The Zen Computational Linguistics Toolkit
Version 2.3.2

December 24th, 2010

Gérard Huet

Copyright © 2002-2010 INRIA

Contents

I Dictionaries **3**

1 Pidgin ML **4**

2 Basic Utilities **5**
- 2.1 Miscellaneous primitives 6
- 2.2 List processing 6
- 2.3 Words 9

3 Zippers **12**
- 3.1 Top-down structures vs bottom-up structures 12
- 3.2 Lists and stacks 12
- 3.3 Contexts as zippers 13
- 3.4 Structured edition on focused trees 15
- 3.5 Zipper operations 16
- 3.6 Zippers as linear maps 17
- 3.7 Zippers for binary trees 18

4 Trie Structures for Lexicon Indexing **21**
- 4.1 Tries as Lexical Trees 21
- 4.2 Ascii encoding 25
- 4.3 Implementing a lexicon as a trie 25
- 4.4 Building a trie lexicon from a byte stream 26

5 Sharing **27**
- 5.1 The Share functor . . . 27
- 5.2 Compressing tries . . . 29
- 5.3 Dagified lexicons . . . 30
- 5.4 Some statistics . . . 31
- 5.5 ISO-LATIN and French . . . 32
- 5.6 Statistics for French . . . 36
- 5.7 Lexicon repositories using tries and decos . . . 37

6 Variation: Ternary trees **37**

7 Decorated Tries for Flexed Forms Storage **42**
- 7.1 Decorated Tries . . . 42
- 7.2 Lexical maps . . . 45
- 7.3 Minimizing lexical maps . . . 48

8 Finite State Machines as Lexicon Morphisms **49**
- 8.1 Finite-state lore . . . 49
- 8.2 Unglueing . . . 50

II Reactive Transducers **56**

9 Introduction **57**

10 A simplistic modular Automaton recognizer **57**
- 10.1 Simplistic aums . . . 57
- 10.2 From automata to reactive processes . . . 58
- 10.3 Dispatching . . . 58
- 10.4 Scheduling and React . . . 59

11 Modular aum transducers **61**

12 Macro-Generation of the *Dispatch* module **66**
- 12.1 Introduction . . . 66
- 12.2 Module Berry_Sethi . . . 67
- 12.3 Module Regexp_system . . . 74
- 12.4 The concrete syntax for modular aums . . . 76
- 12.5 Example: Sanskrit morphology . . . 78
- 12.6 Module Generate_ast . . . 79
- 12.7 Generating the plug-in module . . . 83

13 Producing the engine 85

Abstract

We present in this document a few fundamental structures useful for computational linguistics.

The central structure is that of lexical tree, or *trie*. A crucial observation is that a trie is isomorphic to the state space of a deterministic acyclic automaton. More complex finite-state automata and transducers, deterministic or not, and cyclic or not, may be represented as tries decorated by extra information. Thus we obtain a family of structures underlying lexicon-directed linguistic processes.

First we describe plain tries, which are adequate to represent lexicon indexes. Then we describe decorated tries, or *decos*, which are appropriate to represent symbol tables, and dictionaries associating with the lexicon grammatical or other informations. We then describe how to represent maps and more generally invertible relations between lexicons. We call these structures lexical maps or *lexmaps*. Lexmaps are appropriate for instance to associate flexed forms to lexicon stems and roots, using morphological operations. Such lexmaps are invertible in the sense that we may retrieve from the lexmap entry of a flexed form the stems and operations from which it may be obtained. Finally we show how lexicon directed transducers may be represented using tries decorated with choice points. Such transducers are useful to describe segmentation and taggings processes.

All data structures and algorithms are described in a computational metalanguage called `Pidgin ML`. `Pidgin ML` is a publication language for the ML family of programming languages. All the algorithms described here could be described as well in Standard ML or in Objective CAML, to cite two popular ML implementations, or in the lazy functional language Haskell. They could also be described in a programming language such as LISP or Scheme, but the strong typing discipline of ML, supporting polymorphism and modules, is an insurance that computations cannot corrupt data structures and lead to run-type errors. An initial chapter of these notes gives a quick overview of `Pidgin ML`.

The resulting design may be considered as the reference implementation of a Free Computational Linguistics Toolkit. It may turn useful as an "off the shelf" toolkit for simple operations on linguistics material. Due to its lightweight approach we shall talk of the Zen CL Toolkit.

This toolkit was abstracted from the Sanskrit ML Library, which constitutes its first large-scale application. Thus some of this material already appeared in the documentation of the Sanskrit Segmenter algorithm, which solves Sandhi Analysis [18].

This document was automatically generated from the code of the toolkit using the Ocamlweb package of Jean-Christophe Filliâtre, with the Latex package, in the literate programming style pioneered by Don Knuth.

Part I
Dictionaries

1 Pidgin ML

We shall use as *meta language* for the description of our algorithms a pidgin version of the functional language ML [13, 11, 28, 40]. Readers familiar with ML may skip this section, which gives a crash overview of its syntax and semantics.

Module Pidgin

The core language has types, values, and exceptions. Thus, 1 is a value of predefined type *int*, whereas `"CL"` is a *string*. Pairs of values inhabit the corresponding product type. Thus: $(1,\texttt{"CL"})\ :\ (int\ \times\ string)$. Recursive type declarations create new types, whose values are inductively built from the associated constructors. Thus the Boolean type could be declared as a sum by:

$\mathsf{type}\ bool = [True\ \mid\ False];$

Parametric types give rise to polymorphism. Thus if x is of type t and l is of type $(list\ t)$, we construct the list adding x to l as $[x\ ::\ l]$. The empty list is $[\,]$, of (polymorphic) type $(list\ \alpha)$. Although the language is strongly typed, explicit type specification is rarely needed from the designer, since principal types may be inferred mechanically.

The language is functional in the sense that functions are first class objects. Thus the doubling integer function may be written as $\mathsf{fun}\ x\ \rightarrow\ x+x$, and it has type $int \rightarrow\ int$. It may be associated to the name *double* by declaring:

$value\ double\ =\ \mathsf{fun}\ x\ \rightarrow\ x+x;$

Equivalently we could write:

$value\ double\ x\ =\ x+x;$

Its application to value n is written as $(double\ n)$ or even $double\ n$ when there is no ambiguity. Application associates to the left, and thus $f\ x\ y$ stands for $((f\ x)\ y)$. Recursive functional values are declared with the keyword rec. Thus we may define factorial as:

$value\ \mathsf{rec}\ fact\ n\ =\ \mathsf{if}\ n=0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times (fact\ (n-1));$

Functions may be defined by pattern matching. Thus the first projection of pairs could be defined by:

$value\ fst\ =\ \mathsf{fun}\ [\ (x,y)\ \rightarrow\ x\ ];$

or equivalently (since there is only one pattern in this case) by:

value fst $(x, y) = x$;

Pattern-matching is also usable in match expressions which generalize case analysis, such as: match l with [[] → *True* | _ → *False*], which tests whether list l is empty, using underscore as catch-all pattern.

Evaluation is strict, which means that x is evaluated before f in the evaluation of $(f\ x)$. The let expressions allow the sharing of sub-computations. Thus let x = *fact* 10 in $x + x$ will compute *fact* 10 first, and only once. An equivalent postfix *where* notation may be used as well. Thus the conditional expression if b then *e1* else *e2* is equivalent to:
choose b *where choose* = fun [*True* → *e1* | *False* → *e2*];
Exceptions are declared with the type of their parameters, like in:

exception *Failure* of *string*;

An exceptional value may be raised, like in: *raise* (*Failure* `"div␣0"`) and handled by a try switch on exception patterns, such as:
try *expression* with [*Failure s* → ...];]
Other imperative constructs may be used, such as references, mutable arrays, while loops and I/O commands, but we shall seldom need them. Sequences of instructions are evaluated in left to right regime in do expressions, such as: do {*e1*; ... *en*}.

ML is a *modular* language, in the sense that sequences of type, value and exception declarations may be packed in a structural unit called a module, amenable to separate treatment. Modules have types themselves, called *signatures*. Parametric modules are called *functors*. The algorithms presented in this paper will use in essential ways this modularity structure, but the syntax ought to be self-evident. Finally, comments are enclosed within starred parens like:

value s = `"This␣is␣a␣string"`; (∗ This is a comment ∗)

Readers not acquainted with programming languages may think of ML definitions as recursive equations over inductively defined algebras. Most of them are simple primitive recursive functionals. The more complex recursions of our automata coroutines will be shown to be well-founded by a combination of lexicographic and multiset orderings.

Pidgin ML definitions may actually be directly executed as Objective Caml programs [26], under the so-called revised syntax [30]. The following development may thus be used as the reference implementation of a core computational linguistics platform, dealing with lexical, phonemic and morphological aspects.

2 Basic Utilities

We present in this section some basic utilities libraries.

2.1 Miscellaneous primitives

Module Gen

This module contains various utilities of general use.

```
value dirac b = if b then 1 else 0
;
value optional f = fun [ None → () | Some d → f d ]
;
value active = fun [ None → False | Some _ → True ]
;
```

Dump value v on *file*.

```
value dump v file =
  let cho = open_out file in do { output_value cho v; close_out cho }
;
```

Retrieve value dumped on *file*; its type should be given in a cast.

```
value gobble file =
  let chi = open_in file in
  let v = input_value chi in do { close_in chi; v }
;
```

UNIX touch.

```
value touch file = close_out (open_out file)
;
value notify_error message =
  do { output_string stderr message; flush stderr }
;
```

2.2 List processing

We shall use lists intensively. We assume the standard library *List*.

Module List2

We complement *List* here with a few auxiliary list service functions.
unstack l r = (rev l) @ r
unstack = List.rev_append

```
value rec unstack l1 l2 =
  match l1 with
  [ [] → l2
  | [ a :: l ] → unstack l [ a :: l2 ]
  ]
;
value non_empty = fun [ [] → False | _ → True ]
;
```

Set operations with lists.

```
exception Twice_the_same_value
;
value union1 e l =
  if List.mem e l then l else [ e :: l ]
;
```

Used in ZEN/Lexmap.

```
value union2 e l =
   if List.mem e l then (raise Twice_the_same_value) else [ e :: l ]
;
```

Terminal recursive union of finite sets represented as as lists - does not respect the order of elements in *l1*: *union_f* [1; 2] [] = [2; 1]

```
value rec union_f l1 l2 =
  match l1 with
    [ [] → l2
    | [ e :: l ] → union_f l (union1 e l2)
    ]
;
```

Same, respecting the order:

```
value union l1 l2 = List.fold_right union1 l1 l2
;
value set_of l = let add acc x = if List.mem x acc then acc else [ x :: acc ]
  in List.fold_left add [] l
;
```

last : *list* $\alpha \rightarrow \alpha$

```
value rec last = fun
  [ [] → raise (Failure "last")
  | [ x ] → x
  | [ _ :: l ] → last l
  ]
;
```

truncate n l removes from *l* its initial sublist of length *n*.
$truncate : int \rightarrow list\ \alpha \rightarrow list\ \alpha$

```
value rec truncate n l =
  if n = 0 then l else match l with
   [ [] → failwith "truncate"
   | [ _ :: r ] → truncate (n − 1) r
   ]
;
type ranked α = list (int × α)
;
```

zip n l assumes *l* sorted in increasing order of ranks; it returns a partition of *l* as $(l1, l2)$ with *l1* maximum such that ranks in *l1* are $< n$. *l1* is reversed, i.e. we enforce the invariant: $zip\ n\ l = (l1, l2)$ such that $l = unstack\ l1\ l2$.
$zip : int \rightarrow (ranked\ \alpha) \rightarrow ((ranked\ \alpha) \times (ranked\ \alpha))$

```
value zip n = zip_rec []
  where rec zip_rec acc l = match l with
  [ [] → (acc, [])
  | [ ((m, _) as current) :: rest ] →
       if m < n then zip_rec [ current :: acc ] rest
       else (acc, l)
  ]
;
```

Coercions between *string* and *list char*.
$explode : string \rightarrow list\ char$

```
value explode s =
  let rec expl i accu =
     if i < 0 then accu else expl (i − 1) [ s.[i] :: accu ] in
  expl (String.length s − 1) []
;
```

$implode : list\ char \rightarrow string$

```
value implode l =
  let result = String.create (List.length l) in
  let rec loop i = fun
      [ [] → result
      | [ c :: cs ] → do { String.set result i c; loop (i + 1) cs }
      ] in
  loop 0 l
;
```

Process a list with using *pr* for elements and *sep* for separator
process_list_sep : $(\alpha \rightarrow unit) \rightarrow (unit \rightarrow unit) \rightarrow list\ \alpha \rightarrow unit$

```
value process_list_sep pr sep =
  let rec prl = fun
      [ [] → ()
      | [ s ] → pr s
      | [ s :: ls ] → do { pr s; sep (); prl ls }
      ] in
  prl
;
```

Insert in a list of buckets with increasing keys

```
value in_bucket key element buckets = in_rec [] buckets
  where rec in_rec accu buckets = match buckets with
      [ [] → unstack accu [ (key, [ element ]) ]
      | [ ((k, l) as bucket) :: rest ] →
          if k > key then unstack accu [ (key, [ element ]) :: buckets ]
          else if k = key then unstack accu [ (k, [ element :: l ]) :: buckets ]
          else in_rec [ bucket :: accu ] rest
      ]
;
```

2.3 Words

We assume that the alphabet of string representations is some initial segment of positive integers. Thus a string is coded as a list of integers which will from now on be called a *word*.

For instance, for our Sanskrit application, the Sanskrit alphabet comprises 50 letters, representing 50 phonemes. Finite state transducers convert back and forth lists of such integers into strings of transliterations in the roman alphabet, which encode themselves either letters with diacritics, or Unicode representations of the *devanāgarī* alphabet. Thus 1,2,3,4 etc encode respectively the phonemes /a/, /ā/, /i/, /ī/ etc.

In these notes, we shall assume rather a roman alphabet, and thus 1,2,3,4 etc encode respectively letters a, b, c, d etc.

Module Word

```
type letter = int
and word = list letter (* word encoded as sequence of natural numbers *)
;
```

We remark that we are not using for our word representations the ML type of strings (which in OCaml are arrays of characters/bytes). Strings are convenient for English texts (using the 7-bit low half of ASCII) or other European languages (using the ISO-LATIN subsets of full ASCII), and they are more compact than lists of integers, but basic operations like pattern matching are awkward, and they limit the size of the alphabet to 256, which is insufficient for the treatment of many languages' written representations. New format standards such as Unicode have complex primitives for their manipulation, and are better reserved for interface modules than for central morphological operations. We could have used an abstract type of characters, leaving to module instantiation their precise definition, but here we chose the simple solution of using machine integers for their representation, which is sufficient for large alphabets (in Ocaml, this limits the alphabet size to 1073741823), and to use conversion functions *encode* and *decode* between words and strings. In the Sanskrit application, we use the first 50 natural numbers as the character codes of the Sanskrit phonemes, whereas string translations take care of roman diacritics notations, and encodings of *devanāgarī* characters.
prefix : *word* → *word* → *bool*

```
value rec prefix u v =
  match u with
    [ [] → True
    | [ a :: r ] → match v with
        [ [] → False
        | [ b :: s ] → a = b ∧ prefix r s
        ]
    ]
;
```

Lexicographic ordering on words.
lexico : *word* → *word* → *bool*

```
value rec lexico l1 l2 = match l1 with
  [ [] → True
  | [ c1 :: r1 ] → match l2 with
      [ [] → False
```

```
      | [ c2 :: r2 ] → if c2 < c1 then False
                        else if c2 = c1 then lexico r1 r2
                             else True
      ]
  ]
;
```

Differential words.
A *differential word* is a notation permitting to retrieve a word w from another word w' sharing a common prefix. It denotes the minimal path connecting the words in a trie, as a sequence of ups and downs: if $d = (n, u)$ we go up n times and then down along word u.

```
type delta = (int × word) (* differential words *)
;
```

Natural ordering on differential words.

```
value less_diff (n1, w1) (n2, w2) = n1 < n2 ∨ (n1 = n2) ∧ lexico w1 w2
;
```

We compute the difference between w and w' as a differential word $diff\ w\ w' = (|\ w1\ |, w2)$ where $w = p.w1$ and $w' = p.w2$, with maximal common prefix p.
diff : *word* → *word* → *delta*

```
value rec diff = fun
  [ [] → fun x → (0, x)
  | [ c :: r ] as w → fun
      [ [] → (List.length w, [])
      | [ c' :: r' ] as w' → if c = c' then diff r r'
                              else (List.length w, w')
      ]
  ]
;
```

Now w' may be retrieved from w and $d = diff\ w\ w'$ as $w' = patch\ d\ w$.
patch : *delta* → *word* → *word*

```
value patch (n, w2) w =
  let p = List2.truncate n (List.rev w) in
  List2.unstack p w2
;
```

3 Zippers

Zippers encode the context in which some substructure is embedded. They are used to implement applicatively destructive operations in mutable data structures. They are also used to navigate in complex data structures, such as state spaces of non-deterministic search processes, while keeping operations local and preserving sharing.

