name Children (Tel | Email)*]
```

A person element is composed by a sequence formed of a name element, a children element, and zero or more telephone and e-mail elements, in this order.

Name elements contain strings. These are encoded as sequences of characters. The **PCDATA** keyword is equivalent to the regexp **Char***, then **String**, **[Char*]**, **[PCDATA]**, **[PCDATA* PCDATA]**, ..., are all equivalent notations. Children are composed of zero or more Person elements. Telephone elements have an optional (as indicated by **=?**) string attribute whose value is either ``home'' or ``work'' and they are formed by a single string of two non-empty sequences of numeric characters separated by an optional dash character. Had we wanted to state that a phone number is an integer with at least, say, 5 digits (of course this is meaningful only if no phone number starts by 0) we would have used an interval type such as **<tel kind=?"home"|"work">[10000--*]**, where ***** here denotes plus infinity, while on the lefthand side of **--** (as in ***--100**) it denotes minus infinity.

Echar is the type of characters in e-mails addresses. It is used in the regular expression defining Email to precisely constrain the form of the addresses. An XML document satisfying these constraints is shown

2 First functions

2.1 First functions

A first example of transformation is **names**, which extracts the sequences of all names of parents in a **ParentBook** element:

```
let names (ParentBook -> [Name*])
    <parentbook>x -> (map x with <person ..>[ n  _*] -> n)
```

The name of the transformation is followed by an *interface* that states that **names** is a function from **ParentBook** elements to (possibly empty) sequences of **Name** elements. This is obtained by matching the argument of the function against the pattern

```
<parentbook>x
```

which binds **x** to the sequence of person elements forming the parentbook. The operator **map** applies to each element of a sequence (in this case **x**) the transformation defined by the subsequent pattern matching. Here **map** returns the sequence obtained by replacing each person in **x** by its **Name** element. Note that we use the pattern

```
<person ..>[ n _*],
```

to match the person elements: **n** matches (and captures) the **Name** element-that is, the first element of the sequence-, **_*** matches (and discards) the sequence of elements that follow, and **person** matches the tag of the person. Since elements of type **Person** contain attributes (actually, just the attribute gender) then we use **..** to match (and discard) them. This is not necessary for the parentbook elements, but we could have specified it as well as **<parentbook ..>x** since **..** matches any sequence of attibutes, the empty one as well.

The interface and the type definitions ensure that the tags will be the expected ones, so we could optimize the code by defining a body that skips the check of the tags:

```
<_> x -> (map x with <_ ..>[ n _*] -> n)
```

However this optimization would be useless since it is already done by the implementation (for technical details see *this paper*) and, of course, it would make the code less readable. If instead of extracting the list of *all* parents we wanted to extract the sublist containing only parents with exactly two children, then we had to replace **transform** for **map**:

```
let names2 (ParentBook -> [Name*])
   <parentbook> x ->
      transform x with <person ..>[ n <children>[Person Person] _*] -> [n]
```

While **map** must be applicable to all the elements of a sequence, **transform** filters only those that make its pattern succeed. The right-hand sides return sequences which are concatenated in the final result. In this case **transform** returns the names only of those persons that match the pattern **<person ..>[n <children>[Person Person] _*]**. Here again, the implementation compiles this pattern exactly as **<_ ..>[n <_>[_ _] _*]**, and in particular avoids checking that sub-elements of **<children>** are of type **Person** when static-typing enforces this property.

These first examples already show the essence of CDuce's patterns: all a pattern can do is to decompose values into subcomponents that are either captured by a variable or checked against a type.

The previous functions return only the names of the outer persons of a **ParentBook** element. If we want to capture all the **name** elements in it we have to recursively apply **names** to the sequence of children:

```
let names (ParentBook -> [Name*])
   <parentbook> x -> transform x with
         <person ..> [ n  <children>c  _*] -> [n]@(names <parentbook>c)
```

where **@** denotes the concatenation of sequences. Note that in order to recursively call the function on the sequence of children we have to include it in a **ParentBook** element. A more elegant way to obtain the same behavior is to specify that names can be applied both to **ParentBook** elements and to **Children** elements, that is, to the union of the two types denoted by **(ParentBook|Children)**:

```
let names ( ParentBook|Children -> [Name*] )
   <_>x -> transform x with <person ..>[ n  c  _*] -> [n]@(names c)
```

Note here the use of the pattern **<_>** at the beginning of the body which makes it possible for the function to work both on **ParentBook** and on **Children** elements.

2.2 Regular Expressions

In all these functions we have used the pattern `_*` to match, and thus discard, the rest of a sequence. This is nothing but a particular regular expression over types. Type regexps can be used in patterns to match subsequences of a value. For instance the pattern `<person ..>[ _ _ Tel+]` matches all person elements that specify no `Email` element and at least one `Tel` element. It may be useful to bind the sequence captured by a (pattern) regular expression to a variable. But since a regexp is not a type, we cannot write, say, `x&Tel+`. So we introduce a special notation `x::R` to bind `x` to the sequence matched by the type regular expression `R`. For instance:

```
let domain (Email ->String) <_>[ _*?  d::(Echar+ '.' Echar+) ] -> d
```

returns the last two parts of the domain of an e-mail (the `*?` is an ungreedy version of `*`, see *regular expressions patterns*). If these ::-captures are used *inside* the scope of the regular expression operators `*` or `+`, or if the same variable appears several times in a regular expression, then the variable is bound to the concatenation of all the corresponding matches. This is one of the distinctive and powerful characteristics of CDuce, since it allows to define patterns that in a single match capture subsequences of non-consecutive elements. For instance:

```
type PhoneItem = {name = String; phones = [String*] }
let agendaitem (Person -> PhoneItem)
    <person ..>[<name>n  _  (t::Tel | _)*] ->
        { name = n ; phones = map t with <tel ..> s ->s }
```

transforms a `person` element into a record value with two fields containing the element's name and the list of all the phone numbers. This is obtained thanks to the pattern `(t::Tel | _)*` that binds to `t` the sequence of all `Tel` elements appearing in the person. By the same rationale the pattern

```
( w::<tel kind="work">_ | t::<tel kind=?"home">_ | e::<email>_ )*
```

partitions the `(Tel | Email)*` sequence into three subsequences, binding the list of work phone numbers to `w`, the list of other numbers to `t`, and the list of e-mails to `e`. Alternative patterns `|` follow a first match policy (the second pattern is matched only if the first fails). Thus we can write a shorter pattern that (applied to `(Tel|Email)*` sequences) is equivalent:

```
( w::<tel kind="work">_ | t::Tel | e::_ )*
```

Both patterns are compiled into

```
```

```
( w::<tel kind="work">_ | t::<tel ..>_ | e::_)*
```

since checking the tag suffices to determine if the element is of type **`Tel`**.

Storing phone numbers in integers rather than in strings requires minimal modifications. It suffices to use a pattern regular expression to strip off the possible occurrence of a dash:

```
let agendaitem2 (Person -> {name=String; phones=[Int*]})
  <person ..>[ <name>n  _  (t::Tel|_)* ] ->
      { name = n; phones = map t with <tel ..>[(s::'0'--'9'|_)*] -> int_of
s }
```

In this case **`s`** extracts the subsequence formed only by numerical characters, therefore **`int_of s`** cannot fail because **`s`** has type **`[ '0'--'9'+ ]`** (otherwise, the system would have issued a warning) (Actually the type system deduces for **`s`** the following type **`[ '0'--'9'+ '0'--'9'+]`** (subtype of the former) since there always are at least two digits).

First use of overloading

Consider the type declaration

```
type PhoneBook = <phonebook>[PhoneItem*]
```

If we add a new pattern matching branch in the definition of the function **`names`**, we make it work both with **`ParentBook`** and **`PhoneBook`** elements. This yields the following *overloaded* function:

```
let names3 (ParentBook -> [Name*] ; PhoneBook -> [String*])
      | <parentbook> x -> (map x with <person ..>[ n  _* ] -> n)
      | <phonebook> x -> (map x with { name=n } -> n)
```

The overloaded nature of **`names3`** is expressed by its interface, which states that when the function is applied to a **`ParentBook`** element it returns a list of names, while if applied to a **`PhoneBook`** element it returns a list of strings. We can factorize the two branches in a unique alternative pattern:

```
let names4 (ParentBook -> [Name*] ; PhoneBook -> [String*])
     <_> x -> map x with ( <person ..>[ n  _* ] | { name=n } ) -> n
```

The interface ensures that the two representations will never mix.

3 Overloading

3.1 Overloaded functions

The simplest form for a toplevel function declaration is

```
let f (t->s) x -> e
```

in which the body of a function is formed by a single branch ***x -> e*** of pattern matching. As we have seen in the previous sections, the body of a function may be formed by several branches with complex patterns. The interface ***t->s*** specifies a constraint on the behavior of the function to be checked by the type system: when applied to an argument of type ***t***, the function returns a result of type ***s***.

Simple Overloading

In general the interface of a function may specify several such constraints, as the *names3* example The general form of a toplevel function declaration is indeed:

```
let f (t1->s1;...;tn->sn) p1 -> e1 | ... | pm -> em
```

(the first vertical bar and the **fun** keyword are optional). Such a function accepts arguments of type (***t1|...|tn***); it has all the types ***ti->si***, and, thus, it also has their intersection ***t1->s1&...&tn->sn***

The use of several arrow types in an interface serves to give the function a more precise type. We can roughly distinguish two different uses of multiple arrow types in an interface:

- when each arrow type specifies the behavior of a different piece of code forming the body of the function, the compound interface serves to specify the *overloaded* behavior of the function. This is the case for the function below

```
let add ( (Int,Int)->Int ; (String,String)->String )
    | (x & Int, y & Int) -> x+y
    | (x & String, y & String) -> x@y
```

where each arrow type in the interface refers to a different branch of the body.

- when the arrow types specify different behavior for the same code, then the compound interface serves to give a more precise description of the behavior of the function. An example is the function *names4* from Section "".

There is no clear separation between these two situations since, in general, an overloaded function has body branches that specify behaviors of different arrow types of the interface but share some common portions of the code.

A more complex example

Let us examine a more complex example. Recall the types used to represent persons that we defined in Section "*Getting started*" that for the purpose of the example we can simplify as follows:

```
type Person   = FPerson | MPerson
type FPerson  = <person gender = "F">[ Name Children ]
type MPerson  = <person gender = "M">[ Name Children ]
type Children = <children>[ Person* ]
type Name     = <name>[ PCDATA ]
```

We want to transform this representation of persons into a different representation that uses different tags **<man>** and **<woman>** instead of the gender attribute and, conversely, that uses an attribute instead of an element for the name. We also want to distinguish the children of a person into two different sequences, one of sons, composed of men (i.e. elements tagged by **<man>**), and the other of daughters, composed of women. Of course we also want to apply this transformation recursively to the children of a person. In practice, we want to define a function **<split>** of type **Person ->(Man | Woman)** where **Man** and **Woman** are the types:

```
type Man = <man name=String>[ Sons Daughters ]
type Woman = <woman name=String>[ Sons Daughters ]
type Sons = <sons>[ Man* ]
type Daughters = <daughters>[ Woman* ]
```

Here is a possible way to implement such a transformation:

```
let fun split (MPerson -> Man ; FPerson -> Woman)
  <person gender=g>[ <name>n <children>[(mc::MPerson | fc::FPerson)*] ] ->
  (* the above pattern collects all the MPerson in mc, and all the FPerson
in fc *)
     let tag = match g with "F" -> `woman | "M" -> `man in
     let s = map mc with x -> split x in
     let d = map fc with x -> split x in
     <(tag) name=n>[ <sons>s  <daughters>d ] ;;
```

The function **split** is declared to be an overloaded function that, when applied to a **MPerson**, returns an element of type **Man** and that, when applied to a **FPerson**,

returns an element of type **Woman**. The body is composed of a single pattern matching

```
  <person gender=g>[ <name>n <children>[(mc::MPerson | fc::FPerson)*] ] ->
```

whose pattern binds four variables: **g** is bound to the gender of the argument of the function, **n** is bound to its name, **mc** is bound to the sequence of all children that are of type **MPerson**, and **fc** is bound to the sequence of all children that are of type **FPerson**.

On the next line we define **tag** to be **`man** or **`woman** according to the value of **g**.

```
    let tag = match g with "F" -> `woman | "M" -> `man
```

Then we apply **split** recursively to the elements of **mc** and **fc**.