3.1 Top-down structures vs bottom-up structures

We understand well top-down structures. They are the representations of initial algebra values. For instance, the structure *bool* has two constant constructors, the booleans $True$ and $False$. The polymorphic structure $list\ \alpha$ admits two constructors, the empty list [] and the list constructor consing a value $x : \alpha$ to a homogeneous list $l : list\ \alpha$ to form $[a :: l] : list\ \alpha$.

Bottom-up structures are useful for creating, editing, traversing and changing top-down structures in a local but applicative manner. They are sometimes called computation contexts, or recursion structures. We shall call them *zippers*, following [14].

Top-down structures are the finite elements inhabiting inductively defined types. Bottom-up structures are also finite, but they permit the progressive definition of (potentially infinite) values of co-inductive types. They permit incremental navigation and modification of very general data types values. We shall also see that they model linear structural functions, in the sense of linear logic.

Finally, bottom-up computing is the right way to build shared structures in an applicative fashion, opening the optimisation path from trees to dags. Binding algebras (λ-calculus expressions for inductive values and Böhm trees for the co-inductive ones) may be defined by either de Bruijn indices or higher-order abstract syntax, and general graph structures may be represented by some spanning tree decorated with virtual adresses, so we see no reason to keep explicit references and pointer objects, with all the catastrophies they are liable for, and we shall stick to purely applicative programming.

3.2 Lists and stacks

Lists are first-in first-out sequences (top-down) whereas stacks are last-in first-out sequences (bottom-up). They are not clearly distinguished in usual programming, because the underlying data structure is the same : the list $[x_1;\ x_2; ...\ x_n]$ may be reversed into the stack $[x_n\ ...; x_2; x_1]$ which is of the same *type* list. So we cannot expect to capture their difference with the type discipline of ML. At best by declaring:
$type\ stack\ \alpha = list\ \alpha;$
we may use type annotations to document whether a given list is used by a function in the rôle of a list or of a stack. But such *intentions* are not enforced by ML's type system,

which just uses freely the type declaration above as an equivalence. So we have to check these intentions carefully, if we want our values to come in the right order. But we certainly wish to distinguish lists and stacks, since stacks are built and analysed in unit time, whereas adding a new element to a list is proportional to the length of the list.

A typical exemple of stack use is *List*2.*unstack* above. In (*unstack* l s), s is an accumulator stack, where values are listed in the opposite order as they are in list l. Indeed, we may define the reverse operation on lists as:

value *rev* l = *unstack* l [];

In the standard Ocaml's library, *unstack* is called *rev_append*. It is efficient, since it is *tail recursive*: no intermediate values of computation need to be kept on the recursion stack, and the recursion is executed as a mere jump. It is much more efficient, if some list l_1 is kept in its reversed stack form s_1, to obtain the result of appending l_1 to l_2 by calling *rev_append* s_1 l_2 than to call *append* l_1 l_2, which amounts to first reversing l_1 into s_1, and then doing the same computation. Similarly, the *List* library defines a function *rev_map* which is more efficient than *map*, if one keeps in mind that its result is the stack order. But no real discipline of these library functions is really enforced.

Here we want to make this distinction precise, favor local operations, and delay as much as possible any reversal. For instance, if some list l_1 is kept in its reversed stack form s_1, and we wish to append list l_2 to it, the best is to just wait and keep the pair (s_1, l_2) as the state of computation where we have l_2 *in the context* s_1. In this computation state, we may finish the construction of the result l of appending l_1 to l_2 by "zipping up" l_1 with *unstack* s_1 l_2, or we may choose rather to "zip down" l_2 with *unstack* l_2 s_1 to get the stack context value *rev* l. But we may also consider that the computation state (s_1, l_2) represents l locally accessed as its prefix l_1 stacked in context value s_1 followed by its suffix l_2. And it is very easy to insert as this point a new element x, either stacked upwards in state $([x :: s_1], l_2)$, or consed downwards in state $(s_1, [x :: l_2])$.

Once this intentional programming methodology of keeping focused structures as pairs $(context, substructure)$ is clear, it is very easy to understand the generalisation to zippers, which are to general tree structures what stacks are to lists, i.e. upside-down access representations of (unary) contexts.

3.3 Contexts as zippers

Module Zipper

We start with ordered trees. We assume the mutual inductive types:

type *tree* = [*Tree* of *arcs*]
and *arcs* = *list tree*

;

The tree zippers are the contexts of a place holder in the arcs, that is linked to its left siblings, right siblings, and parent context:

```
type tree_zipper =
  [ Top
  | Zip of (arcs × tree_zipper × arcs)
  ]
;
```

Let us model access paths in trees by sequences on natural numbers naming the successive arcs 1, 2, etc.

```
type access = list int
and domain = list access
;
```

We usually define the domain of a tree as the set of accesses of its subterms:
dom : *tree* → *domain*

```
value rec dom = fun
  [ Tree arcs →
     let doms = List.map dom arcs in
     let f (n, d) dn = let ds = List.map (fun u → [ n :: u ]) dn
                        in (n + 1, List2.unstack ds d) in
     let (_, d) = List.fold_left f (1, [ [] ]) doms in List.rev d
  ]
;
```

Thus, we get for instance:

```
value tree0 = Tree [Tree [Tree []; Tree []]; Tree []]
;
dom(tree0)
;
```

→ [[]; [1]; [1; 1]; [1; 2]; [2]] : *domain*
Now if $rev(u)$ is in $dom(t)$, we may zip-down t along u by changing focus, as follows:

```
type focused_tree = (tree_zipper × tree)
;
value nth_context n = nthc n []
  where rec nthc n l = fun
    [ [] → raise (Failure "out of domain")
    | [ x :: r ] → if n = 1 then (l, x, r) else nthc (n − 1) [ x :: l] r
```

```
    ]
;
value rec enter u t = match u with
  [ [] → ((Top, t) : focused_tree)
  | [ n :: l ] → let (z, t1) = enter l t
                  in match t1 with
    [ Tree arcs → let (l, t2, r) = nth_context n arcs
                   in (Zip (l, z, r), t2)
    ]
  ]
;
```

and now we may for instance navigate in *tree0*:

```
enter [2; 1] tree0
;
 → (Zip ([Tree []], Zip ([], Top, [Tree []]), []), Tree []) : focused_tree
```

3.4 Structured edition on focused trees

We shall not explicitly use these access stacks and the function *enter*; these access stacks are implicit from the zipper structure, and we shall navigate in focused trees one step at a time, using the following structure editor primitives on focused trees.

```
value down (z, t) = match t with
    [ Tree arcs → match arcs with
       [ [] → raise (Failure "down")
       | [ hd :: tl ] → (Zip ([], z, tl), hd)
       ]
    ]
;
value up (z, t) = match z with
    [ Top → raise (Failure "up")
    | Zip (l, u, r) → (u, Tree (List2.unstack l [ t :: r ]))
    ]
;
value left (z, t) = match z with
    [ Top → raise (Failure "left")
    | Zip (l, u, r) → match l with
        [ [] → raise (Failure "left")
        | [ elder :: elders ] → (Zip (elders, u, [ t :: r ]), elder)
```

```
        ]
    ]
;
value right (z, t) = match z with
    [ Top → raise (Failure "right")
    | Zip (l, u, r) → match r with
        [ [] → raise (Failure "right")
        | [ younger :: youngers ] → (Zip ([ t :: l ], u, youngers), younger)
        ]
    ]
;
value del_l (z, _) = match z with
    [ Top → raise (Failure "del_l")
    | Zip (l, u, r) → match l with
        [ [] → raise (Failure "del_l")
        | [ elder :: elders ] → (Zip (elders, u, r), elder)
        ]
    ]
;
value del_r (z, _) = match z with
    [ Top → raise (Failure "del_r")
    | Zip (l, u, r) → match r with
        [ [] → raise (Failure "del_r")
        | [ younger :: youngers ] → (Zip (l, u, youngers), younger)
        ]
    ]
;
value replace (z, _) t = (z, t)
;
```

Note how *replace* is a local operation, even though all our programming is applicative.

3.5 Zipper operations

The editing operations above are operations on a finite tree represented at a focus. But we may also define operations on zippers alone, which may be thought of as operations on a potentially infinite tree, actually on all trees, finite or infinite, having this initial context. That is, focused trees as pairs (context,structure) refer to finite elements (inductive values), whereas contexts may be seen as finite approximations to streams (co-inductive values), for instance generated by a process. For example, here is an interpreter that takes a command to build progressively a zipper context:

```
type context_construction =
  [ Down | Left of tree | Right of tree ]
;
value build z = fun
  [ Down → Zip ([], z, [])
  | Left t → match z with
      [ Top → raise (Failure "build␣Left")
      | Zip (l, u, r) → Zip ([ t :: l ], u, r)
      ]
  | Right t → match z with
      [ Top → raise (Failure "build␣Right")
      | Zip (l, u, r) → Zip (l, u, [ t :: r ])
      ]
  ]
;
```

But we could also add to our commands some destructive operations, to delete the left or right sibling, or to pop to the upper context.

3.6 Zippers as linear maps

We developed the idea that zippers were dual to trees in the sense that they may be used to represent the approximations to the coinductive structures corresponding to trees as inductive structures. We shall now develop the idea that zippers may be seen as linear maps over trees, in the sense of linear logic. In the same way that a stack *st* may be thought of as a representation of the function which, given a list l, returns the list *unstack st* l, a zipper z may be thought of as the function which, given a tree t, returns the tree *zip_up* z t, with:

```
value rec zip_up z t = match z with
  [ Top → t
  | Zip (l, up, r) → zip_up up (Tree (List2.unstack l [ t :: r ]))
  ]
;
```

Thus *zip_up* may be seen as a coercion between a zipper and a map from trees to trees, which is linear by construction.
Alternatively to computing *zip_up* z t, we could of course just build the focused tree (z, t), which is a "soft" representation which could be rolled in into *zip_up* z t if an actual term is needed later on.
Applying a zipper to a term is akin to substituting the term in the place holder represented by the zipper. If we substitute another zipper, we obtain zipper composition, as follows. First, we define the reverse of a zipper:

```
value rec zip_unstack z1 z2 = match z1 with
  [ Top → z2
  | Zip (l, z, r) → zip_unstack z (Zip (l, z2, r))
  ]
;
value zip_rev z = zip_unstack z Top
;
```

And now composition is similar to concatenation of lists:

```
value compose z1 z2 =
  zip_unstack (zip_rev z2) z1
;
```

It is easy to show that *Top* is an identity on the left and on the right for composition, and that composition is associative. Thus we get a category, whose objects are trees and morphisms are zippers, which we call the Zipper category of linear tree maps.
We end this section by pointing out that tree splicing, or *adjunction* in the terminology of Tree Adjoint Grammars, is very naturally expressible in this framework. Indeed, what is called a rooted tree in this tradition is here directly expressed as a zipper `zroot`, and adjunction at a tree occurrence is prepared by decomposing this tree at the given occurrence as a focused tree (z, t). Now the adjunction of *zroot* at this occurrence is simply computed as:

```
value splice_down (z, t) zroot = (compose z zroot, t)
;
```

if the focus of attention stays at the subtree t, or

```
value splice_up (z, t) zroot = (z, zip_up zroot t)
;
```

if we want the focus of attention to stay at the adjunction occurrence. These two points of view lead to equivalent structures, in the sense of tree identity modulo focusing:

```
value equiv (z, t) (z', t') = (zip_up z t = zip_up z' t')
;
```

3.7 Zippers for binary trees

We end this section by showing the special case of zippers for binary trees.

Module Bintree

```
type bintree =
  [ Null
  | Bin of (bintree × bintree)
  ]
;
```

Occurrences as boolean lists (binary words).

```
type binocc = list bool
and domain = list binocc
;
```

binlexico : *binocc* → *binocc* → *bool*

```
value rec binlexico l1 l2 = match l1 with
  [ [] → True
  | [ b1 :: r1 ] → match l2 with
      [ [] → False
      | [ b2 :: r2 ] → if b1 = b2 then binlexico r1 r2 else b2
      ]
  ]
;
```

occurs : *binocc* → *bintree* → *bool*

```
value rec occurs occ bt = match occ with
  [ [] → True
  | [ b :: rest ] → match bt with
     [ Null → False
     | Bin (bl, br) → occurs rest (if b then br else bl)
     ]
  ]
;
```

paths : *bintree* → *domain*

```
value paths = pathrec [] []
  where rec pathrec acc occ = fun
  [ Null → [ List.rev occ :: acc ]
  | Bin (bl, br) → let right = pathrec acc [ True :: occ ] br
                   in [ List.rev occ :: pathrec right [ False :: occ ] bl ]
  ]
;
```

Note: $occurs\ occ\ t\ =\ List.mem\ occ\ (paths\ t)$. We assume $paths\ t$ to be in *binlexico* order.
$bintree_of1\ :\ binocc\ \rightarrow\ bintree$

```
value rec bintree_of1 = fun
  [ [] → Null
  | [ b :: occ ] → if b then Bin (Null, bintree_of1 occ)
                        else Bin (bintree_of1 occ, Null)
  ]
;
```

Zippers

binary contexts = linear bintree maps

```
type binzip =
  [ Top
  | Left of (binzip × bintree)
  | Right of (bintree × binzip)
  ]
;
```

$zip_up\ :\ binzip\ \rightarrow\ bintree\ \rightarrow\ bintree$

```
value rec zip_up z bt = match z with
  [ Top → bt
  | Left (up, br) → zip_up up (Bin (bt, br))
  | Right (bl, up) → zip_up up (Bin (bl, bt))
  ]
;
```

$extend\ :\ bintree\ \rightarrow\ binocc\ \rightarrow\ bintree$

```
value extend tree = enter_edit Top tree
  where rec enter_edit z t occ = match occ with
      [ [] → zip_up z t
      | [ b :: rest ] → match t with
          [ Bin (bl, br) → if b then enter_edit (Right (bl, z)) br rest
                                else enter_edit (Left (z, br)) bl rest
          | Null → zip_up z (bintree_of1 occ)
          ]
      ]
;
```

We maintain $extend\ t\ occ\ =$ if $occurs\ occ\ t$ then t else $bintree_of\ [\ occ\ ::\ paths\ t\ ]$.
$bintree_of\ :\ domain\ \rightarrow\ bintree$

```
value bintree_of = binrec Null
  where rec binrec acc = fun
  [ [] → acc
  | [ occ :: dom ] → binrec (extend acc occ) dom
  ]
;
```

Invariants:

- $paths\ (bintree_of\ \ dom) = \{occ \mid binlexico\ occ\ o$ with $o \in dom\}$
- $bintree_of\ (paths\ tree) = tree$
- $bintree_of1\ \ occ = bintree_of\ [occ]$

4 Trie Structures for Lexicon Indexing

Tries are tree structures that store finite sets of strings sharing initial prefixes.

4.1 Tries as Lexical Trees

Tries (also called *lexical trees*) may be implemented in various ways. A node in a trie represents a string, which may or may not belong to the set of strings encoded in the trie, together with the set of tries of all suffixes of strings in the set having this string as a prefix. The forest of sibling tries at a given level may be stored as an array, or as a list if we assume a sparse representation. It could also use any of the more efficient representations of finite sets, such as search trees [3]. Here we shall assume the simple sparse representation with lists (which is actually the original presentation of tries by René de la Briantais (1959)), yielding the following inductive type structure.

Module Trie

Tries store sparse sets of words sharing initial prefixes.

```
type trie = [ Trie of (bool × arcs) ]
and arcs = list (Word.letter × trie)
;
```

Trie (b, l) stores the empty word [ ] iff b, and all the words of arcs in l, while the arc (n, t) stores all words [n :: c] for c a word stored in t.

Note that letters decorate the *arcs* of the *trie*, not its nodes. For instance, the trie storing the set of words [[1]; [2]; [2; 2]; [2; 3]] is represented as
Trie (*False*, [(1, *Trie* (*True*, [ ])); (2, *Trie* (*True*, [(2, *Trie* (*True*, [ ])); (3, *Trie* (*True*, [ ]))]))]) .