```
    let s = map mc with x -> split x in
    let d = map fc with x -> split x in
    <(tag) name=n>[ <sons>s  <daughters>d ] ;;
```

Here is the use of overloading: since **mc** is of type **[MPerson*]**, then by the overloaded type of **split** we can deduce that **s** is of type **[Man*]**; similarly we deduce for **d** the type **[Woman*]**. From this the type checker deduces that the expressions **<sons>s** and **<daughters>** are of type **Sons** and **Daughters**, and therefore it returns for the **split** function the type **(MPerson -> Man) & (FPerson -> Woman)**. Note that the use of overloading here is critical: although **split** has *also* type **Person ->(Man | Woman)** (since **split** is of type **MPerson->Man & FPerson->Woman**, which is a subtype), had we declared **split** of that type, the function would not have type-checked: in the recursive calls we would have been able to deduce for **s** and for **d** the type **[(Man | Woman)*]**, which is not enough to type-check the result. If, for example, we wanted to define the same transformation in XDuce we would need first to apply a filter (that is our transform) to the children so as to separate male from females (while in CDuce we obtain it simply by a pattern) and then resort to two auxiliary functions that have nearly the same definition and differ only on their type, one being of type **MPerson -> Man**, the other of type **FPerson -> Woman**. The same transformation can be elegantly defined in XSLT with a moderate nloc increase, but only at the expense of loosing static type safety and type-based optimizations.

4 Patterns

4.1 Key concepts

4.2 Pair and Record Patterns

4.3 Sequence patterns

4.4 XML elements and attributes

Up to now we used for XML elements (and their types) an abbreviated notation as for **<table align="center" valign="top">[***some_content***]**. Actually, the precise syntax of XML elements is

```
<(expr1) (expr2)>expr3
```

where ***expr1***, ***expr2***, and***expr3*** are generic expressions. The same holds true for record patterns, but where the generic expressions are replaced by generic patterns (that is, **<(*p1*) (*p2*)>*p3***). It is important to notice that the parentheses (in red) **<(*expr1*) (*expr2*)>*expr3*** are part of the syntax. Even if ***expr1***, ***expr2***, and ***expr3*** may be any expression, in practice they mostly occur in a very precise form. In particular, ***expr1*** is an *atom*, ***expr2*** is a *record value*, while ***expr3*** is a sequence. Since this corresponds, by far, to the most common use of XML elements we have introduced some handy abbreviations: in particular we allow the programmer to omit the surrounding **(` )** when ***expr1*** is an atom, and to omit the surrounding **{ }** and the infix semicolons **;** when ***expr2*** is a record value. This is why we can write **<table align="center" valign="top">[** ... **]**, rather than **<(`table) ({align="center"; valign="top"}) >[** ... **]**

While these abbreviations are quite handy, they demand some care when used in record patterns. As we said, the general form of a record pattern is:

```
<(p1) (p2)>p3
```

and the same abbreviations as for expressions apply. In particular, this means that, say, the pattern **<t (a)>_** stands for **<(`t) (a)>_**. Therefore while **<t (a)>_** matches all the elements of tag **t** (and captures in the variable **a** the attributes), the pattern **<(t) (a)>_** matches all XML elements (whatever their tag is) and captures their tag in the variable **t** (and their attributes in **a**). Another point to notice is that **<t>_** stands for **<t ({})>_** (more precisely, for **<(`t) ({})>_**). Since **{}** is the *closed* empty record type, then it matches only the empty record. Therefore **<t>_** matches all elements of tag **t** that have no attibute. We have seen at the beginning of this tutorial that in order to match all element of tag **t** independently from whether they have attributes or not, we have to use the pattern **<t ..>_** (which stands for **<(`t) ({..})>_**).

In the following we enumerate some simple examples to show what we just explained. In these examples we use the following definitions for bibliographic data:

```
type Biblio  = [(Paper|Book)*]
type Paper   = <paper isbn=?String year=String>[ Author+ Title Conference
Url? ]
type Book    = <book isbn=String> [ Author+ Title Url? ]
type Author  = <author>[ PCDATA ]
type Title   = <title>[ PCDATA ]
type Conference = <conference>[ PCDATA ]
type Url     = <url>[ PCDATA ]
```

Let **bib** be of type **Biblio** then

```
transform bib with
     <book (a)> [ (x::(Any\Url)|_)* ] -> [ <book (a)> x ]
```

returns the list of all books without their Url element (if any).

```
transform bib with
     <(book) (a)> [ (x::(Any\Url)|_)* ] -> [ <(book) (a)> x ]
```

returns the bibliography in which all entries (either books or papers) no longer have their Url elements (**book** is now a capture variable). Equivalently we could have pushed the difference on tags:

```
transform bib with
     <(book) (a)> [ (x::<(Any\`url)>_|_)* ] -> [ <(book) (a)> x ]
```

We can perform many kinds of manipulations on the attributes by using the *operators for records*, namely ***r\l*** which deletes the field ***l*** in the record ***r*** whenever it is present, and ***r1 + r2*** which merges the records ***r1*** and ***r2*** by giving the priority to the fields in the latter. For instance

```
transform bib with
     <(t) (a)> x  -> [ <(x) (a\isbn)> x ]
```

strips all the ISBN attributes.

```
transform bib with
     <_ (a)> [(x::(Author|Title|Url)|_)*]  -> [ <book
({isbn="fake"}+a\year)> x ]
```

returns the bibliography in which all Paper elements are transformed into books; this is done by forgetting the Conference elements, by removing the year attributes and possibly adding a fake isbn attribute. Note that since record concatenation gives priority to the record on the righ handside, then whenever the record captured by **a** already contains an isbn attribute, this is preserved.

As an example to summarize what we said above, consider the the elements **table**, **td** and **tr** in XHTML. In transitional XHTML these elements can have an attribute **bgcolor** which is deprecated since in strict XHTML the background color must be specified by the **style** attribute. So for instance **<table bgcolor="#ffff00" style="font-family:Arial">**... must be rewritten as **<table style="bgcolor:#ffff00; font-family:Arial">**... to be XHTML strict compliant. Here is a function that does this transformation on a very simplified version of possibly nested tables containing strings.

```
type Table = <table { bgcolor=?String; style=?String }>[ Tr+]
type Tr = <tr { bgcolor=?String; style=?String }>[ Td+]
type Td = <td { bgcolor=?String; style=?String }>[ Table* | PCDATA ]

let strict ([Table*]->[Table*]; [Tr+]->[Tr+]; [Td+]->[Td+];
[PCDATA]->[PCDATA])
  x ->
    map x with
       <(t) (a& { bgcolor=c; style=s })> l
            -> <(t) (a\bgcolor+{style=(s@"; bgcolor:"@c)})>(strict l)
    |  <(t) (a& { bgcolor=c })> l
            -> <(t) (a\bgcolor+{style=("bgcolor:"@c)})>(strict l)
    |  <(t) (a)> l -> <(t) (a)>(strict l)
    |   c -> c
```

As an exercise the reader can try to rewrite the function **strict** so that the first three branches of the map are condensed into a unique branch.

4.5 Handling optional attributes

The blend of type constructors and boolean combinators can be used to reduce verbosity in writing pattern matching. As an example we show how to handle tags with several optional attributes.

Consider the following fragment of code from site.cd from the CDuce distribution that

we have changed a bit so that it stands alone:

```
type Sample = <sample highlight=?"true"|"false">String

let content (Sample -> String)
  | <sample highlight="false">_ -> "non-higlighted code"
  | <sample ..>_ -> "highlighted code"
```

The idea here is to use the highlight attribute to specify that certain pieces of **<sample>** should be emphasized. When the higlight attribute is missing, the default value of "true" is presumed.

But what if we have two optional attributes? The naive solution would be to write the *four* possible cases:

```
type Sample = <sample lineno=?"true"|"false"
highlight=?"true"|"false">String

let content (Sample -> String)
  | <sample highlight="false" lineno="false">_ -> "lineno=false,
highlight=false"
  | <sample lineno="false">_ -> "lineno=false, highlight=true"
  | <sample highlight="false">_ -> "lineno=true, highlight=false"
  | <sample ..>_ -> "lineno=true, highlight=true,"
```

The intended use for the **lineno** attribute is to tell us whether line numbers should be displayed alongside the sample code. While this situation is still bearable it soon become unfeasible with more than two optional attributes. A much better way of handling this situation is to resort to intersection and default patterns as follows:

```
let content (Sample -> String)
  | <sample ( ({ highlight = h ..} | (h := "true"))
            &({ lineno = l ..} | (l := "true")) )>_
         -> ['lineno=' !l ', highlight=' !h]
```

The intersection pattern **&** makes both patterns to be matched against the record of attributes: each pattern checks the presence of a specific attribute (the other is ignored by matching it with **..**), if it is present it captures the attribute value in a given variables while if the attribute is absent the default sub-pattern is used to assign the variable a default value.

The use of patterns of the form **({ label1= x } | (x := *v*)) & { label2 = y }** is so common in handling optional fields (hence, XML attributes) that CDuce has a special syntax for this kind of patterns: **{ label1 = x else (x := *v*) ; label2 = y }**

4.6 Recursive patterns

Recursive patterns use the same syntax as recursive types: ***P* where *P1*=*p1* and ... and *Pn*=*pn*** with *P, P1,..., Pn* being variables ranging over pattern identifiers (i.e., identifiers starting by a capital letter). Recursive patterns allow one to express complex extraction of information from the matched value. For instance, consider the pattern **`P where P = (x & Int, _) | (_, P)`**; it extracts from a sequence the first element of type **`Int`** (recall that sequences are encoded with pairs). The order is important, because the pattern **`P where P = (_, P) | (x & Int, _)`** extracts the *last* element of type **`Int`**.

A pattern may also extract and reconstruct a subsequence, using the convention described before that when a capture variable appears on both sides of a pair pattern, the two values bound to this variable are paired together. For instance, **`P where P = (x & Int, P) | (_, P) | (x := `nil)`** extracts all the elements of type **`Int`** from a sequence (**`x`** is bound to the sequence containing them) and the pattern **`P where P = (x & Int, (x & Int, _)) | (_, P)`** extracts the first pair of consecutive integers.

4.7 Compiling regular expression patterns

CDuce provides syntactic sugar for defining patterns working on sequences with regular expressions built from patterns, usual regular expression operators, and *sequence capture variables* of the form **`x::`*`R`*** (where *R* is a pattern regular expression).

Regular expression operators **`*`**, **`+`**, **`?`** are *greedy* in the sense that they try to match as many times as possible. Ungreedy versions **`*?`**, **`+?`** and **`??`** are also provided; the difference in the compilation scheme is just a matter of order in alternative patterns. For instance, **`[_* (x & Int) _*]`** is compiled to **`P where P = (_,P) | (x & Int, _)`** while **`[_*? (x & Int) _*]`** is compiled to **`P where P = (x & Int, _) | (_,P)`**.

Let us detail the compilation of an example with a sequence capture variable:

```
[ _*?  d::(Echar+ '.' Echar+) ]
```

The first step is to propagate the variable down to simple patterns:

```
[ _*?  (d::Echar)+ (d::'.') (d::Echar)+ ]
```

which is then compiled to the recursive pattern:

```
P where P = (d & Echar, Q) | (_,P)
    and Q = (d & Echar, Q) | (d & '.', (d & Echar, R))
    and R = (d & Echar, R) | (d & `nil)
```

The **(d & `nil)** pattern above has a double purpose: it checks that the end of the matched sequence has been reached, and it binds **d** to **`nil**, to create the end of the new sequence.

Note the difference between **[x&Int]** and **[x::Int]**. Both patterns accept sequences formed of a single integer **i**, but the first one binds **i** to **x**, whereas the second one binds to **x** the sequence **[i]**.

A mix of greedy and ungreedy operators with the first match policy of alternate patterns allows the definition of powerful extractions. For instance, one can define a function that for a given person returns the first work phone number if any, otherwise the last e-mail, if any, otherwise any telephone number, or the string **"no contact"**:

```
let preferred_contact(Person -> String)
   <_>[  _  _  ( _*? <tel kind="work">x)  |  (_* <email>x) |  <tel ..>x ]
-> x
  |  _ -> "no contact"
```

(note that **<tel ..>x** does not need to be preceded by any wildcard pattern as it is the only possible remaining case).

5 Error messages and Warnings

5.1 Key concepts

CDuce, statically detects a large class of error and tries to help their debugging by providing precise error messages and, in case of type errors, by showing a description (we call it a "sample") of specific values that would make the computation fail.

CDuce signals the classic syntax errors as well as those for instance of unbound variables. It also checks that pattern matching is exhaustive (*). For instance if we declare the type **Person** defined in Section "" and try the following definition:

```
fun name (Person -> String)
     | <person gender = "F">[ n ;_] -> n
```

then we obtain the following message error (frames of the same form as the following denote text taken verbatim from the on line demo, no color or formatting added):