This example exhibits one invariant of our representation, namely that the integers in successive sibling nodes are in increasing order. Thus a top-down left-to-right traversal of the *trie* lists its strings in lexicographic order. The algorithms below maintain this invariant.

Zippers as Trie contexts.

Let us show how to add words to a trie in a completely applicative way, using the notion of a trie zipper.

```
type zipper =
  [ Top
  | Zip of (bool × arcs × Word.letter × arcs × zipper)
  ]
and edit_state = (zipper × trie)
;
```

An *edit_state* $(z, t0)$ stores the editing context as a zipper z and the current subtrie $t0$. We replace this subtrie by a trie t by closing the zipper with *zip_up* t z as follows.

```
exception Redundancy
;

zip_up : zipper → trie → trie

value rec zip_up z t = match z with
  [ Top → t
  | Zip (b, left, n, right, up) →
      zip_up up (Trie (b, List2.unstack left [ (n, t) :: right ]))
  ]
;
```

We need two auxiliary routines. The first one, *zip*, was given in module *List2*. Its name stems from the fact that it looks for an element in an a-list while building an editing context in the spirit of a zipper, the role of *zip_up* being played by *unstack*. The second routine, given a word w, returns the singleton filiform trie containing w as *trie_of* w.

```
trie_of : word → trie

value rec trie_of = fun
  [ [ ] → Trie (True, [ ])
  | [ n :: rest ] → Trie (False, [ (n, trie_of rest) ])
  ]
;
```

Insertion and lookup.

We are now ready to define the insertion algorithm:

```
enter : trie → word → trie

value enter trie = enter_edit (Top, trie)
  where rec enter_edit (z, t) = match t with
    [ Trie (b, l) → fun
      [ [] → if b then raise Redundancy
               else zip_up z (Trie (True, l))
      | [ n :: rest ] →
          let (left, right) = List2.zip n l in
          match right with
            [ [] → zip_up (Zip (b, left, n, [], z)) (trie_of rest)
            | [ (m, u) :: r ] →
              if m = n then enter_edit (Zip (b, left, n, r, z), u) rest
              else zip_up (Zip (b, left, n, right, z)) (trie_of rest)
            ]
      ]
    ]
;

contents : trie → list word
```

Note that *contents* lists words in lexicographic order.
It should be used only on small lexicons.

```
value contents = contents_prefix []
  where rec contents_prefix pref = fun
    [ Trie (b, l) →
      let down = let f l (n, t) = l @ (contents_prefix [ n :: pref ] t) in
                 List.fold_left f [] l in
      if b then [ (List.rev pref) :: down ] else down
    ]
;

mem : word → trie → bool

value rec mem w = fun
  [ Trie (b, l) → match w with
    [ [] → b
    | [ n :: r ] → try mem r (List.assoc n l)
                    with [ Not_found → False ]
    ]
  ]
;
```

Tries may be considered as deterministic finite state automata graphs for accepting the (finite) language they represent. This remark is the basis for many lexicon processing libraries. Actually, the *mem* algorithm may be seen as an interpreter for such an automaton, taking its state graph as its trie argument, and its input tape as its word one. The boolean information in a trie node indicates whether or not this node represents an accepting state. These automata are not minimal, since while they share initial equivalent states, there is no sharing of accepting paths, for which a refinement of lexical trees into dags is necessary. We shall look at this problem in the next section. First we give the rest of the *Trie* module.

```
value empty = Trie (False, [])
;
```

next_trie returns the first element of its *trie* argument.

```
value next_trie = next_rec []
  where rec next_rec acc = fun
    [ Trie (b, l) →
      if b then List.rev acc
      else match l with
            [ [] → failwith "next_trie"
            | [ (n, u) :: _ ] → next_rec [ n :: acc ] u
            ]
    ]
;
```

last_trie returns the last element of its *trie* argument.

```
value last_trie = last_rec []
   where rec last_rec acc = fun
     [ Trie (b, l) → match l with
        [ [] → if b then List.rev acc else failwith "last_trie"
        | _ → let (n, u) = List2.last l in
                last_rec [ n :: acc ] u
        ]
     ]
;
```

size trie is the number of words stored in *trie*.

```
value rec size = fun
  [ Trie (b, arcs) →
     let s = List.fold_left count 0 arcs
               where count n (_, t) = n + size t in
     s + Gen.dirac b
  ]
```

```
;
```

A *trie* iterator

```
iter : (word → unit) → trie → unit

value iter f t = iter_prefix [] t
  where rec iter_prefix pref = fun
    [ Trie (b, arcs) → do
        { if b then f (List.rev pref) else ()
        ; let phi (n, u) = iter_prefix [ n :: pref ] u in
          List.iter phi arcs
        }
    ]
;
```

4.2 Ascii encoding

The *Ascii* module defines coercions *encode* from strings to words and *decode* from words to strings.

Module Ascii

A very simple encoding scheme : ASCII

```
encode : string → word
decode : word → string

value encode string = List.map int_of_char (List2.explode string)
and decode word = List2.implode (List.map char_of_int word)
;
```

4.3 Implementing a lexicon as a trie

Now, using the coercion *encode* from strings to words from the *Ascii* module, we build a lexicon trie from a list of strings by function *make_lex*, using Ocaml's *fold_left* from the *List* library (the terminal recursive list iterator).

Module Lexicon

make_lex raises *Redundancy* if duplicate elements in its argument.

```
make_lex : list string → trie
```

```
value make_lex =
  List.fold_left (fun lex c → Trie.enter lex (Ascii.encode c)) Trie.empty
;
strings_of : trie → list string
value strings_of t = List.map Ascii.decode (Trie.contents t)
;
```

strings_of (*make_lex* *l*) gives *l* in lexicographic order.

```
assert (strings_of (make_lex [ "a"; "b"; "ab" ]) = [ "a"; "ab"; "b" ])
;
```

4.4 Building a trie lexicon from a byte stream

The function *trie_of_strings* reads on its standard input a stream of strings separated by newline characters, builds the corresponding trie lexicon, and writes its representation on its standard output.

It depends on a module *Encoding*, which defines the string encoding used through conversion functions *encode* and *decode*.

Module Make_lex

Trie lexicon building from text file containing lists of words

```
module Make_lex (Encoding : sig
  value encode : string → Word.word;
  value decode : Word.word → string; end) = struct
value lexicon = ref Trie.empty
;
value trie_of_strings =
  let lexicon = ref Trie.empty in process_strings
    where rec process_strings () =
      try while True do
        { let str = read_line () in
          lexicon.val := Trie.enter lexicon.val (Encoding.encode str)
        }
      with [ End_of_file → output_value stdout lexicon.val
           | Trie.Redundancy → output_string stderr "Fatal error: duplicated word\n"
           ]
;
```

```
end
;
```

For instance, with `english.lst` storing a list of 173528 English words, as a text file of size 2Mb, the command `make_lex < english.lst > english.rem` produces a trie representation as a file of 4.5Mb. Obviously we are wasting storage because we create a huge structure which shares the words along with their common initial prefixes, but which ignores the potential space saving of sharing common suffixes. We shall develop such sharing in a completely generic manner, as follows.

5 Sharing

Sharing data representation is a very general problem. Sharing identical representations is ultimately the responsibility of the runtime system, which allocates and desallocates data with dynamic memory management processes such as garbage collectors.

But sharing of representations of the same type may also be programmed by bottom-up computation. All that is needed is a memo function building the corresponding map without duplications. Let us show the generic algorithm, as an ML *functor*.

5.1 The Share functor

This functor (that is, parametric module) takes as parameter an algebra with its domain seen here as an abstract type. Here is its public interface declaration:

Interface for module Share

```
module Share : functor (Algebra :sig type domain; value size : int; end)
→ sig value share : Algebra.domain → int → Algebra.domain;
        value reset : unit → unit;
        value memo : array (list Algebra.domain); (* for debugging *)
    end;
```

Module Share

```
module Share (Algebra : sig type domain; value size : int; end) = struct
```

Share takes as argument a module *Algebra* providing a type *domain* and an integer value *size*, and it defines a value *share* of the stated type. We assume that the elements from the

domain are presented with an integer key bounded by *Algebra.size*. That is, *share x k* will assume as precondition that $0 \leq k < Max$ with Max =*Algebra.size*.
We shall construct the sharing map with the help of a hash table, made up of buckets $(k, [e_1; e_2; ...e_n])$ where each element e_i has key k.

```
type bucket = list Algebra.domain
;
```

A bucket stores a set e of elements of domain of a given key these sets are here implemented as lists invariant : *e* = [*e_1*; ... *e_n*] with *e_i* = *e_j* only if *i* = *j*. That is, a bucket consists of distinct elements.
The memory is a hash-table of a given size and of the right bucket type.

```
value memo = Array.create Algebra.size ([] : bucket)
;
```

Resetting the hash-table

```
value reset () = for i = 0 to Algebra.size − 1 do { memo.(i) := [] }
;
```

We shall use a service function *search*, such that *search e l* returns the first y in l such that $y = e$ or or else raises the exception *Not_found*.
Note *search e* = *List.find* (fun $x \rightarrow x = e$).

```
value search e = searchrec
  where rec searchrec = fun
   [ [] → raise Not_found
   | [ x :: l ] → if x = e then x else searchrec l
   ]
;
```

Now *share x k*, where k is the key of x, looks in k-th bucket l (this is meaningful since we assume that the key fits in the size: 0≤ k < *Algebra.size*) and returns y in l such that $y = x$ if it exists, and otherwise returns x memorized in the new k-th bucket [x :: e]. Since *share* is the only operation on buckets, we maintain that such y is unique in its bucket when it exists.

```
value share element key = (∗ assert 0 ≤ key < Algebra.size ∗)
  let bucket = memo.(key) in
  try search element bucket with
    [ Not_found → do { memo.(key) := [ element :: bucket ]; element } ]
;
```

Instead of *share* we could have used the name *recall*, or *memory*, since either we recall a previously archived equal element, or else this element is archived for future recall. It is an

associative memory implemented with a hash-code. But the hash function is external to the memory, it is given as a key with each item .

It is an interesting property of this modular design that sharing and archiving are abstracted as a common notion.

Algorithm. A recursive structure of type *domain* is *fully shared* if any two distinct subelements have different values. If such a structure is traversed in a bottom-up way with systematic memoisation by *share*, replacing systematically an element by its memoised equal if possible, then it is reconstructed with full sharing. This only assumes that two equal elements have the same key.

```
end
;
```

5.2 Compressing tries

We may for instance instantiate *Share* on the algebra of tries, with a size *hash_max* depending on the application.

Module Mini

```
value hash_max = 9689 (* Mersenne 21 *)
;
module Dag = Share.Share (struct type domain = Trie.trie;
                                 value size = hash_max; end)
;
```

And now we compress a *trie* into a minimal dag using *share* by a simple bottom-up traversal, where the key is computed along by hashing. For this we define a general bottom-up traversal function, which applies a parametric *lookup* function to every node and its associated key.

```
value hash0 = 1 (* linear hash-code parameters *)
and hash1 letter key sum = sum + letter × key
and hash b arcs = (abs (arcs + Gen.dirac b)) mod hash_max
```

Caution - *abs* is needed because possible integer overflow, since otherwise mod may return a negative result, leading to error out-of-bound array at execution.

```
;
value traverse lookup = travel
 where rec travel = fun
  [ Trie.Trie (b, arcs) →
     let f (tries, span) (n, t) =
         let (t0, k) = travel t in
```

```
        ([ (n, t0) :: tries ], hash1 n k span) in
    let (arcs0, span) = List.fold_left f ([], hash0) arcs in
    let key = hash b span in
    (lookup (Trie.Trie (b, List.rev arcs0)) key, key)
  ]
;
```

Now we make a dag from a trie by recognizing common subtries.

```
value compress = traverse Dag.share
;
value minimize trie = let (dag, _) = compress trie in dag
;
value reset = Dag.reset
;
```

5.3 Dagified lexicons

We now return to our problem of building a lexicon which shares common suffixes of words as well as common prefixes.

Module Dagify

For instance, we may dagify a *trie* value read on the standard input stream with *minimize*, and write the resulting dag on standard output by calling *dagify*(), with:

```
value rec dagify () =
  let lexicon = (input_value stdin : Trie.trie) in
  let dag = Mini.minimize lexicon in
  output_value stdout dag
;
```

Module Make_english_lexicon

English words, using ASCII encoding

```
open Make_lex
;
module Make_lexicon = Make_lex Ascii
;
Make_lexicon.trie_of_strings ()
;
```

5.4 Some statistics

If we apply this technique to our English lexicon, with command:
`dagify <english.rem >small.rem`, we now get an optimal representation which only needs 1Mb of storage, half of the original ASCII string representation.

The recursive algorithms given so far are fairly straightforward. They are easy to debug, maintain and modify, due to the strong typing safeguard of ML, and even easy to formally certify. They are nonetheless efficient enough for production use, thanks to the optimizing native-code compiler of Objective Caml.

In our Sanskrit application, the trie of 11500 entries is shrunk from 219Kb to 103Kb in 0.1s, whereas the trie of 120000 flexed forms is shrunk from 1.63Mb to 140Kb in 0.5s on a 864MHz PC.

Our list of 173528 English words, represented as an ASCII file of 1.92 Mbytes, is represented as a trie of 4.5 Mbytes, which shrinks to 1.1 Mbytes by sharing (in 2.7s).

Measurements showed that the time complexity is linear with the size of the lexicon (within comparable sets of words). This is consistent with algorithmic analysis, since it is known that tries compress dictionaries up to a linear entropy factor, and that perfect hashing compresses trees in dags in linear time [12].

Tuning of the hash function parameters leads to many variations. For instance if we assume an infinite memory we may turn the hash calculation into a one-to-one Gödel numbering, and at the opposite end taking *hash_max* to 1 we would do list lookup in the unique bucket, with worse than quadratic performance.

Using hash tables for sharing with bottom-up traversal is a standard dynamic programming technique, but the usual way is to delegate computation of the hash function to some hash library, using a generic low-level package. This is what happens for instance if one uses the module hashtbl from the Ocaml library. Here the *Share* module does *not* compute the keys, which are computed on the client side, avoiding re-exploration of the structures. That is, *Share* is just an associative memory. Furthermore, key computation may take advantage of specific statistical distribution of the application domain.

We shall see later another application of the *Share* functor to the minimization of the state space of (acyclic) finite automata. Actually, what we just did is minimization of acyclic deterministic automata represented as lexical dags.

More sophisticated compression techniques are known, which may combine with array implementations insuring fast access, and which may extend to possibly cyclic automata state spaces. Such techniques are used in lexical analysers for programming languages, for which speed is essential. See for instance the table-compression method described in section 3.9 of [1].

5.5 ISO-LATIN and French

The next modules explain how to define the ISO-LATIN encoding, and how to use it to represent French words.

First we give a simple lexer, which is used to parse raw text with Camlp4 grammars. Next we give such a grammar, used to define a transducer from notations such as `e'` to ISO-LATIN character é. Finally, we give a module Latin which defines ISO-LATIN encoding.

Module Zen_lexer

A very simple lexer recognizing 1 character idents and integers and skipping spaces and comments between % and eol; used for various transduction tasks with Camlp4 Grammars.

```
open Camlp4.PreCast;
open Format;

 module Loc = Loc; (* Using the PreCast Loc *)
 module Error = Camlp4.Struct.EmptyError; (* Dummy Error module *)
 module Token = struct
   module Loc = Loc
   ;
   type t =
     [ KEYWORD of string
     | LETTER of string
     | INT of int
     | EOI
     ]
   ;
   module Error = Error
   ;
   module Filter = struct
     type token_filter = Camlp4.Sig.stream_filter t Loc.t
     ;
     type t = string → bool
     ;
     value mk is_kwd = is_kwd
     ;
     value rec filter is_kwd = parser
       [ [: `((KEYWORD s, loc) as p); strm :] →
             if is_kwd s then [: `p; filter is_kwd strm :]
             else failwith ("Undefined token: " ^ s)
```

```
        | [: `x; s :] → [: `x; filter is_kwd s :]
        | [: :] → [: :]
        ]
      ;
      value define_filter _ _ = ()
      ;
      value keyword_added _ _ _ = ()
      ;
      value keyword_removed _ _ = ()
      ;
    end
    ;
    value to_string = fun
      [ KEYWORD s → sprintf "KEYWORD␣%S" s
      | LETTER s → sprintf "LETTER␣%S" s
      | INT i → sprintf "INT␣%d" i
      | EOI → "EOI"
      ]
    ;
    value print ppf x = pp_print_string ppf (to_string x)
    ;
    value match_keyword kwd = fun
      [ KEYWORD kwd' when kwd' = kwd → True
      | _ → False
      ]
    ;
    value extract_string = fun
      [ INT i → string_of_int i
      | LETTER s | KEYWORD s → s
      | EOI → ""
      ]
    ;
  end
  ;
  open Token
  ;
```

The string buffering machinery.

```
value store buf c = do { Buffer.add_char buf c; buf }
;
```

```
value rec number buf =
  parser
  [ [: `('0'..'9' as c); s :] → number (store buf c) s
  | [: :] → Buffer.contents buf
  ]
;
value rec skip_to_eol =
  parser
  [ [: `'\n' | '\026' | '\012'; s :] → ()
  | [: `c ; s :] → skip_to_eol s
  ]
;
value next_token_fun =
  let rec next_token =
    parser _bp
    [ [: `'%' ; _ = skip_to_eol; s :] → next_token s
    | [: `('a'..'z' | 'A'..'Z' | '\192'..'\246' | '\248'..'\255' (* | '_' *)
           as c) :] → LETTER (String.make 1 c)
    | [: `('0'..'9' as c); s = number (store (Buffer.create 80) c) :] →
          INT (int_of_string s)
    | [: `c :] _ep → KEYWORD (String.make 1 c)
    ] in
  let rec next_token_loc =
    parser bp
    [ [: `' ' | '\n' | '\r' | '\t' | '\026' | '\012'; s :] → next_token_loc s
    | [: tok = next_token :] ep → (tok, (bp, ep))
    | [: _ = Stream.empty :] → (EOI, (bp, succ bp))
    ] in
  next_token_loc
;
value mk () =
  fun init_loc cstrm → Stream.from
    (fun _ →
      let (tok, (bp, ep)) = next_token_fun cstrm in
      let loc = Loc.move `start bp (Loc.move `stop ep init_loc) in
      Some (tok, loc))
;
```

Module Transducer

```
module Gram = MakeGram Zen_lexer;

open Zen_lexer.Token;

value transducer trad t =
  try Gram.parse_string trad Loc.ghost t
  with
  [ Loc.Exc_located loc e → do
      { print_string "\n\n%!"
      ; Format.eprintf "In string \"%s\", at location %a:@."
                            t Loc.print loc
      ; raise e
      }
  ]
;
```

French with accents.

```
value french = Gram.Entry.mk "french encoding"
;
value french_word = Gram.Entry.mk "french word encoding"
;
EXTEND Gram (* french to code *)
  french :
    [ [ LETTER "a"; "^" → "â"
      | LETTER "a"; "‘" → "à"
      | LETTER "a" → "a"
      | LETTER "e"; "’" → "é"
      | LETTER "e"; "‘" → "è"
      | LETTER "e"; "^" → "ê"
      | LETTER "e"; "\"" → "ë"
      | LETTER "e" → "e"
      | LETTER "i"; "^" → "î"
      | LETTER "i"; "\"" → "ï"
      | LETTER "i" → "i"
      | LETTER "o"; "^" → "ô"
      | LETTER "o" → "o"
      | LETTER "u"; "‘" → "ù"
      | LETTER "u"; "^" → "û"
      | LETTER "u"; "\"" → "ü"
      | LETTER "y"; "\"" → "y\"" (* oulipo *)
```

```
        | LETTER "u" → "u"
        | LETTER "c"; "/" → "ç"
        | LETTER "c" → "c"
        | "-" → "-"
        | "." → "."
        | "'" → "'" (* aujourd'hui *)
        | l = LETTER → l
        ] ];
  french_word :
    [ [ w = LIST0 french; `EOI → String.concat "" w ] ];
END
;
value latin_of_string = transducer french_word
;
```

Module Latin

The ISO-latin encoding scheme
encode : *string* → *word*
decode : *word* → *string*

```
value encode string =
  let iso = Transducer.latin_of_string string in
  List.map int_of_char (List2.explode iso)
and decode word = List2.implode (List.map char_of_int word);
```

5.6 Statistics for French

We may now instanciate the functor *make_lex* with the Latin module.

Module Make_french_lexicon

French words, using Latin encoding.

```
module Make_lexicon = Make_lex Latin
;
Make_lexicon.trie_of_strings ()
;
```

A list of 138257 French words, represented as an 8-bit ASCII file of 1.52 Mbytes, is represented as a trie of 2.4 Mbytes, which shrinks to 450 Kbytes by sharing.

5.7 Lexicon repositories using tries and decos

In a typical computational linguistics application, grammatical information (part of speech role, gender/number for substantives, valency and other subcategorization information for verbs, etc) may be stored as decoration of the lexicon of roots/stems. From such a decorated trie a morphological processor may compute the lexmap of all flexed forms, decorated with their derivation information encoded as an inverse map. This structure may itself be used by a tagging processor to construct the linear representation of a sentence decorated by feature structures. Such a representation will support further processing, such as computing syntactic and functional structures, typically as solutions of constraint satisfaction problems.

Let us for example give some information on the indexing structures `trie`, `deco` and `lexmap` used in our computational linguistics tools for Sanskrit.

The main component in our tools is a structured lexical database, described in [15, 16]. From this database, various documents may be produced mechanically, such as a printable dictionary through a TEX/Pdf compiling chain, and a Web site (`http://pauillac.inria.fr/~huet/SKT`) with indexing tools. The index CGI engine searches the words by navigating in a persistent trie index of stem entries. In the current version, the database comprises 12000 items. The corresponding trie (shared as a dag) has a size of 103KB.

When computing this index, another persistent structure is created. It records in a deco all the genders associated with a noun entry (nouns comprise substantives and adjectives, a blurred distinction in Sanskrit). At present, this deco records genders for 5700 nouns, and it has a size of 268KB.

A separate process may then iterate on this genders structure a grammatical engine, which for each stem and associated gender generates all the corresponding declined forms. Sanskrit has a specially prolific morphology, with 3 genders, 3 numbers and 7 cases. The grammar rules are encoded into 84 declension tables, and for each declension suffix an internal sandhi computation is effected to compute the final flexed form. All such words are recorded in a flexed forms lexmap, which stores for every word the list of pairs (stem,declension) which may produce it. This lexmap records about 120000 such flexed forms with associated grammatical information, and it has a size of 341KB (after minimization by sharing, which contracts approximately by a factor of 10). A companion trie, without the information, keeps the index of flexed words as a minimized structure of 140KB.

A similar process produces the conjugated forms of root verbs.

6 Variation: Ternary trees

Let us now try a variation on lexicon structure, using the notion of a ternary tree.

This notion is fairly natural if one wants to restore for ordered trees the locality of zipper navigation in binary trees. Remark that when we go up to the current father, we have to close the list of elder siblings in order to restore the full list of children of the upper node.

With ternary trees each tree node has two lists of children, elders and youngers. When we go up in the zipper structure, it is now a constant cost operation. Remark that this partition into elders and youngers is not intrinsic and carries no information, except the memory of the previous navigation. This is again an idea of optimizing computation by creating redundancy in the data structure representations. We may for instance exploit this redundancy in balancing our trees for faster access.

Ternary trees are inspired from Bentley and Sedgewick[3].

Module Tertree

Trees are ternary trees for use as two-ways tries with zippers. *Tree* (b, l, i, t, r) at occurrence u stores u as a word iff $b =$ *True*, and gives access to t at occurrence [u :: i] as a son, having l and r as respectively left stack of elder and right list of younger brothers; *Leaf True* at occurrence u stores u as a word with no descendants; *Leaf False* is only needed to translate *Trie.empty* = *Trie* (*False*, [ ]).

```
type tree =
  [ Tree of (bool × forest × int × tree × forest)
  | Leaf of bool
  ]
and forest = list (int × tree)
;
```

Invariant : integers are in increasing order in siblings, no repetition.
Simple translation of a trie as a tree.

```
value rec trie_to_tree = fun
  [ Trie.Trie (b, arcs) → match arcs with
      [ [] → Leaf b
      | [ (n, t) :: arcs ] → Tree (b, [], n, trie_to_tree t, List.map f arcs)
         where f (n, t) = (n, trie_to_tree t)
      ]
  ]
;
exception Anomaly
;
```

More sophisticated translation as a balanced tree.

```
value rec balanced = fun
  [ Trie.Trie (b, arcs) → match arcs with
      [ [] → Leaf b
      | _ → (* bal balances k first arcs of l and stacks them in acc *)
```

```
      let rec bal acc l k = (* assert | l |  ≥  k *)
          if k = 0 then (acc, l)
          else match l with
            [ [] → raise Anomaly (* impossible by assertion *)
            | [ (n, t) :: r ] → bal [ (n, balanced t) :: acc ] r (k − 1)
            ] in
      let (stack, rest) = let half = (List.length arcs)/2 in
                          bal [] arcs half in
      match rest with
        [ [] → raise Anomaly (* | rest | = | arcs | − half > 0 *)
        | [ (n, t) :: right ] →
            Tree (b, stack, n, balanced t, List.map f right)
            where f (n, t) = (n, balanced t)
        ]
    ]
  ]
;
type zipper =
  [ Top
  | Zip of (bool × forest × int × forest × zipper)
  ]
;
zip_up : tree → zipper → tree
value rec zip_up t = fun
  [ Top → t
  | Zip (b, left, n, right, up) → zip_up (Tree (b, left, n, t, right)) up
  ]
;
```

tree_of c builds the filiform *tree* containing c.

```
tree_of : word → trie
value rec tree_of = fun
  [ [] → Leaf False
  | [ n ] → Tree (False, [], n, Leaf True, [])
  | [ n :: rest ] → Tree (False, [], n, tree_of rest, [])
  ]
;
mem_tree : word → tree → bool
```

```
value rec mem_tree c = fun
  [ Tree (b, l, n, t, r) → match c with
    [ [] → b
    | [ i :: s ] →
      let rec memrec l n t r =
          if i = n then mem_tree s t
          else if i < n then match l with
            [ [] → False
            | [ (m, u) :: l′] → memrec l′ m u [ (n, t) :: r ]
            ]
          else match r with
            [ [] → False
            | [ (m, u) :: r′ ] → memrec [ (n, t) :: l] m u r′
            ] in
      memrec l n t r
    ]
  | Leaf b → b ∧ c = []
  ]
;
```

We assume that *enter* used over tries, and that *trees* are not updated.
Translates trie in *entries_file* into corresponding tree.

```
value translate_entries entries_file result_file =
  let entries_trie = (Gen.gobble entries_file : Trie.trie) in
  Gen.dump (balanced entries_trie) result_file
;
```

Module Minitertree

Similarly to *Mini* for tries, we may dagify ternary trees.

```
value hash_max = 9689 (* Mersenne 21 *)
;
module Dag = Share.Share (struct type domain = Tertree.tree;
                                 value size = hash_max; end)
;
value hash0 = 1 (* linear hash-code parameters *)
and hash1 letter key sum = sum + letter × key
and hash b arcsl k n arcsr = (arcsl + arcsr + n × k + Gen.dirac b) mod hash_max;
```

```
value leaff = Tertree.Leaf False
and leaft = Tertree.Leaf True
;
value traverse lookup = travel
 where rec travel = fun
  [ Tertree.Tree(b,fl,n,t,fr) →
     let f (trees,span) (n,t) =
         let (t0,k) = travel t in
         ([ (n,t0) :: trees ],hash1 n k span) in
     let (arcsl,spanl) = List.fold_left f ([],hash0) fl
     and (t1,k1) = travel t
     and (arcsr,spanr) = List.fold_left f ([],hash0) fr in
     let key = hash b spanl k1 n spanr in
     (lookup (Tertree.Tree(b,List.rev arcsl,n,t1,List.rev arcsr)) key, key)
  | Tertree.Leaf b → if b then (leaft,1) else (leaff,0)
  ]
;
```

Now we make a dag from a trie by recognizing common subtries.

```
value compress = traverse Dag.share
;
value minimize tree = let (dag,_) = compress tree in dag
;
value rec dagify_tree () =
  let lexicon = (input_value stdin : Tertree.tree) in
  let dag = minimize lexicon in
  output_value stdout dag
;
value reset = Dag.reset
;
```

Ternary trees are more complex than tries, but use slightly less storage. Access is potentially faster in balanced trees than tries. A good methodology seems to use tries for edition, and to translate them to balanced ternary trees for production use with a fixed lexicon.

The ternary version of our English lexicon takes 3.6Mb, a savings of 20% over its trie version using 4.5Mb. After dag minimization, it takes 1Mb, a savings of 10% over the trie dag version using 1.1Mb. In the case of our Sanskrit lexicon index, the trie takes 221Kb and the tertree 180Kb, whereas shared as dags the trie takes 103Kb and the tertree 96Kb.

7 Decorated Tries for Flexed Forms Storage

7.1 Decorated Tries

A set of elements of some type τ may be identified as its characteristic predicate in $\tau \rightarrow bool$. A trie with boolean information may similarly be generalized to a structure representing a map, or function from words to some target type, by storing elements of that type in the information slot.

In order to distinguish absence of information, we could use a type *(option info)* with constructor *None*, presence of value *v* being indicated by *Some(v)*. We rather choose here a variant with lists, which are versatile to represent sets, feature structures, etc. Now we may associate to a word a non-empty list of information of polymorphic type α, absence of information being encoded by the empty list. We shall call such associations a *decorated trie*, or *deco* in short.

Module Deco

Same as *Trie*, except that info carries a list. A *deco* associates to a *word* a non-empty list of attributes.
Tries storing decorated words.

```
type deco α = [ Deco of (list α × darcs α) ]
and darcs α = list (Word.letter × deco α)
;
```

Invariant: integers are in increasing order in darcs, no repetition.
The zipper type is adapted in the obvious way, and algorithm *zip_up* is unchanged.

```
type zipd α =
  [ Top
  | Zip of ((list α) × (darcs α) × Word.letter × (darcs α) × (zipd α))
  ]
;
```

```
zip_up : (zipd α) → (deco α) → (deco α)
```

```
value rec zip_up z t = match z with
  [ Top → t
  | Zip (i, left, n, right, up) →
        zip_up up (Deco (i, List2.unstack left [ (n, t) :: right ]))
  ]
;
```

Function *trie_of* becomes *deco_of*, taking as extra argument the information associated with the singleton trie it constructs.
deco_of i w builds the filiform *deco* containing *w* with info *i*.
deco_of : (*list* α) → *word* → (*deco* α)

```
value deco_of i = decrec
  where rec decrec = fun
    [ [] → Deco (i, [])
    | [ n :: rest ] → Deco ([], [ (n, decrec rest) ])
    ]
;
```

Note how the empty list [] codes absence of information. We generalize algorithm *enter* into *add*, which unions new information to previous one:
add : (*deco* α) → *word* → (*list* α) → (*deco* α)

```
value add deco word i = enter_edit Top deco word
  where rec enter_edit z d = fun
     [ [] → match d with [ Deco (j, l) → zip_up z (Deco (List2.union i j, l)) ]
     | [ n :: rest ] → match d with
          [ Deco (j, l) → let (left, right) = List2.zip n l in
                              match right with
            [ [] → zip_up (Zip (j, left, n, [], z)) (deco_of i rest)
            | [ (m, u) :: r ] →
               if m = n then enter_edit (Zip (j, left, n, r, z)) u rest
               else zip_up (Zip (j, left, n, right, z)) (deco_of i rest)
            ]
          ]
     ]
;
value empty = Deco ([], [])
;
```

Invariant: *contents* returns words in lexicographic order.
contents : *deco* α → *list* (*word* × *list* α)

```
value contents t = contents_prefix [] t
  where rec contents_prefix pref = fun
    [ Deco (i, l) →
       let down = let f l (n, t) = l @ (contents_prefix [ n :: pref ] t) in
                  List.fold_left f [] l in
       if i = [] then down else [ (List.rev pref, i) :: down ]
    ]
;
```

```
iter : (word → α → unit) → (deco α) → unit
value iter f t = iter_prefix [] t
  where rec iter_prefix pref = fun
    [ Deco (i, l) → do
        { List.iter (f (List.rev pref)) i (* no action if i = [] *)
        ; let phi (n, u) = iter_prefix [ n :: pref ] u in
          List.iter phi l
        }
    ]
;
fold : (α → word → (list β) → α) → α → (deco β) → α
value fold f x t = iter_prefix [] x t
  where rec iter_prefix pref x = fun
    [ Deco (i, l) →
      let accu = if i = [] then x else (f x (List.rev pref) i)
      and g x (n, t) = iter_prefix [ n :: pref ] x t in
      List.fold_left g accu l
    ]
;
assoc : word → (deco α) → (list α)
value rec assoc c = fun
  [ Deco (i, arcs) → match c with
    [ [] → i
    | [ n :: r ] → try assoc r (List.assoc n arcs)
                   with [ Not_found → [] ]
    ]
  ]
;
```

next t returns the first element of *deco t* with non-empty info.

```
value next t = next_rec [] t
  where rec next_rec pref = fun
    [ Deco (i, arcs) →
      if i = [] then match arcs with
          [ [] → raise (Failure "next_deco")
          | [ (n, u) :: _ ] → next_rec [ n :: pref ] u
          ]
      else List.rev pref
    ]
```

```
;
```

last t returns the last element of *deco t*.

```
value last t = last_rec [] t
  where rec last_rec acc = fun
    [ Deco (i, l) → match l with
      [ [] → List.rev acc
      | _ → let (m, u) = List2.last l in
              last_rec [ m :: acc ] u
      ]
    ]
;
```

Now the forgetful functor: $forget_deco : (deco\ \alpha) \rightarrow trie$

```
value rec forget_deco = fun
  [ Deco (i, l) →
     Trie.Trie (List2.non_empty i, List.map (fun (n, t) → (n, forget_deco t)) l)
  ]
;
```

7.2 Lexical maps

We can easily generalize sharing to decorated tries. However, substantial savings will result only if the information at a given node is a function of the subtrie at that node, i.e. if such information is defined as a *trie morphism*. This will not be generally the case, since this information is in general a function of the word stored at that point, and thus of all the accessing path to that node. The way in which the information is encoded is of course crucial. For instance, encoding morphological derivation as an operation on the suffix of a flexed form is likely to be amenable to sharing common suffixes in the flexed trie, whereas encoding it as an operation on the whole stem will prevent any such sharing.