```
Error at chars 228-298:
fun name (Person -> String)
     | <person gender = "F">[ n ;_] -> n
This pattern matching is not exhaustive
Residual type:
<person gender = [ 'M' ]>[ Name Children ]
Sample:
<person {| gender = [ 'M' ] |}>[ <name {| |}>[  ] <children
{| |}>[  ] ]
```

This error message tells us three things: (1) that pattern matching is not defined for

(*) It checks it in functions, **match**, and **map** expressions, but not for **transform** and **xtransform** for which a default branch returning the empty sequence is always defined

all the possible input types (as we forgot the case when the attribute is **"M"**); (2) it gives us the exact type of the values of the type we have forgotten in our matching (in this case this is exactly **MPerson**); (3) it shows us a "sample" of the residual type, that is a simplified representation of a value that would make the expression fail (in this case it shows us the value **<person gender="M">[<name>[ ] <children>[ ]]**).

> **Note:** Samples are simplified representations of values in the sense that they show only that part of the value that is relevant for the error and may omit other parts that are needed to obtain an effective value.

Warnings

CDuce use warnings to signal possible subtler errors. So for instance it issues a warning whenever a capture variable of a pattern is not used in the subsequent expression. This is very useful for instance to detect misprinted types in patterns such as in:

```
transform [ 1 "c" 4 "duce" 2 6 ] with
   x & Sting -> [ x ]
```

The intended semantics of this expression was to extract the sequence of all the strings occuring in the matched sequence. But because of the typo in **St(r)ing** the transformation is instead the identity function: **Sting** is considered as a fresh capture variable. CDuce however detects that **Sting** is never used in the subsequent expression and it pinpoints the possible presence of an error by issuing the following warning:

```
Warning at chars 42-60:
   x & Sting -> [ x ]
The capture variable Sting is declared in the pattern but
not used in
the body of this branch. It might be a misspelled or
undeclared type
or name (if it isn't, use _ instead).
transform [ 1 "c" 4 "duce" 2 6 ] with
   x & Sting -> [ x ]
- : [ 1 [ 'c' ] 4 [ 'duce' ] 2 6 ] =
    [ 1 "c" 4 "duce" 2 6 ]

Ok.
```

5.2 Empty types

CDuce's type system can find very nasty errors. For instance look at this DTD declaration

```
<!ELEMENT person (name,children)>
<!ELEMENT children (person+)>
<!ELEMENT name (#PCDATA)>
```

Apparently this declaration does not pose any problem. But if you consider it more carefully you will see that there exists no document that can be valid for such a DTD, as a person contains a sequence of children that contain a non empty sequence of persons, etc generating an infinite tree.

Let us write the same type in CDuce and look at the result returned by the type-checker

```
type Person = <person>[ Name Children ]
type Children = <children>[Person+]
type Name = <name>[PCDATA]

Warning at chars 57-76:
type Children = <children>[Person+]
This definition yields an empty type for Children
Warning at chars 14-39:
type Person = <person>[ Name Children ]
This definition yields an empty type for Person
```

The type checker correctly issues a "Warning" to signal that the first two types are empty. Note that instead the declarations

```
type Person = <person>[ Name Children ]
type Children = <children>[(ref Person)+]
type Name = <name>[PCDATA]
```

correctly do not yield any warning: in this case it is possible to build a value of type person (and thus of type children), for instance by using a recursive definition where a person is a child of itself.

We paid special care in localizing errors and suggesting solutions. You can try it by yourself by picking the examples available on the *on line demo* and putting in them random errors.

5.3 Unused branches

The emptiness test is used also to check for possible errors in the definition of patterns. If the type checker statically determines that a pattern in a match operation can never be matched then it is very likely that even if the match expression is

well-typed, the programmer had made an error. This is determined by checking whether the intersection of set of all values that can be fed to the branch and the set of all values that Consider for example the following code:

```
type Person = <person>[<name>String <tel>String  (<email>String)?]

fun main_contacts(x : [Person*]):[String*] =
  transform x with
     | <_>[_ _ <emal>s] -> [s]
     | <_>[_  <tel>s ] -> [s]
```

This function was supposed extract the list of contacts from a list of persons elements giving priority to email addresses over telephone numbers. Even if there is a typo in the pattern of the first branch, the function is well typed. However because of the typo the first branch will never be selected and emails never printed. The CDuce type-checker however recognizes that this branch has no chance to be selected since **`Person & <_>[_ _ <emal>s]`**=**`Empty`** and it warns the programmer by issuing the following warning message:

```
Warning at chars 144-167:
        | <_>[_ _ <emal>;s] -> [s]
This branch is not used
fun main_contacts(x : [Person*]):[String*] =
    transform x with
       | <_>[_ _ <emal>s] -> [s]
       | <_>[_  <tel>s ] -> [s]
- : [ Person* ] -> [ String* ] = <fun>

Ok.
```

6 References

6.1 Introduction

6.2 Advanced programming

The fact that reference types are encoded, rather than primitive, brings some advantages. Among these it is noteworthy that, thanks to the encoding, the default behavior of the **`get`** and **`set`** functions can be modified. So a programmer can define a reference that whenever is read, records the access in a log file, or it performs some sanity checks before performing a writing.

For instance the following template program, shows a way to define an integer reference **`x`** that whenever it is read executes some extra code, while whenever it is written performs some checks and possibly raises an exception:

```
let x : ref Int =
   let hidden = ref Int 0 in
     { get = (some_extra_code; hidden . get) ;
       set = fun (x :Int):[] =
               if some_condition
                  then hidden . set(x)
                  else raise "Invalid assignment"
     }
```

Another advantage is that it is possible to define the types for read only and write only channels, which can be specialized respectively in a covariant and contravariant way. For instance if the body of a function performs on some integer reference passed as argument only read operations, then it can specify its input type as **`fun ( x :{ get = []->T ; ..} )...`**. In this case the function can accept as argument any reference of type **`ref S`**, with **`S`** subtype of **`T`**.

6.3 'ref Empty' is not Empty?!

However the use of the encoding also causes some weirdness. For instance a

consequence of the encoding is that the type **`ref Empty`** is inhabited (that is there exists some value of this type). We invite the reader to stop reading the rest of this section and try as an exercise to define a value of type **`ref Empty`**.

The key observation to define a value of **`ref Empty`** is that for every type ***`T`*** the type ***`T`*** **`-> Empty`** is inhabited. Of course it is inhabited only by functions that loop forever (since if such a function returned a value this value would be of type **`Empty`**). But, for instance, **`fun f(x :`*`T`*`):Empty = f x`** is a value of type ***`T`*** **`-> Empty`**.

By using the observation above it is then easy to explicitly define a reference **`y`** of type **`ref Empty`**, as follows:

```
let y : ref Empty =
  let fun f (x :[]):Empty = f x in
    { get = f ;
      set = fun (x :Empty):[] = []}
```

Of course such a reference is completely useless, but its existence yields some unexpected behavior when matching reference types. Consider the following function:

```
```

The matching expression is not exhaustive since it does not deal with the case where the argument is of type **`ref (Int & Bool)`** that is **`ref Empty`**

7 Queries

7.1 Select from where

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 b
```

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

The **select_from_where** construction is translated into:

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

7.2 XPath-like expressions

XPath-like expressions are of two kind : ***e/t*** , ***e/@a*** , (and ***e//t***) where ***e*** is an expression, ***t*** a type, and ***a*** an attribute.