In order to facilitate the sharing of mappings which preserve an initial prefix of a word, we shall use the notion of *differential word* above.

We may now store inverse maps of lexical relations (such as morphology derivations) using the following structures (where the type parameter α: codes the relation).

Module Lexmap

A specialisation of Deco, with info localised to the current word.

```
type inverse α = (Word.delta × α)
and inv_map α = list (inverse α)
;
```

Such inverse relations may be used as decorations of special lexical trees called lexical maps.

```
open Deco;
```

```
type lexmap α = deco (inverse α)
;
```

Typically, if word w is stored in a *lexmap* at a node whose decoration carries (d, r), this represents the fact that w is the image by relation r of $w' = patch\ d\ w$. Such a *lexmap* is thus a representation of the image by r of a source lexicon. This representation is invertible, while preserving maximally the sharing of prefixes, and thus being amenable to sharing.
Here α is *list morphs*. When word w has info [... $(delta, l)$ ...] with $delta = diff\ w\ w'$ it tells that $R\ w'\ w$ for every morph relation R in l where $w' = patch\ delta\ w$.

```
value single (d, i) = (d, [ i ])
;
```

add_inv : (*inverse* α) → (*inv_map* (*list* α)) → (*inv_map* (*list* α))

```
value rec add_inv ((delta, flex) as i) = fun
  [ [] → [ single i ]
  | [ (d, lflex) :: l ] as infos →
      if d = delta then [ (d, List2.union1 flex lflex) :: l ]
      else if Word.less_diff d delta then [ (d, lflex) :: add_inv i l ]
                                     else [ single i :: infos ]
  ]
;
```

add_inv2 : (*inverse* α) → (*inv_map* (*list* α)) → (*inv_map* (*list* α))
Similar to *add_inv* but raises the exception *List2.Twice_the_same_value* when trying to add twice the same decoration for a same word with the same *delta*.

```
value rec add_inv2 ((delta, flex) as i) = fun
    [ [] → [ single i ]
    | [ (d, lflex) :: l ] as infos →
        if d = delta then [ (d, List2.union2 flex lflex) :: l ]
        else if Word.less_diff d delta then [ (d, lflex) :: add_inv i l ]
        else [ single i :: infos ]
    ]
;
```

addl : (*lexmap* (*list* α)) → *word* → (*inverse* α) → (*lexmap* (*list* α))

```
value addl lexmap word i = enter_edit Top lexmap word
  where rec enter_edit z d = fun
  [ [] → match d with [ Deco (j, l) → zip_up z (Deco (add_inv i j, l)) ]
  | [ n :: rest ] →
      match d with
      [ Deco (j, l) →
        let (left, right) = List2.zip n l in
        match right with
        [ [] → zip_up (Zip (j, left, n, [], z)) (deco_of [ single i ] rest)
        | [ (m, u) :: r ] →
          if m = n then enter_edit (Zip (j, left, n, r, z)) u rest
          else zip_up (Zip (j, left, n, right, z)) (deco_of [ single i ] rest)
        ]
      ]
  ]
;
exception Duplication
;
addl2 : (lexmap (list α)) → word → (inverse α) → (lexmap (list α))
value addl2 lexmap word i =
  try enter_edit Top lexmap word
      where rec enter_edit z d = fun
      [ [] → match d with [ Deco (j, l) → zip_up z (Deco (add_inv2 i j, l)) ]
      | [ n :: rest ] →
          match d with
          [ Deco (j, l) →
            let (left, right) = List2.zip n l in
            match right with
            [ [] → zip_up (Zip (j, left, n, [], z)) (deco_of [ single i ] rest)
            | [ (m, u) :: r ] →
              if m = n then enter_edit (Zip (j, left, n, r, z)) u rest
              else zip_up (Zip (j, left, n, right, z)) (deco_of [ single i ] rest)
            ]
          ]
      ]
  with [ List2.Twice_the_same_value → raise Duplication ]
;
```

7.3 Minimizing lexical maps

We may now profit of the local structure of lexical maps to share them optimally as dags.

Interface for module Minimap

Minimization of Lexical Maps.

```
module Minimap : functor (Map :sig type inflected; end)
→ sig type inflected_map = Lexmap.lexmap (list Map.inflected);
        value minimize : inflected_map → inflected_map;
        value reset : unit → unit; end;
```

Module Minimap

```
module Minimap (Map :sig type inflected; end) = struct
```

Minimization of lexmaps of inflected forms as dags by bottom-up hashing.

```
type inflected_map = Lexmap.lexmap (list Map.inflected);
```

Use Mersenne 21 = 9689 for small dictionaries.

```
value hash_max = 216091 (* Mersenne 31 *)
;
module Inflected = struct type domain = inflected_map; value size = hash_max; end
;
module Memo = Share.Share Inflected
;
```

Bottom-up traversal with lookup computing a $key < hash_max$.

```
value hash0 = 0
and hash1 letter key sum = sum + letter × key
and hash i arcs = (abs (arcs + List.length i)) mod hash_max
;
value traverse_map lookup = travel
 where rec travel = fun
  [ Deco.Deco (i, arcs) →
     let f (tries, span) (n, t) =
         let (t0, k) = travel t in
         ([ (n, t0) :: tries ], hash1 n k span) in
     let (arcs0, span) = List.fold_left f ([], hash0) arcs in
     let key = hash i span in
```

```
    (lookup (Deco.Deco (i, List.rev arcs0)) key, key)
  ]
;
```

Make a dag of *flexed_map* by recognizing common substructures.

```
value compress_map = traverse_map Memo.share
;
value minimize map = let (dag, _) = compress_map map in dag
;
value reset = Memo.reset
;
end;
```

8 Finite State Machines as Lexicon Morphisms

8.1 Finite-state lore

Computational phonetics and morphology is one of the main applications of finite state methods: regular expressions, rational languages, finite-state automata and transducers, rational relations have been the topic of systematic investigations [27, 37], and have been used widely in speech recognition and natural language processing applications. These methods usually combine logical structures such as rewrite rules with statistical ones such as weighted automata derived from hidden Markov chains analysis in corpuses. In morphology, the pioneering work of Koskenniemi [24] was put in a systematic framework of rational relations and transducers by the work of Kaplan and Kay [21] which is the basis for the Xerox morphology toolset [22, 23, 2]. In such approaches, lexical data bases and phonetic and morphological transformations are systematically compiled in a low-level algebra of finite-state machines operators. Similar toolsets have been developed at University Paris VII, Bell Labs, Mitsubishi Labs, etc.

Compiling complex rewrite rules in rational transducers is however rather subtle. Some high-level operations are more easily expressed over deterministic automata, certain others are easier to state with ϵ-transitions, still others demand non-deterministic descriptions. Inter-traductions are well known, but tend to make the compiled systems bulky, since for instance removing non-determinism is an exponential operation in the worst case. Knowing when to compact and minimize the descriptions is a craft which is not widely disseminated, and thus there is a gap between theoretical descriptions, widely available, and operational technology, kept confidential.

Here we shall depart from this fine-grained methodology and propose more direct translations which preserve the structure of large modules such as the lexicon. The resulting algorithms will not have the full generality of the standard approach, and the ensuing

methodology may be thought by some as a backward development. Its justification lies in the greater efficiency of such direct translations, together with a simpler understanding of high-level operations which may be refined easily e.g. with statistical refinements, whereas the automata compiled by complex sequences of fine-grained operations are opaque blackboxes which are not easily amenable to heuristic refinements by human programming. Furthermore, the techniques are complementary, and it is envisioned that a future version of our toolset will offer both fine-grained and lexicon-based technologies.

The point of departure of our approach is the above remark that a lexicon represented as a lexical tree or trie is *directly* the state space representation of the (deterministic) finite state machine that recognizes its words, and that its minimization consists *exactly* in sharing the lexical tree as a dag. Thus we are in a case where the state graph of such finite languages recognizers is an acyclic structure. Such a pure data structure may be easily built without mutable references, and thus allocatable in the static part of the heap, which the garbage collector need not visit, an essential practical consideration. Furthermore, avoiding a costly reconstruction of the automaton from the lexicon data base is a computational advantage.

In the same spirit, we shall define automata which implement non-trivial rational relations (and their inversion) and whose state structure is nonetheless a more or less direct decoration of the lexicon trie. The crucial notion is that the state structure is a *lexicon morphism*.

8.2 Unglueing

We shall start with a toy problem which is the simplest case of juncture analysis, namely when there are no non-trivial juncture rules, and segmentation consists just in retrieving the words of a sentence glued together in one long string of characters (or phonemes). Let us consider an instance of the problem say in written English. You have a text file consisting of a sequence of words separated with blanks, and you have a lexicon complete for this text (for instance, 'spell' has been successfully applied). Now, suppose you make some editing mistake, which removes all spaces, and the task is to undo this operation to restore the original.

We shall show that the corresponding transducer may be defined as a simple navigation in the lexical tree state space, but now with a measure of non-determinism. Let us give the detailed construction of this unglueing automaton.

The transducer is defined as a functor, taking the lexicon trie structure as parameter.

Module Unglue

The unglueing problem is the simplest case of juncture analysis, namely when there are no non-trivial juncture rules, and segmentation consists just in retrieving the words of a sentence glued together in one long string of characters (or phonemes).

We shall show that the corresponding transducer may be defined as a simple navigation in the lexical tree state space, but now with a measure of non-determinism. The unglueing transducer is a lexicon morphism.

```
module Unglue (Lexicon : sig value lexicon : Trie.trie; end) = struct

type input = Word.word (* input sentence as a word *)
and output = list Word.word (* output is sequence of words *)
;
type backtrack = (input × output)
and resumption = list backtrack (* coroutine resumptions *)
;
```

Now we define our unglueing reactive engine as a recursive process which navigates directly on the (flexed) lexicon trie (typically the compressed trie resulting from the Dag module considered above). The reactive engine takes as arguments the (remaining) input, the (partially constructed) list of words returned as output, a backtrack stack whose items are (*input*, *output*) pairs, the path *occ* in the state graph stacking (the reverse of) the current common prefix of the candidate words, and finally the current *trie* node as its current state. When the state is accepting, we push it on the *backtrack* stack, because we want to favor possible longer words, and so we continue reading the input until either we exhaust the input, or the next input character is inconsistent with the lexicon data.

```
value rec react input output back occ = fun
  [ Trie.Trie (b, arcs) →
      let continue cont = match input with
        [ [] → backtrack cont
        | [ letter :: rest ] →
          let opt_state = try Some (List.assoc letter arcs) with
                              [ Not_found → None ] in
          match opt_state with
          [ Some s → react rest output cont [ letter :: occ ] s
          | None → backtrack cont
          ]
        ] in
      if b then
        let pushout = [ occ :: output ] in
        if input = [] then Some (pushout, back) (* solution found *)
        else let pushback = [ (input, pushout) :: back ] in
              (* we first try the longest possible matching word *)
              continue pushback
      else continue back
  ]
```

```
and backtrack = fun
  [ [] → None
  | [ (input, output) :: back ] → react input output back [] Lexicon.lexicon
  ]
;
```

Now, unglueing a sentence is just calling the reactive engine from the appropriate initial backtrack situation:

```
value unglue sentence = backtrack [ (sentence, []) ]
;

value print_out solution = List.iter pr (List.rev solution)
  where pr word = print_string (Ascii.decode (List.rev word) ^ " ")
;

resume : resumption → int → resumption

value resume cont n = match backtrack cont with
  [ Some (output, resumption) → do
      { print_string "\n Solution "
      ; print_int n
      ; print_string " : "
      ; print_out output
      ; Some resumption
      }
  | _ → None
]
;
value unglue_first sentence = (* similar to unglue *)
  resume [ (sentence, []) ] 1
;
value unglue_all sentence = restore [ (sentence, []) ] 1
  where rec restore cont n = match resume cont n with
  [ Some resumption → restore resumption (n + 1)
  | None → print_string (if n = 1 then " No solution found\n" else "\n")
  ]
;

end;
```

Module Unglue_test

The unglueing process is complete, relatively to the lexicon: if the input sentence may be obtained by glueing words from the lexicon, *unglue sentence* will return one possible solution. For instance, assuming the sentence is in French Childish Scatology:

```
module Childtalk = struct
value lexicon = Lexicon.make_lex ["boudin";"caca";"pipi"];
end
;
module Childish = Unglue Childtalk
;
```

Now, calling *Childish.unglue* on the encoding of the string `"pipicacaboudin"` produces a pair (*sol*, *cont*) where the reverse of *sol* is a list of words which, if they are themselves reversed and decoded, yields the expected sequence [`"pipi"`; `"caca"`; `"boudin"`].

```
match Childish.unglue (Ascii.encode "pipicacaboudin") with
  [ Some (sol, _) → Childish.print_out sol
  | None → failwith "Error"
  ]
;
```

We recover as expected: `pipi caca boudin`.
Another example, this time American street talk:

```
module Streettalk = struct
value lexicon = Lexicon.make_lex["a"; "brick"; "fuck"; "shit"; "truck"];
end
;
module Slang = Unglue Streettalk
;
```

```
match Slang.unglue (Ascii.encode "fuckatruckshitabrick") with
  [ Some (sol, cont) → Slang.print_out sol
  | None → failwith "Error"
  ]
;
```

We get as expected: `fuck a truck shit a brick`.
Of course there may be several solutions to the unglueing problem, and this is the rationale of the *cont* component, which is a *resumption*. For instance, in the previous example, *cont* is empty, indicating that the solution *sol* is unique.
We saw above that we could use the process *backtrack* in coroutine with the printer *print_out*

within the *unglue_all* enumerator.
Let us test this segmenter to solve an English charade (borrowed from "Palindroms and Anagrams", Howard W. Bergerson, Dover 1973).

```
module Short = struct
  value lexicon = Lexicon.make_lex
    ["able"; "am"; "amiable"; "get"; "her"; "i"; "to"; "together"]
  ;
  end
;
module Charade = Unglue Short
;
Charade.unglue_all (Ascii.encode "amiabletogether")
;
```

We get 4 solutions to the charade, printed as a quatrain polisson:

```
Solution 1 : amiable together
Solution 2 : amiable to get her
Solution 3 : am i able together
Solution 4 : am i able to get her
```

Unglueing is what is needed to segment a language like Chinese. Realistic segmenters for Chinese have actually been built using such finite-state lexicon driven methods, refined by stochastic weightings [38].

Several combinatorial problems map to variants of unglueing. For instance, over a one-letter alphabet, we get the Fröbenius problem of finding partitions of integers into given denominations (except that we get permutations since here the order of coins matters). Here is how to give the change in pennies, nickels and dimes:

```
value rec unary = fun [ 0 → "" | n → "|" ^ (unary (n − 1)) ]
;
```

The coins are the words of this arithmetic language:

```
value penny = unary 1 and nickel = unary 5 and dime = unary 10
;
module Coins = struct
  value lexicon = Lexicon.make_lex [penny; nickel; dime];
  end
;
module Frobenius = Unglue Coins
;
value change n = Frobenius.unglue_all (Ascii.encode (unary n))
```

;
change 17
;

This returns the 80 ways of changing 17 with our coins:

```
 Solution 1 : |||||||||| ||||| | |
...
 Solution 80 : | | | | | | | | | | | | | | | | |
```

Now we try phonemic segmentation in phonetic French.

```
module Phonetic = struct
value lexicon = Lexicon.make_lex ["gal";"aman";"de";"la";"rene";"ala";
"tour";"magn";"a";"nime";"galaman";"l";"arene";"magnanime"];
end
;
module Puzzle = Unglue Phonetic
;
Puzzle.unglue_all (Ascii.encode "galamandelarenealatourmagnanime")
;
```

Here we get 36 solutions, among which we find the two classic verses:

```
Solution 25 : gal aman de la rene ala tour magnanime
Solution 10 : galaman de l arene a la tour magn a nime
```

One last exemple, in Latin.

```
module Latin = struct
value lexicon = Lexicon.make_lex ["collectam";"ex";"ilio";"pubem";"exilio"];
end
;
module Virgil = Unglue Latin
;
Virgil.unglue_all (Ascii.encode "collectamexiliopubem")
;
```

Here the good solution is

```
 Solution 1 : collectam exilio pubem
```

(a people gathered for exile) and *not*

```
 Solution 2 : collectam ex ilio pubem
```

(a people gathered from Troy) as Donat interpreted Virgil's Aeneid, incurring the criticism of his fellow grammarian Servius (*Borrowed from Alberto Manguel*, *A History* of *Reading*.)

We remark that nondeterministic programming is basically trivial in a functional programming language, provided one identifies well the search space, states of computation are stored as pure data structures (which cannot get corrupted by pointer mutation), and fairness is taken care of by a termination argument (here this amounts to proving that `react` always terminate).

Nondeterminism is best handled by a generating process which delivers one solution at a time, and which thus may be used in coroutine fashion with a solution handler.

The reader will note that the very same state graph which was originally the state space of the deterministic lexicon lookup is used here for a possibly non-deterministic transduction. What changes is not the state space, but the way it is traversed. That is we clearly separate the notion of finite-state graph, a data structure, from the notion of a reactive process, which uses this graph as a component of its computation space, other components being the input and output tapes, possibly a backtrack stack, etc.

We shall continue to investigate transducers which are lexicon mappings, but now with an explicit non-determinism state component. Such components, whose structure may vary according to the particular construction, are decorations on the lexicon structure, which is seen as the basic deterministic state skeleton of all processes which are lexicon-driven; we shall say that such processes are *lexicon morphisms* whenever the decoration of a lexicon trie node is a function of the sub-trie at that node. This property entails an important efficiency consideration, since the sharing of the trie as a dag may be preserved when constructing the automaton structure:

Fact. Every lexicon morphism may minimize its state space isomorphically with the dag maximal sharing of the lexical tree. That is, we may directly decorate the lexicon dag, since in this case decorations are invariant by sub-tree sharing.

There are numerous practical applications of this general methodology. For instance, it is shown in [18] how to construct a Sanskrit segmenter as a decorated flexed forms lexicon, where the decorations express application of the euphony (sandhi) rules at the juncture between words. This construction is a direct extension of the unglueing construction, which is the special case when there are no euphony rules, or when they are optional.

Part II
Reactive Transducers

9 Introduction

This second part gives additional tools for manipulating variants of finite-state machines. They are a natural extension of the unglueing process presented at the end of Part I.

The general idea is to represent applicatively the state graph of finite-state machines as a decorated dictionary. The dictionary, used as spanning tree of the state transition graph, is a deterministic subset of this graph. The rest of the structure of the finite-state machine, permitting the representation of non-determinism, of loops, and of transducer operations, is encoded as attributes decorating the dictionary nodes. This general framework of *mixed automata* or *aums*, is described in reference [17]. Its application to the problem of segmentation and tagging of Sanskrit is described in [18].

We provide here various specific examples of this general methodology, and a mechanism for composing such finite-state descriptions in a modular fashion [19].

This methodology has been lifted more recently to a very general paradigm of relational programming within the framework of Eilenberg machines by Benoît Razet [20, 32, 33, 34].

10 A simplistic modular Automaton recognizer

The simplest aum structure is the one reduced to deterministic acyclic finite-state automata, where the aum structure is reduced to the underlying dictionary (Trie). Provided all states are accessible from the initial one, the reduced structure obtained by applying the Sharing functor yields the minimal deterministic automaton. This framework applies to the simple but important subcase of finite languages.

We assume known the modules of the first part of the toolkit documentation.

10.1 Simplistic aums

Interface for module Aum0

The auto structure for simplistic aums. This is a very simplified model with deterministic dictionaries; phase transition occurs at accepting states.

```
module Auto : sig
```

```
type auto = [ State of (bool × deter) ]
    (* bool is True for accepting states *)
and deter = list (Word.letter × auto)
    (* deter is the deterministic structure *)
;
end;
```

10.2 From automata to reactive processes

We consider finite state automata and transducers as data structures representing the states and their transitions. Such automata are interpreted by a computational process, which will manage the input tape, possibly an output tape in the case of transducers, and a backtrack stack used to manage non-deterministic search through fair backtracking. This is the point of view of finite-state control as presented in the reactive programming methodology, and thus we shall speak of this interpretative process as a *reactive engine*. In our first level of automata, this engine will be a simple recognizer for its input string: il will successfully terminate when this input string is a word belonging to the language, and raise the exception *Finished* otherwise.

Our methodology is *modular*, in the sense that it allows the composition of automata by layers - at the upper level, we consider a regular expression over a finite alphabet of *phases*, while at the lower level each phase corresponds to a finite automaton over letters of the global language. The global language corresponds to the substitution of each phase language in the given regular expression. The handling of control from the automaton of some phase into the automaton of the next phase is effected by the reactive engine through a scheduling switch implemented as a *Dispatch* module.

10.3 Dispatching

We compile our regular expressions using the Berry-Sethi method, which linearizes the expression, and computes the local automaton associated to this linearization [4, 5]. We call *phases* the morphology categories, defining the alphabet of the regular expression. The local automaton is described by an *initial* phase, a set of *terminal* phases, here represented as a boolean function over phases, and a *dispatch* transition function, mapping each phase to a set of following phases, sequentialized here as a list. In terms of Berry-Sethi [4], *initial* is called *1*, *dispatch* is called *follow*, and *terminal* is implicit from use of an endmarker symbol. In the terminology of Eilenberg [10], the phase language presented by Dispatch is a local set over phases.

The Dispatch module is generated by meta-programming from the regular expression, as we shall explain in section 12.

10.4 Scheduling and React

We are now ready to start the description of the reactive engine, as a functor *React* taking a module *Dispatch* as parameter, and using the *Dispatch.dispatch* function as a local scheduler. We assume the utility programming functions *fold_right*, *assoc*, *length*, *mem*, etc. from the *List* standard library.

Corresponding to simplistic aums *Aum0*, we have a simplified *React0* implementation.

Module React0

The reactive engine - an aum interpreter. Simplistic model with just deterministic transitions and no loops. Phase transition occurs at accepting states and jump to initial states of next phases.

```
open Aum0.Auto; (* auto State deter *)

module React
  (Disp : sig
          type phase;
          value transducer : phase → auto;
          value initial : phase;
          value terminal : phase → bool;
          value dispatch : phase → list phase;
          end) = struct

open Disp
;
type input = Word.word
;
```

Access stack in the deterministic part of the automaton:

```
type stack = list (Word.letter × auto)
;
type backtrack =
  [ Advance of phase and input ]
and resumption = list backtrack (* coroutine resumptions *)
;
```

The scheduler gets its phase transitions from dispatch.

```
value schedule phase input cont =
  let add phase cont = [ Advance phase input :: cont ] in
  List.fold_right add (dispatch phase) cont (* respects dispatch order *)
;
```

The reactive engine: *phase* is the parsing phase, *input* is the input tape represented as a *word*, *back* is the backtrack stack of type *resumption*.

```
value rec react phase input back = fun
  [ State (b, det) →
    let deter cont = match input with
      [ [] → continue cont
      | [ letter :: rest ] →
        let opt_state = try Some (List.assoc letter det) with
                            [ Not_found → None ] in
        match opt_state with
        [ Some s → react phase rest cont s
        | None → continue cont
        ]
      ] in
    if b then
      if input = [] then
        if terminal phase then Some back (* solution found *)
        else continue back
      else let cont = schedule phase input back in
           deter cont
    else deter back
  ]
and continue = fun
  [ [] → None
  | [ resume :: back ] → match resume with
      [ Advance phase input →
          react phase input back (transducer phase)
      ]
  ]
;
value init_react sentence = [ Advance initial sentence ]
;
value react1 sentence = continue (init_react sentence)
;
```

Computing multiplicities

```
value multiplicity sentence =
  count 0 (init_react sentence)
  where rec count n c =
    match continue c with
```

```
    [ Some next → count (n + 1) next
    | None → n
    ]
;
end;
```

The function *react1* is a recognizer for the rational language which is the image by the *transducer* morphism of the regular expression over phases. It stops at the first solution - when the input string is a word in this language - otherwise it raises the exception *Finished*. However, note that the general mechanism for managing non-determinism through coroutine resumptions allows restarting the computation to find other solutions. This mechanism will be specially important later when our engine is used for transductions, where we may be interested in the various solutions.

We give above an example of using *continue* as a coroutine by computing the *multiplicity* function, which counts the number of ways in which the input string may be solution to the regular expression. We remark that standard formal language theory deals with languages as *sets* of words, whereas here we formalize the finer notion of a *stream* (i.e. a potentially infinite list) of words recursively enumerating a *multiset* of words.

11 Modular aum transducers

So far our automata were mere deterministic recognizers for finite sets of words (although a dose of non-determinism arises from the choice, at any accepting node, between external transition to the next phase and continuing the local search, since the local language may contain a word and one of its proper prefixes). We now consider a more general framework where we handle loops in the transition relation, non-deterministic transitions, and output.

Interface for module Aumt

The auto structure: model with both External and Internal transitions.

```
module Auto : sig

type continuation = (Word.word × Word.word)
and transition =
  [ External of (Word.word × continuation)
  | Internal of (Word.word × Word.delta)
  ]
;
```

An internal transition $Internal(w, d)$ recognizes w on the input tape and jumps to the state relatively addressed by d. An external transition $External(w, c)$ recognizes w on the

input tape and executes the continuation c. A continuation (u, v) stores words u as output parameter and v as access parameter in the next phase transducer.

The role of the continuation (u, v) depends on the application at hand. Typically, in a segmentation application, u will indicate some suffix of a word of the current phase language (the maximal suffix of a word which may be affected by a phonetic change), while v will indicate the minimal prefix of a word in the next phase language which provokes the phonetic change. Thus, in the Sanskrit application, where segmentation does the analysis of euphony transformation, the transition *External*$(w, (u, v))$ will correspond to the sandhi rule $(rev\ u) \mid v \rightarrow w$.

```
type auto = [ State of (deter × choices) ]
and deter = list (Word.letter × auto)
    (* deter is the deterministic structure *)
and choices = list transition;

end
;
```

Note that in this model there is no acceptance boolean. Their role is played by the existence in the current choices of an *External* transition.

The acceptance condition for the full language is that, when the input string has been completely read, we are in a terminal phase, and the final *External* transition has a v component which is recognized as *final* - here we shall adopt the convention that v must be empty. Otherwise, when the input string is not empty, it must contain v as a prefix.

The next module will present the corresponding reactive engine, which will be used to realize possibly non-deterministic transductions.

Module Reactt

The reactive engine - an aum simulator used by transducers.
This model uses both External and Internal transitions.

```
open Aumt.Auto; (* auto State transition External Internal deter choices *)

module React
  (Disp : sig
        type phase;
        value transducer : phase → auto;
        value initial : phase;
        value terminal : phase → bool;
        value dispatch : phase → list phase;
        end) = struct
```

```
open Disp
;
type input = Word.word
and output = list (phase × Word.word)
;
```

Access stack in the deterministic part of the automaton.

```
type stack = list (Word.letter × auto)
;
type backtrack =
  [ Choose of phase and input and output and auto and stack and choices
  | Advance of phase and input and output and Word.word
  ]
and resumption = list backtrack (* coroutine resumptions *)
;
```

A few service routines.
advance : *int* → *word* → *word*
advance n [*a1*; ... *aN*] = [*ap*; ... *aN*] where $p = N - n$
precondition: $n \leq N = |\, w \,|$ exception: *Guard*.

```
exception Guard
;
value rec advance n w = if n = 0 then w else match w with
  [ [] → raise Guard
  | [ _ :: tl ] → advance (n − 1) tl
  ]
;
```

access : *phase* → *word* → (*auto* × *stack*)

```
value access phase = acc (transducer phase) []
    where rec acc state stack = fun
        [ [] → (state, stack)
        | [ c :: rest ] → match state with
          [ State (deter, _) →
              acc (List.assoc c deter) [ (c, state) :: stack ] rest
          ]
        ]
;
```

The scheduler gets its phase transitions from dispatch.

```
value schedule phase input output w cont =
  let add phase cont = [ Advance phase input output w :: cont ] in
  List.fold_right add (dispatch phase) cont (* respects dispatch order *)
;
value rec pop n state stack =
  if n = 0 then (state, stack)
  else match stack with
      [ [] → raise (Failure "Wrong␣Internal␣jump")
      | [ (_, st) :: rest ] → pop (n − 1) st rest
      ]
and push w state stack = match w with
      [ [] → (state, stack)
      | [ c :: rest ] → match state with
        [ State (deter, _) →
          push rest (List.assoc c deter) [ (c, state) :: stack ]
        ]
      ]
;
value jump (n, w) state stack =
  let (state0, stack0) = pop n state stack in
  push w state0 stack0
;
value extract stack (_, (u, _)) =
  List.fold_left unstack u stack
    where unstack acc (c, _) = [ c :: acc ]
;
value final v =
  v = [] (* or some end of input marker *)
;
```

The non deterministic reactive engine:
phase is the parsing phase,
input is the input tape represented as a *word*,
output is the current result of type *output*,
back is the backtrack stack of type *resumption*,
stack is the current reverse access path in the deterministic part and
state is the current state of type *auto*.

```
value rec react phase input output back stack state = match state with
  [ State (det, choices) →
    (* we try the deterministic space before the non deterministic one *)
```

```
    let cont = if choices = [] then back
               else [ Choose phase input output state stack choices :: back ] in
    match input with
      [ [] → continue cont
      | [ letter :: rest ] →
          let opt_state = try Some (List.assoc letter det) with
                              [ Not_found → None ] in
          match opt_state with
            [ None → continue cont
            | Some next_state →
                let next_stack = [ (letter, state) :: stack ] in
                react phase rest output cont next_stack next_state
            ]
      ]
  ]
and choose phase input output back state stack = fun
  [ [] → continue back
  | [ External((w, (u, v)) as rule) :: others ] →
      let cont = if others = [] then back
                 else [ Choose phase input output state stack others :: back ] in
      let opt_tape = try Some (advance (List.length w) input) with
                         [ Guard → None ] in
      match opt_tape with
        [ None → continue cont
        | Some tape →
            let out = [ (phase, extract stack rule) :: output ] in
            if tape = [] (* input finished *) then
                  if terminal phase ∧ final v then Some (out, cont)
                  else continue cont
            else continue (schedule phase tape out v cont)
        ]
  | [ Internal(w, delta) :: others ] →
      let cont = if others = [] then back
                 else [ Choose phase input output state stack others :: back ] in
      let opt_tape = try Some (advance (List.length w) input) with
                           [ Guard → None ] in
      match opt_tape with
        [ None → continue cont
        | Some tape →
            let (next_state, next_stack) = jump delta state stack in
```

```
                react phase tape output cont next_stack next_state
            ]
    ]
and continue = fun
    [ [] → None
    | [ resume :: back ] → match resume with
        [ Choose phase input output state stack choices →
            choose phase input output back state stack choices
        | Advance phase input output word →
            let opt_access = try Some (access phase word) with
                                [ Not_found → None ] in
            match opt_access with
                [ None → continue back
                | Some (next_state, next_stack) →
                    react phase input output back next_stack next_state
                ]
        ]
    ]
;
value init_react sentence = [ Advance initial sentence [] [] ]
;
value react1 sentence = continue (init_react sentence)
;
end;
```

Similarly to above, we could use *continue* to compute the stream of solutions in coroutine fashion. Such examples have already been given in Part I, for the unglueing process.