They are syntactic sugar for :

```
 flatten(select x from <_ ..>[(x::t | _ )*] in e)
```

and

```
select x from <_ a=x ..>_ in  e
```

7.3 Examples

Types and data for the examples

Let us consider the following DTD and the CDuce types representing a Bibliography

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>

type bib = <bib>[book*]
type book = <book year=String>[title  (author+ | editor+ ) publisher price
]
type author = <author>[last  first ]
type editor = <editor>[last  first  affiliation ]
type title  = <title>[PCDATA ]
type last  = <last>[PCDATA]
type first  = <first>[PCDATA]
type affiliation = <affiliation>[PCDATA]
type publisher  = <publisher>[PCDATA]
type price  = <price>[PCDATA]
```

and some values

```
let biblio : bib =
           <bib>[
              <book year="1994">[
                 <title>['TCP/IP Illustrated']
                 <author>[
                    <last>['Stevens']
                    <first>['W.']]
                 <publisher>['Addison-Wesley']
                 <price>['65.95'] ]
             <book year="1992">[
                <title>['Advanced Programming in the Unix environment']
                <author>[
                   <last>['Stevens']
                   <first>['W.']]
                <publisher>['Addison-Wesley']
                <price>['65.95'] ]
             <book year="2000">[
                <title>['Data on the Web']
                <author>[
                  <last>['Abiteboul']
                  <first>['Serge']]
                <author>[
```

```
                  <last>['Buneman']
                  <first>['Peter']]
               <author>[
                  <last>['Suciu']
                  <first>['Dan']]
               <publisher>['Morgan Kaufmann Publishers']
               <price>['39.95'] ]
            <book year="1999">[
               <title>['The Economics of Technology and Content for
Digital TV']
               <editor>[
                  <last>['Gerbarg']
                  <first>['Darcy']
                  <affiliation>['CITI'] ]
               <publisher>['Kluwer Academic Publishers']
               <price>['129.95']]
            ]
```

Projections

- All titles in the bibliography biblio

```
let titles = [biblio]/book/title
```

Which yields to:

```
val titles : [ title* ] = [ <title>[ 'TCP/IP Illustrated' ]
                            <title>[ 'Advanced Programming in the Unix
environment' ]
                            <title>[ 'Data on the Web' ]
                            <title>[ 'The Economics of Technology and
Content for Digital TV' ]
                          ]
```

- All authors in the bibliography biblio

```
let authors = [biblio]/book/<author>_
```

Yielding the result:

```
val authors : [ <author>[ last first ]* ] =
                            [ <author>[ <last>[ 'Stevens' ] <first>[
'W.' ] ]
                              <author>[ <last>[ 'Stevens' ] <first>[
'W.' ] ]
                              <author>[
                                <last>[ 'Abiteboul' ]
                                <first>[ 'Serge' ] ]
                              <author>[
                                <last>[ 'Buneman' ]
                                <first>[ 'Peter' ] ]
                              <author>[ <last>[ 'Suciu' ] <first>[ 'Dan'
] ]
                            ]
```

```
```

Note the difference between this two projections. In the fist one, we use the preset type title (type title = <title>[PCDATA]). In the second one, the type <author>_ means all the xml fragments beginning by the tag author (_ means Any), and this tag is without attribute. In contrary, we write note <author ..>_.

- All books having an editor in the bibliography biblio

```
let edibooks = [biblio]/<book year=_>[_* editor _*]
```

Yielding:

```
val edibooks : [ <book year=String>[ title editor+ publisher price]* ] =
          [ <book year="1999">[
                <title>[ 'The Economics of Technology and Content for
Digital TV']
                <editor>[
                   <last>[ 'Gerbarg' ]
                   <first>[ 'Darcy' ]
                <affiliation>[ 'CITI' ] ]
                <publisher>[ 'Kluwer Academic Publishers' ]
                <price>[ '129.95' ] ]
          ]
```

- All books without authors.

```
let edibooks2 = [biblio]/<book ..>[(Any\author)*]
```

Yielding:

```
val edibooks2 : [ <book year=String>[ title editor+ publisher price]* ]
=
           [ <book year="1999">[
                <title>[ 'The Economics of Technology and Content for
Digital TV']
                <editor>[
                   <last>[ 'Gerbarg' ]
                   <first>[ 'Darcy' ]
                <affiliation>[ 'CITI' ] ]
                <publisher>[ 'Kluwer Academic Publishers' ]
                <price>[ '129.95' ] ]
          ]
```

- The year of books having a price of 65.95 in the bibliography biblio

```
let books = [biblio]/<book ..>[_* <price>['65.95']]/@year
```

Yielding:

```
val books : [ String* ] = [ "1994" "1992" ]
```

- All the authors and editors in biblio

```
let aebooks = [biblio]/book/(author|editor)
```

Yielding:

```
val aebooks : [ (editor | author)* ] = [ <author>[
                                           <last>[ 'Stevens' ]
                                           <first>[ 'W.' ]
                                           ]
                                         <author>[
                                           <last>[ 'Stevens' ]
                                           <first>[ 'W.' ]
                                           ]
                                         <author>[
                                           <last>[ 'Abiteboul' ]
                                           <first>[ 'Serge' ]
                                           ]
                                         <author>[
                                           <last>[ 'Buneman' ]
                                           <first>[ 'Peter' ]
                                           ]
                                         <author>[
                                           <last>[ 'Suciu' ]
                                           <first>[ 'Dan' ]
                                           ]
                                         <editor>[
                                           <last>[ 'Gerbarg' ]
                                           <first>[ 'Darcy' ]
                                           <affiliation>[ 'CITI' ]
                                           ]
                                         ]
```

An interesting point in Cduce is the static typing of an expression. By example if we consider the third projection, "All books having an editor in the bibliography", CDuce knows the type of the result of the value ***edibooks***:

```
val edibooks : [ <book year=String>[ title editor+ publisher price]* ] =
...
```

This type represents a book without author (see the book type in the type declaration in the top of this section). Now if we want to know all authors of this list of books ***edibooks***:

```
let authorsofedibooks = edibooks/author
```

Yelding:

```
Warning at chars 24-39:
This projection always returns the empty sequence
val authorsofedibooks : [  ] = ""
```

In fact the value ***edibooks*** must be a subtype of [<_ ..>[Any* author Any*] *], and here this is not the case. If you want to be sure, test this:

```
match edibooks with [<_ ..>[_* author _*] *] -> "this is a subtype" | _ ->
"this is not a subtype" ;;
```

An other projection should be convince you, is the query:

```
let freebooks = [biblio]/book/<price>['0']
]
```

Yelding:

```
val freebooks : [ <price>[ '0' ]* ] = ""
```

There is no free books in this bibliography, That is not indicated by the type of biblio. Then, this projection returns the empty sequence (**""**)

Select_from_where

The same queries we wrote above can of course be programmed with the **select_from_where** construction

All the titles

```
let tquery = select y
             from x in [biblio]/book ,
                  y in [x]/title
```

This query is programmed in a XQuery-like style largely relying on the projections. Note that **x** and **y** are CDuce's patterns. The result is:

```
val tquery : [ title* ] = [ <title>[ 'TCP/IP Illustrated' ]
                            <title>[ 'Advanced Programming in the Unix
environment' ]
                            <title>[ 'Data on the Web' ]
                            <title>[ 'The Economics of Technology and
Content for Digital TV' ]
                          ]
```

Now let's program the same query with the translation given previously thus eliminating the **y** variable

```
let withouty = flatten(select [x] from x in [biblio]/book/title)
```

Yielding:

```
val tquery : [ title* ] = [ <title>[ 'TCP/IP Illustrated' ]
                            <title>[ 'Advanced Programming in the Unix
environment' ]
                            <title>[ 'Data on the Web' ]
                            <title>[ 'The Economics of Technology and
Content for Digital TV' ]
                          ]
```

But the **select_from_where** expressions are likely to be used for more complex queries such as the one that selects all titles whose at least one author is "Peter Buneman" or "Dan Suciu"

```
let sel = select y
          from x in [biblio]/book ,
               y in [x]/title,
               z in [x]/author
where ( (z = <author>[<last>['Buneman']<first>['Peter']])
 || (z = <author>[<last>['Suciu'] <first>['Dan']]) )
```

Which yields:

```
val sel : [ title* ] = [ <title>[ 'Data on the Web' ]
                         <title>[ 'Data on the Web' ]
                         ]
```

Note that the corresponding semantics, as in SQL, is a multiset one. Thus duplicates are not eliminated. To discard them, one has to use the **distinct_values** operator.

A pure pattern example

This example computes the same result as the previous query except that duplicates are eliminated. It is written in a pure pattern form (i.e., without any XPath-like projections)

```
let sel = select t
from <_ ..>[(x::book| _ )*] in [biblio],
     <_ ..>[ t&title _* (<author>[<last>['Buneman']<first>['Peter']]|
<author>[<last>['Suciu'] <first>['Dan']]) _*  ; _]  in x
```

```

```

Note the pattern on the second line in the **from** clause. As the type of an element in **x** is **type book = <book year=String>[title (author+ | editor+) publisher price]**, we skip the tag : **<_ ..>**, then we then capture the corresponding title **(t &title)** then we skip authors **_*** until we find either Peter Buneman or Dan Suciu **(<author>[<last>['Buneman']<first>['Peter']]| <author>[<last>['Suciu'] <first>['Dan']])**, then we skip the remaining authors **_***, and then ignore the tail of the sequence by writing **; _**

Result:

```
val sel : [ title* ] = [ <title>[ 'Data on the Web' ] ]
```

This pure pattern form of the query yields (in general) better performance than the same one written in an XQuery-like programming style. However, the query optimiser automatically translates the latter into a pure pattern one

Joins

This example is the exact transcription of *query Q5 of XQuery use cases*. On top of this section we give the corresponding CDuce types. We give here the type of the document to be joined, and the sample value.

```
type Reviews =<reviews>[Entry*]
type Entry = <entry> [ Title Price Review]
type Title = <title>[PCDATA]
type Price= <price>[PCDATA]
type Review =<review>[PCDATA]

let bstore2 : Reviews =
<reviews>[
    <entry>[
        <title>['Data on the Web']
        <price>['34.95']
        <review>
               ['A very good discussion of semi-structured database
               systems and XML.']
     ]
    <entry>[
        <title>['Advanced Programming in the Unix environment']
        <price>['65.95']
        <review>
               ['A clear and detailed discussion of UNIX programming.']
     ]
    <entry>[
        <title>['TCP/IP Illustrated']
        <price>['65.95']
        <review>
               ['One of the best books on TCP/IP.']
    ]
]
```

The queries are expressed first in an XQuery-like style, then in a pure pattern style: the first pattern-based query is the one produced by the automatic translation from the first one. The last query correponds to a pattern aware programmer's version.

XQuery style

```
<books-with-prices>
select <book-with-price>[t1 <price-bstore1>([p1]/Char)
<price-bstore2>([p2]/Char)]
from b in [biblio]/book ,
     t1 in [b]/title,
     e in [bstore2]/Entry,
     t2 in [e]/Title,
     p2 in [e]/Price,
     p1 in [b]/price
 where t1=t2
```

Automatic translation of the previous query into a pure pattern (thus more efficient) one

```
<books-with-prices>
select <book-with-price>[t1 <price-bstore1>x10 <price-bstore2>x11 ]
from <_ ..>[(x3::book|_)*] in [biblio],
     <_ ..>[(x9::price|x5::title|_)*] in x3,
     t1 in x5,
     <_ ..>[(x6::Entry|_)*] in [bstore2],
     <_ ..>[(x7::Title|x8::Price|_)*] in x6,
     t2 in x7,
     <_ ..>[(x10::Char)*] in x9,
     <_ ..>[(x11::Char)*] in x8
 where t1=t2
```

Pattern aware programmer's version of the same query (hence hand optimised). This version of the query is very efficient. Be aware of patterns.