Remark that we may now understand unglueing as a special case of a reactive transducer. There is only one phase L, and the global regular expression is $L*$. The dictionary nodes *Trie(b,arcs)* play the role of *State(arcs,c)* where *c=[]* if *b=False* and *c=[External([],([],[]))]* if *b=True*.

Remark that when we have only *External* transitions, the engine simplifies, since the *stack* access path is not needed any more to interpret the internal jumps. However, its first component, i.e. the access *word*, may be necessary as an argument to the transducer output.

12 Macro-Generation of the *Dispatch* module

12.1 Introduction

The meta-programming of the *Dispatch* module is effected by the *Regular* module, which uses the Berry-Sethi algorithm to linearize the given regular expression, compute the follow

relation, and macro-generate the source of a specific dispatching module, seen as a user-specified plug-in to the generic toolkit.

This facility uses the excellent Camlp4 Caml preprocessor, both for input (yielding a parser for the regular expression minilanguage), and for output (macro-generating the abstract syntax of the resulting module, piped into the Pidgin ML pretty-printer).

Module Regular

12.2 Module Berry_Sethi

First we define a module *Berry_Sethi* implementing the Berry-Sethi algorithm for compiling regular expressions. We provide a data type for representing regular expressions, a type for standard local automata and a function *compile* that takes an initial state, a regular expression, and returns a local automaton representation.

```
module Berry_Sethi : sig
  type regexp α =
    [ One
    | Symb of α (* α is the type parameter of symbols *)
    | Union of regexp α and regexp α
    | Conc of regexp α and regexp α
    | Star of regexp α
    | Epsilon of regexp α
    | Plus of regexp α
    ]
  ;
  type marked α = (α × int)
  and local_auto α =
      ( marked α
      × list (marked α)
      × list (marked α × list (marked α))
      × list (marked α)
      )
  ;
  value compile : marked α → regexp α → local_auto α
;
end = struct
```

We describe regular expressions using a type abstracted from a basis alphabet α. This type provides extra constructors such as *Plus*, useful for practical applications

```
type regexp α =
  [ One
  | Symb of α
  | Union of regexp α and regexp α
  | Conc of regexp α and regexp α
  | Star of regexp α
  | Epsilon of regexp α
  | Plus of regexp α
  ]
;
```

One, *Symb*, *Union*, *Conc*, and *Star* are the classical operators. *Epsilon e* has the meaning of *Union One e* and *Plus e* the meaning of *Union e* (*Star e*), but treated as an atomic expression. In the following we shall abbreviate "regular expression" into regexp.
Berry-Sethi compilation applies to linear regexps to produce a local automaton. Let us recall that a linear regexp has the property that any symbol appears only once in it. This can be made by adding an integer to the symbol making the whole unique in the regexp, thus it changes a bit the structure of symbols for linear regexps. A local automaton is an automaton in which the guarded symbol of the transition corresponds to the target state of the transition. The types for marked symbols and local automata are:

```
type marked α = (α × int)
and local_auto α =
    ( marked α (* initial state *)
    × list (marked α) (* other states *)
    × list (marked α × list (marked α)) (* transitions *)
    × list (marked α) (* terminal states *)
    )
;
```

A local automata is structurally represented with four components. The first is the initial state, this shows that we have decided to produce standard local automata. The second is for the set of states of the automaton, excluding the initial one. The third is the set of transitions: a transition is a state and the list of states associated with the transition. The forth component is for the set of terminal states.
Now that we have given the data-structures for representing regexps let us give our algorithm to mark a regexp linear. To get the list of symbols from any regexp, reading left to right, we use the function:
symb_lr : *regexp α* → *list α*

```
value symb_lr e = symb [] e
  where rec symb accu = fun
  [ One → accu
```

```
  | Symb s → [s :: accu]
  | Union e1 e2 | Conc e1 e2 →
    symb (symb accu e2) e1
  | Star e | Epsilon e | Plus e → symb accu e
  ]
;
```

Having computed the resulting list from *symb_lr*, we want to append to each symbol a unique number, meaning the number of times we have encountered a symbol reading left to right the list, beginning to count from 1 except when a symbol is present only once in the list, in which case we count 0.

mark_list : *list* α → *list* (*marked* α)

```
value mark_list l = markr [] l
  where rec markr laccu = fun
  [ [] → []
  | [ x :: l ] →
    let count_x = count 1
                  where rec count sum = fun
                      [ [] → sum
                      | [y :: l] → count (if y = x then sum + 1 else sum) l
                      ] in
    if List.mem x laccu ∨ List.mem x l (* multiples *)
    then [ (x, count_x laccu) :: markr [x :: laccu] l ]
    else [ (x, 0) :: markr laccu l ]
  ]
;
```

The *mark_list* function has a *laccu* list that records symbols present at least twice.

Now we can define the *mark* function using a function *map_lr* that replaces in a regexp, from left to right, its symbols with a list of marked symbols (resulting from *mark_list*).

mark : *regexp* α → (*regexp* (*marked* α) × *list* (*marked* α))

```
value mark e =
  let rec map_lr li = fun
    [ One → (One, li)
    | Symb s →
        (Symb (List.hd li), List.tl li)
    | Union e1 e2 →
        let (e1_m, lj) = map_lr li e1 in
        let (e2_m, lk) = map_lr lj e2 in
        (Union e1_m e2_m, lk)
    | Conc e1 e2 →
```

```
        let (e1_m, lj) = map_lr li e1 in
        let (e2_m, lk) = map_lr lj e2 in
        (Conc e1_m e2_m, lk)
    | Star e1 →
        let (e1_m, lj) = map_lr li e1 in
        (Star e1_m, lj)
    | Epsilon e1 →
        let (e1_m, lj) = map_lr li e1 in
        (Epsilon e1_m, lj)
    | Plus e1 →
        let (e1_m, lj) = map_lr li e1 in
        (Plus e1_m, lj)
    ] in
  let symbols = symb_lr e in
  let symbols_m = mark_list symbols in
  let (e_m, li) = map_lr symbols_m e in
    (* note li must be [] *)
  (e_m, symbols_m)
;
```

The list of symbols mapped into the expression is *symbols_m* . Note that the first component of a marked symbol is the original one, so one can easily recover the original symbol from a marked one.
By now, we assume our regexps linear and we define the functions of Berry-Sethi compiling. It is very useful to get the information whether the empty string belongs to the regexp to compute the automaton efficiently. We thus present a new type *d_regexp* for discriminating regexps that generate or not the empty string. It is almost the same as type *regexp* but with an information encoding the discrimination, and this for all sub-constructions of the expression. It is represented with a boolean: *True* if the empty string is generated by the regexp, *False* otherwise.

```
type d_regexp α =
  [ DOne
  | DSymb of α
  | DUnion of bool and d_regexp α and d_regexp α
  | DConc of bool and d_regexp α and d_regexp α
  | DStar of d_regexp α
  | DEpsilon of d_regexp α
  | DPlus of bool and d_regexp α
  ]
;
```

DOne, *DSymb*, *DStar* and *DEpsilon* don't need this boolean because they have the information implicitly.
We simply extract this information from a regexp in a constant time analyzing the top node of a regexp:

```
delta : d_regexp α → bool

value delta = fun
  [ DOne → True
  | DSymb _ → False
  | DUnion b _ _ | DConc b _ _ → b
  | DStar _ | DEpsilon _ → True
  | DPlus b _ → b
  ]
;
```

The following algorithm transforms a regexp of type *regexp* into the discriminating one of type *d_regexp*.

```
discr : regexp α → d_regexp α

value rec discr = fun
  [ One → DOne
  | Symb s → DSymb s
  | Union e1 e2 →
      let de1 = discr e1
      and de2 = discr e2 in
      DUnion (delta de1 ∨ delta de2) de1 de2
  | Conc e1 e2 →
      let de1 = discr e1
      and de2 = discr e2 in
      DConc (delta de1 ∧ delta de2) de1 de2
  | Star e → DStar (discr e)
  | Epsilon e → DEpsilon (discr e)
  | Plus e →
      let de = discr e in
      DPlus (delta de) de
  ]
;
```

The cost of this computation is linear in the size of the regexp. Then we give an implementation of the *first* function, that gives the first symbols from a regexp, in linear time :

```
first : list α → d_regexp α → list α
```

```
value rec first l = fun
  [ DOne → l
  | DSymb d → [ d :: l ]
  | DUnion _ e1 e2 → first (first l e2) e1
  | DConc _ e1 e2 →
      let b1 = delta e1 in
      if b1 then first (first l e2) e1
      else first l e1
  | DStar e | DEpsilon e | DPlus _ e → first l e
  ]
;
```

The parameter l is for already computed first elements, a partial result.
A follow set is the list of directly accessible symbols from a given one in a regexp, it corresponds to the notion of continuation in the Berry-Sethi article. Now we have all the routines to present an implementation of the 'F' function from the Berry-Sethi article for computing the set of all follow sets.
follow : $\alpha \rightarrow$ *regexp* $\alpha \rightarrow$ *list* $(\alpha \times$ *list* $\alpha)$

```
value follow initial exp =
  let rec f1 exp l fol =
    match exp with
    [ DOne → fol
    | DSymb d → [ (d, l) :: fol ]
    | DUnion _ e1 e2 →
        let fol2 = f1 e2 l fol in
        f1 e1 l fol2
    | DConc _ e1 e2 →
        let fol2 = f1 e2 l fol in
        let l1 = if delta e2 then first l e2 else first [] e2 in
        f1 e1 l1 fol2
    | DStar e | DPlus _ e →
        let l_res = first l e in
        f2 e l_res fol
    | DEpsilon e → f1 e l fol ]
  and f2 exp l fol = (* (first [] exp) already in l *)
    match exp with
    [ DOne → fol
    | DSymb d → [ (d, l) :: fol ]
    | DUnion _ e1 e2 →
        let fol2 = f2 e2 l fol in
```

```
        f2 e1 l fol2
    | DConc _ e1 e2 →
        let b1 = delta e1
        and b2 = delta e2 in
        if b1 (* l1 and l2 in l *)
        then if b2
            then f2 e1 l (f2 e2 l fol)
            else f1 e1 (first [] e2) (f2 e2 l fol)
        else if b2
            then f2 e1 (first l e2) (f1 e2 l fol)
            else f1 e1 (first [] e2) (f1 e2 l fol)
    | DStar e | DEpsilon e | DPlus _ e → f2 e l fol
    ] in
  let fol_sets = f1 exp [] []
  and initials = first [] exp in
  [ (initial, initials) :: fol_sets ]
;
```

The initial state is a parameter of the function because it is not a state derived from symbols of the regexp. Its name must be chosen as a fresh symbol, not already occurring in the regexp. Because of our implementation of sets with lists we must guarantee not returning a list with duplicated elements. This can be done having the property that (*first exp*) is already in *l* or not. The computation is different in both cases, this is why we have *f1* and *f2*. We add a link between the initial state and the first states of the regexp.

The function *last* returns the last symbols from a regexp.

last : $\alpha \rightarrow$ *d_regexp* $\alpha \rightarrow$ *list* α

```
value last initial e =
  let rec last_rec exp l = match exp with
    [ DOne → l
    | DSymb d → [ d :: l ]
    | DUnion _ e1 e2 →
        let l2 = last_rec e2 l in
        last_rec e1 l2
    | DConc _ e1 e2 →
        let b2 = delta e2 in
        if b2 then last_rec e1 (last_rec e2 l)
        else last_rec e2 l
    | DStar e | DEpsilon e | DPlus _ e → last_rec e l
    ] in
  let l = last_rec e [] in
```

```
  if delta e then [ initial :: l ] else l
;
```

We add the initial state to the set of last states if the empty word belongs to the language. In terms of the note on "Local languages and the Berry-Sethi algorithm" by Jean Berstel and Jean-Eric Pin, we have presented algorithms to compute a linear regular expression into a local standard automaton. Adding *Plus e* directly as an operator in the abstract syntax is an optimisation because the equivalent *Conc e* (*Star e*) duplicates marked symbols and thus the number of states of the automaton.
We now present the *compile* function that computes the automaton from a regexp.
compile : *marked* α $\rightarrow$ *regexp* α $\rightarrow$ *local_auto* α

```
value compile initial exp =
  let (exp_m, states) = mark exp in
  let d_exp = discr exp_m in
  let fol = follow initial d_exp
  and lasts = last initial d_exp in
  (initial, states, fol, lasts)
;
```

```
end;
```

This is the end of our Berry-Sethi compilation algorithm. We ensure that we have presented an implementation that takes care of respecting the theoretical complexity, that is quadratic in the number of symbols in the regexp. The proof of the complexity is by induction on the regexp structure.

12.3 Module Regexp_system

We extend the way to define regexps with a regexp system that allows some degree of sharing, but no recursion.

```
module Regexp_system = struct
```

We use the structure of regular expressions defined in module *Berry_Sethi*.

```
open Berry_Sethi;
```

We mean by modularity the possibility of naming a regexp which may postmaster@inria.fr be used in another regexp as if it were a basic symbol. Now a *Symb* in a regexp can be a name or a symbol from an alphabet and we define a type *mix_symb* for describing this mix.

```
type name = string
and mix_symb α =
    [ Name of name
    | Alph of α
    ]
;
```

Then we define a *system* as a list of names and associated regexp:

```
type system α = list (name × regexp (mix_symb α))
;
```

We give an algorithm to transform a system into a simple regular expression.
flatten : *system* α → *regexp* α

```
value flatten sys =
  let rec flatten_regexp system l = fun
    [ One → (One, l)
    | Symb (Name s) →
      try (* we try to find s in already flattened regexp *)
        let e_flattened = List.assoc s l in
        (e_flattened, l)
      with [ Not_found →
               let rec extract_s = fun
                   [ [] → failwith "no␣extraction"
                   | [ (s2, e) :: sys ] → if s = s2 then (e, sys)
                                                    else extract_s sys
                   ] in
               (* knowing that dependencies must be in the rest of system *)
               let (e, new_sys) = extract_s system in
               let (e_flattened, new_l) = flatten_regexp new_sys l e in
               (e_flattened, [ (s, e_flattened) :: new_l]) ]
    | Symb (Alph s) → (Symb s, l)
    | Union e1 e2 →
        let (e1_f, l_left) = flatten_regexp system l e1 in
        let (e2_f, l_right) = flatten_regexp system l_left e2 in
        (Union e1_f e2_f, l_right)
    | Conc e1 e2 →
        let (e1_f, l_left) = flatten_regexp system l e1 in
        let (e2_f, l_right) = flatten_regexp system l_left e2 in
        (Conc e1_f e2_f, l_right)
    | Star e →
        let (e_f, new_l) = flatten_regexp system l e in
```

```
        (Star e_f, new_l)
    | Epsilon e →
        let (e_f, new_l) = flatten_regexp system l e in
        (Epsilon e_f, new_l)
    | Plus e →
        let (e_f, new_l) = flatten_regexp system l e in
        (Plus e_f, new_l)
    ] in
  let (e, system) = match sys with
    [ [] → failwith "empty␣system!!"
    | [ (_, e) :: system ] → (e, system)
    ] in
  let (e_f, _) = flatten_regexp system [] e in
  e_f
;
end;
```

The parameter l of *flatten_regexp* defines the list of regular expressions already flattened for a kind of lazy evaluation, it is initialized to the empty list. The parameter *system* represents the list of couple - name and regexp - not yet treated. The function *flatten_regexp*, providing a system and a list of expressions already flattened, replaces each symbol that is a name of regular expression by the associated one.