```
<books-with-prices>
select <book-with-price>[t2 <price-bstore1>p1 <price-bstore2>p2]
from <bib>[b::book*] in [biblio],
     <book ..>[t1&title _* <price>p1] in b,
     <reviews>[e::Entry*] in [bstore2],
     <entry>[t2&Title <price>p2 ;_] in e
where t1=t2
```

More complex Queries: on the power of patterns

```
let biblio = [biblio]/book;;

<bib>
    select <book (a)> x
      from <book (a)>[ (x::(Any\editor)|_ )* ] in biblio
```

This expression returns all book in bib but removoing the editor element. If one wants to write more explicitly:

```
    select <book (a)> x
      from <book (a)>[ (x::(Any\<editor ..>_)|_ )* ] in biblio
```

Or even:

```
    select <book (a)> x
      from <book (a)>[ (x::(<(_\`editor) ..>_)|_ )* ] in biblio
```

Back to the first one:

```
<bib>
    select <book (a)> x
      from <(book) (a)>[ (x::(Any\editor)|_ )* ] in biblio
```

This query takes any element in bib, tranforms it in a book element and removes sub-elements editor, but you will get a warning as capture variable **book** in the **from** is never used: we should have written **<(_) (a)>** instead of **<(book) (a)>** the from

```
    select <(book) (a)> x
      from <(book) (a)>[ (x::(Any\editor)|_ )* ] in biblio
```

Same thing but without tranforming tag to "book". More interestingly:

```
    select <(b) (a\id)> x
      from <(b) (a)>[ (x::(Any\editor)|_ )* ] in biblio
```

removes all "id" attribute (if any) from the attributes of the element in bib.

```
    select <(b) (a\id+{bing=a.id})> x
      from <(b) (a)>[ (x::(Any\editor)|_ )* ] in biblio
```

Changes attribute **id=x** into **bing=x** However, one must be sure that each element in **bib** has an **id** attribute if such is not the case the expression is ill-typed. If one wants to perform this only for those elements which certainly have an **id** attribute then:

```
    select <(b) (a\id+{bing=a.id})> x
      from <(b) (a&{id=_})>[ (x::(Any\editor)|_ )* ] in biblio
```

An unorthodox query: Formatted table generation

The following program generates a 10x10 multiplication table:

```
let bg ((Int , Int) -> String)
  (y, x) -> if (x mod 2 + y mod 2 <= 0) then "lightgreen"
            else if (y mod 2 <= 0) then "yellow"
            else if (x mod 2 <= 0) then "lightblue"
            else "white";;

<table border="1">
  select <tr> select <td align="right" style=("background:"@bg(x,y)) >[
(x*y) ]
               from y in [1 2 3 4 5 6 7 8 9 10] : [1--10*]
  from x in [1 2 3 4 5 6 7 8 9 10] : [1--10*];;
```

The result is the xhtml code that generates the following table:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

8 Higher-order functions

8.1 Introduction

8.2 A complex example

```
type Program = <program>[ Day* ]
type Day = <date day=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
type Author = <author> String
type Title = <title> String

let patch_program
  (p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program] =
  xtransform p with (Invited | Talk) & x -> [ (f x) ]

let first_author ([Program] -> [Program];
                  Invited -> Invited;
                  Talk -> Talk)
| [ Program ] & p -> patch_program (p,first_author)
| <(k)>[ t a _* ] -> <(k)>[ t a ]
```

9 Exercises

9.1 Tree navigation

XPath expressions

Write a function that implements `//t` without using references types and `xtransform`

1. Give a non tail-recursive version
2. Give a tail-recursive version

9.2 Patterns

Sort (by Artur Miguel Diaz: http://ctp.di.fct.unl.pt/~amd)

Write a non recursive function of type `Int -> Latin1` which given a non-negative number produces all its digits in the order.

The function is given below nearly completely programmed. Define the patterns that allows to produce the result.

```
let sortAlg (n :Int):Latin1 =
     match string_of n with
         PATTERN -> RESULT
;;
```

Example:

```
fact 200 =
788657867364790503552363213932185062295135977687173263294742533
244359449963403342920304284011984623904177212138919638830257642
790242637105061926624952829931113462857270763317237396988943922
445621451664240254033291864131227428294853277524242407573903240
321257405579568660226031904170324062351700858796178922222789623
703897374720000000000000000000000000000000000000000000000000000
```

```
sortAlg (fact 200) =
"00000000000000000000000000000000000000000000000000000000000000000000
00000000000001111111111111111111111111222222222222222222222222
22222222222222222222222222222333333333333333333333333333333333
33333333444444444444444444444444444444444455555555555555555555
55555666666666666666666666666666667777777777777777777777777777
77777778888888888888888888888899999999999999999999999999999999"
```

9.3 Solutions

Tree navigation

```
type t =   specify here a type to test

fun ( x :[Any*]):[t*] =
  let f( x :[Any*]):[t*]) = ...
```

Note here that the recursive function **`f`** is wrapped by a second anonymous function so that it does not expose the recursion variable.

```
fun (e : [Any*]):[ T*] =
  let f( accu :[T*] , x :[Any*]):[T*] =
     match x with
        [ h&T&<_ ..>(k&[Any*]) ;t] ->  f( accu@[h], k@t)
      | [ <_ ..>(k&[Any*]) ;t] ->  f( accu, k@t)
      | [ h&T ;t] ->  f( accu@[h], t)
      | [ _ ;t] ->  f( accu, t)
      | [] -> accu
   in f ([], e);;
```

Note that this implementation may generate empty branch warnings in particular

- for the first branch if **`T&<_ ..>(k&[Any*])`** is **`Empty`**
- for the second branch if **`<_ ..>(k&[Any*])`** is smaller than **`T&<_>(k&[Any*])`**
- for the first branch if **`t`** is smaller than **`<_ ..>(k&[Any*])`**

Patterns

```
let sortAlg (n :Int):Latin1 =
    match string_of n with
        [ (s0::'0' | s1::'1' | s2::'2' | s3::'3' | s4::'4' |
           s5::'5' | s6::'6' | s7::'7' | s8::'8' | s9::'9')+ ] ->
                s0 @ s1 @ s2 @ s3 @ s4 @ s5 @ s6 @ s7 @ s8 @ s9
        | _ -> raise "Invalid argument for sortAlg."
;;
```

10 Tutorial Index

This tutorial is under construction!

Sections:

TODO PAGES TABLE OF CONTENTS