12.4 The concrete syntax for modular aums

A modular aum is a two-level structure: a regexp defined over an aum alphabet. And now we precise the concrete syntax for defining modular aums. It includes a name for the initial state, together with the name of the aum which recognizes the empty word, a basic alphabet of symbols represented as lower case strings between delimiters **alphabet** and **end** (corresponding to names of aums), then follows the definition of the regexp as a regexp system, between delimiters **automaton** and **end** together with a parameter for the name of the module implementing the state transitions.

```
module Id = struct
  value name = "Regular";
  value version = "2.3";
end;
```

```
module Regular (Ast : Camlp4.Sig.Camlp4Ast) = struct
  module Ast = Ast;
  open Ast;
  module Token = Camlp4.Struct.Token.Make Loc;
  module Lexer = Camlp4.Struct.Lexer.Make Token;
  open Camlp4.Sig;
  module Parser = struct
```

```
open Berry_Sethi;
open Regexp_system;
```

Using a standard lexer.

```
module Gram = Camlp4.Struct.Grammar.Static.Make Lexer;
```

We define the entry point of grammar *def_auto*:

```
value def_auto = Gram.Entry.mk "def_auto"
;
```

Here is the definition of the grammatical construction for a concrete system of regular expressions.

```
EXTEND Gram
  GLOBAL : def_auto;
  def_auto :
    [ [ "initial"; init = LIDENT; empty_aum = LIDENT;
        "alphabet"; aums = aum_names ; "end";
        "automaton"; module_name = UIDENT;
        system = rule_list; "end"; `EOI →
          (empty_aum, init, aums, module_name, system)
      ]
    ];
  aum_names :
    [ [ l = LIST1 aum_name SEP ";" → l ]
    ];
  aum_name :
    [ [ l = LIDENT → l ]
    ];
  rule_list :
    [ [ r = LIST1 rule SEP "in" → r ]
    ];
  rule :
    [ [ "node"; name = UIDENT; "="; e = expreg → (name, e) ]
    ];
```

```
  expreg :
    [ [ e1 = expreg; "|"; e2 = expreg → Union e1 e2 ]
    | [ e1 = expreg; "."; e2 = expreg → Conc e1 e2 ]
    | [ e = expreg; "*" → Star e
      | e = expreg; "?" → Epsilon e
      | e = expreg; "+" → Plus e ]
    | [ n = INT →
            match int_of_string n with
              [ 1 → One
              | _ → failwith "integer not autorized"
              ]
      | name = LIDENT → Symb (Alph name)
      | name = UIDENT → Symb (Name name)
      | "("; e = expreg ; ")" → e ]
    ];
END;
end;
```

N.B. Since we use a pre-defined lexer, one must take care of possible conflicts: “a*.b” would be interpreted with the floating-point times instead of suffix “*” followed by infix “.”.
The construction `automaton ... end` is parameterized by a string which will be the name of the module for the resulting automaton.

12.5 Example: Sanskrit morphology

We give a concrete example in the case of Sanskrit morphology. We have various phases for nominal forms:

- Noun for declined substantives
- Iic for beginnings of compounds
- Ifc for endings of compounds

similarly for verbal forms:

- Root for conjugated forms of roots
- Pv1 for preverb sequences
- Auxi for auxiliary verb forms

- Iiv for periphrastic prefixes

and finally for adverbs and particles:

- Unde undeclinable forms (infinitives, adverbs, etc.)
- Abso (absolutives)
- Pv2 for preverb sequences

Pv1 and Pv2 are two occurrences of the language of preverbs, issued from the linearization of the regular expression defining a Sanskrit sentence as a non-empty sequence of inflected word forms:

```
initial init epsilon_aum

alphabet
  noun ; root; unde; abso; iic; iiv; auxi; ifc; prev
end

automaton Disp
  node INVAR = prev.abso | unde in
  node CONJUG = prev? . root in
  node SUBST = iic* .noun | iic+ .ifc in
  node VERB = CONJUG | iiv.auxi in
  node PHRASE = (SUBST | VERB | INVAR)+
end
```

We remark that our language permits a succinct expression to what would be a complex regular expression if it had to be flattened.

12.6 Module Generate_ast

Let us define a module for generating the abstract syntax tree of a program implementing a phase automaton, assuming we provided a name for the aum of empty word, a list of aums used in the definition of the automaton, a name for the module associated to the automaton, the phase representing the first state of the automaton, the list of phases, the follow sets and the list of terminal states.
Dummy location needed for the quotation mechanism

```
value _loc = Loc.ghost
;
module Generate_ast : sig

  type aum_name = string
  and module_name = string
  and phase = (string × int)
  and follows = list (phase × list phase)
  and program = Ast.str_item;
  value gen_ast : aum_name → list aum_name → module_name →
        phase → list phase → follows →
        list phase → program;

  end = struct
```

The implementation consists in encapsulating the structures in the given datatypes and functions, with help of the macro-generating facilities of Camlp4. The reader is advised to skip this section at first reading.

```
  type aum_name = string
  and module_name = string
  and phase = (string × int)
  and follows = list (phase × list phase)
  and program = Ast.str_item
;
```

To append a number i to *name* capitalized, except for the number 0:

```
value convert_uid_int (name, i) =
  let cap = String.capitalize name in
  if i = 0 (* the name is used once in the regular expression *)
  then cap else cap ^ string_of_int i
;
```

Generates the type record *auto_vect*.

```
value gen_type_vect phases =
let f n acc = <:ctyp< $lid:n$ : Auto.auto; $acc$ >> in
let type_record = List.fold_right f phases <:ctyp<>> in
    <:str_item< type auto_vect = { $type_record$ } >>
;
```

Generates the type *phase*.

```
value gen_type_phase phases =
  (* first compute the names of all phases *)
  let list_type = List.map convert_uid_int phases in
  let sslt = List.fold_right (fun x acc → <: ctyp < $uid : x$ | $acc$ >>) list_type <:
ctyp <>> in
  <:str_item < type phase = [ $sslt$ ] >>
;
```

Generates the *transducer* function.

```
value gen_fun_morphism phases =
  let mc =
      let process (x, y) acc =
         <:match_case < $uid : x$ → Fsm.autos.$lid : y$ | $acc$ >> in
      List.fold_right process phases <: match_case <>> in
  <:str_item < value transducer = fun [ $mc$ ] >>
;
```

Generates the *dispatch* function.

```
value gen_fun_dispatch follows =
  (* translates a follow *)
  let trad_a_follow (n, ln) acc =
    let tln = List.fold_right tr ln <: expr < [] >>
        where tr x l =
           let x' = convert_uid_int x in
           <:expr < [ $uid : x'$ :: $l$ ] >> in
    <:match_case < $uid : convert_uid_int n$ → $tln$ | $acc$ >> in
  (* translates follow sets *)
  let match_cases = List.fold_right trad_a_follow follows <: match_case <>> in
  <:str_item < value dispatch = fun [ $match_cases$ ] >>
;
value gen_initial_state initial_phase =
  <:str_item < value initial = $uid : convert_uid_int initial_phase$ >>
;
```

Generation of the list of terminal states.

```
value gen_fun_terminal l =
  let the_list =
    List.fold_right tr l <: expr < [] >>
    where tr e l =
          let e' = <: expr < $uid : convert_uid_int e$ >> in
          <:expr < [ $ e'$ :: $l$ ] >> in
```

```
  <:str_item < value terminal phase = List.mem phase $the_list$ >>
;
```

Generates the module with name *module_name*

```
value gen_module empty_aum module_name initial_phase phases
                  follows terminal =
  (* declaration and definition of types and functions *)
  let type_phase = gen_type_phase [ initial_phase :: phases ]
  and fun_morphism = gen_fun_morphism
    [ (convert_uid_int initial_phase, empty_aum) ::
        (List.map (fun (x, y) → (convert_uid_int (x, y), x)) phases) ]
  and fun_dispatch = gen_fun_dispatch follows
  and value_initial = gen_initial_state initial_phase
  and fun_terminal = gen_fun_terminal terminal in
  let st = <: str_item < $type_phase$;
                          $fun_morphism$;
                          $fun_dispatch$;
                          $value_initial$;
                          $fun_terminal$ >> in
  (* end of decl and def *)
  <:str_item < module $module_name$ =
      functor ( Fsm : sig value autos : auto_vect; end ) → struct $st$ end>>
;
```

Generates all declarations of the file we want to generate

```
value gen_ast empty_aum aums module_name initial_phase phases
              follows terminal =
  let type_vect = gen_type_vect [ empty_aum :: aums ] in
  let module_body =
    gen_module empty_aum module_name initial_phase phases
               follows terminal in
  let module_contents = <: str_item < $type_vect$; $module_body$ >> in
  let automata_functor =
      <:str_item < module Automata =
        (* Automata is a functor with parameter module Auto *)
        functor (Auto : sig type auto; end) → struct $module_contents$ end >> in
  automata_functor
;
end;
```

12.7 Generating the plug-in module

open *Berry_Sethi*;
open *Parser*;

Reads on input channel *ch* the phase automaton description, parses it using the entry point *def_auto*, calls the Berry-Sethi algorithm, and returns the resulting plug-in module as Ocaml abstract syntax.

```
  value parse_implem ?directive_handler _loc strm =
    let (empty_aum, init, aums, module_name, system) =
        Gram.parse def_auto _loc strm in
    let exp = Regexp_system.flatten (List.rev system)
    and initial_phase = (init, 0) in
    let (initial_phase, phases, follows, terminal) = compile initial_phase exp in
    Generate_ast.gen_ast empty_aum aums module_name initial_phase phases follows terminal
  ;
  value parse_interf ?directive_handler _loc _strm = assert False
  ;
end
;
let module M = Camlp4.Register.OCamlParser Id Regular in ()
;
```

For generating the code from a concrete automaton in a file *xxx.aut* one may call:
camlp4 pr_r.cmo ./regular.cmo − impl xxx.aut
This will pretty-print the result to standard output, where it may be redirected to a file.

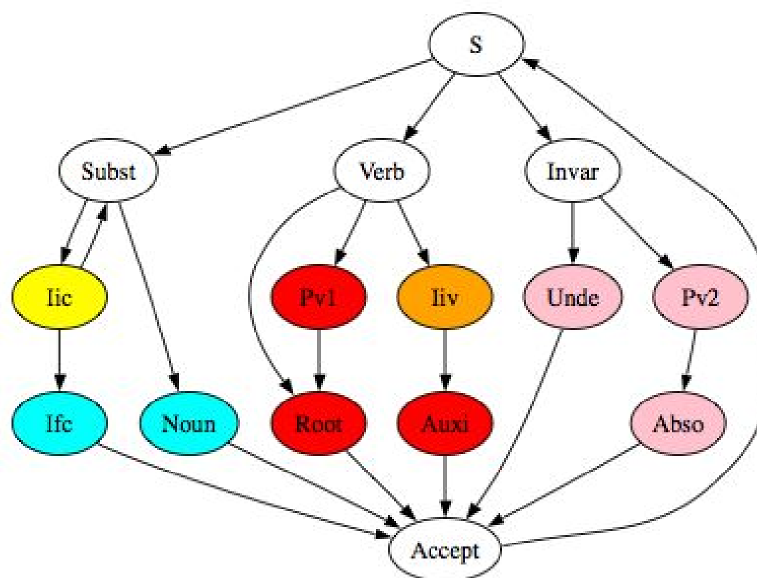
For instance, the modular aum described above for the Sanskrit example [sanskrit.aut] generates the following code :

```
module Automata (Auto : sig type auto = 'a; end) =
   struct
    type auto_vect =
      { epsilon_aum : Auto.auto;
        noun : Auto.auto;
        root : Auto.auto;
        unde : Auto.auto;
        abso : Auto.auto;
        iic : Auto.auto;
        iiv : Auto.auto;
        auxi : Auto.auto;
        ifc : Auto.auto;
```

```
      prev : Auto.auto }
  ;
  module Disp (Fsm : sig value autos : auto_vect; end) =
    struct
      type phase =
        [ Init
        | Iic1
        | Noun
        | Iic2
        | Ifc
        | Prev1
        | Root
        | Iiv
        | Auxi
        | Prev2
        | Abso
        | Unde ]
      ;
      value transducer =
        fun
        [ Init -> Fsm.autos.empty_aum
        | Iic1 -> Fsm.autos.iic
        | Noun -> Fsm.autos.noun
        | Iic2 -> Fsm.autos.iic
        | Ifc -> Fsm.autos.ifc
        | Prev1 -> Fsm.autos.prev
        | Root -> Fsm.autos.root
        | Iiv -> Fsm.autos.iiv
        | Auxi -> Fsm.autos.auxi
        | Prev2 -> Fsm.autos.prev
        | Abso -> Fsm.autos.abso
        | Unde -> Fsm.autos.unde ]
      ;
      value dispatch =
        fun
        [ Init -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
        | Iic1 -> [Iic1; Noun]
        | Noun -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
        | Iic2 -> [Iic2; Ifc]
        | Ifc -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
```

```
          | Prev1 -> [Root]
          | Root -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
          | Iiv -> [Auxi]
          | Auxi -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
          | Prev2 -> [Abso]
          | Abso -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
          | Unde -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde] ]
        ;
        value initial = Init;
        value terminal phase =
          List.mem phase [Noun; Ifc; Root; Auxi; Abso; Unde]
        ;
      end
    ;
  end
;
```

Here is the resulting local automaton (adding empty nodes S, Subst, Verb, Invar and Accept for better reading):



13 Producing the engine

We now have all the pieces to connect our dispatch plug-in to the generic reactive engine, parameterized by the automata vector provided by the user for recognizing the various

phases. Let us give a concrete example.

Given the *automata* functor corresponding to Sanskrit morphology in a *Sanskrit_dispatch* module, we show how to link it to the *Reactt* functor in order to produce a Sanskrit engine generator:

Module Sanskrit_engine

Engine *sanskrit_engine* using aumt structure with sanskrit.aut.

```
open Aumt; (* Auto *)
open Reactt; (* React *)
open Sanskrit_dispatch; (* Automata *)

module Automata_Aumt = Automata Auto
;
open Automata_Aumt; (* auto_vect Disp *)

module Gen_engine
 (Fsm : sig value autos : auto_vect; end) = struct
  module Phases = Disp Fsm
  ;
  open Phases (* phase, transducer, etc *)
  ;
  module Engine = React Phases
  ;
 end
;
```

Now we may provide the Sanskrit lexicons for the various lexical sorts as a vector
auto_vect = {*epsilon_aum* = *aum_0*; *noun* = *aum_noun*; ... *prev* = *aum_prev*}
in a module *Sanskrit_Aumt*.
We may then call the properly instanciated functor (*Gen_engine Sanskrit_Aumt*) in order to get e.g. *Engine.react1*.

What we just constructed is a simple engine which may recognize a Sanskrit sentence as a sequence of inflected word forms. Actually such forms are glued together using a euphony junction process known as *sandhi*. It is possible to invert the sandhi relation while doing the recognition, and to use the transducer output to give a trace of the sandhi relation between the words. Piping this process through a lemmatizer, which itselfs inverts the flexional morphology, yields a Sanskrit tagger. This application is described in [18].

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. “Compilers - Principles, Techniques and Tools.” Addison-Wesley, 1986.

[2] Kenneth R. Beesley and Lauri Karttunen. “Finite-State Morphology: Xerox Tools and Techniques.” Private communication, April 2001.

[3] Jon L. Bentley and Robert Sedgewick. “Fast Algorithms for Sorting and Searching Strings.” Proceedings, 8th Annual ACM-SIAM Symposium on Discrete Algorithms, Jan. 1997.

[4] Gérard Berry and Ravi Sethi. From regular expressions to deterministic automata. Theoretical Computer Science 48 (1986), pp. 117–126.

[5] Jean Berstel and Jean-Eric Pin. Local languages and the Berry-Sethi algorithm. Theoretical Computer Science 155 (1996), pp. 439–446.

[6] Eric Brill. “A simple rule-based part of speech tagger.” In Proceedings, Third Conference on Applied Natural Language Processing, 1992. Trento, Italy, 152–155.

[7] W. H. Burge. “Recursive Programming Techniques.” Addison-Wesley, 1975.

[8] Guy Cousineau and Michel Mauny. “The Functional Approach to Programming.” Cambridge University Press, 1998.

[9] Jan Daciuk, Stoyan Mihov, Bruce W. Watson and Richard E. Watson. “Incremental Construction of Minimal Acyclic Finite-State Automata.” Computational Linguistics 26,1 (2000).

[10] Samuel Eilenberg. Automata, Languages, and Machines, volume A. Academic Press, 1974.

[11] Matthias Felleisen and Daniel P. Friedman. “The Little MLer”. MIT Press, 1998.

[12] Philippe Flajolet, Paola Sipala and Jean-Marc Steyaert. “Analytic Variations on the Common Subexpresssion Problem.” Proceedings of 17th ICALP Colloquium, Warwick (1990), LNCS 443, Springer-Verlag, pp. 220–234.

[13] M. Gordon, R. Milner, C. Wadsworth. “A Metalanguage for Interactive Proof in LCF.” Internal Report CSR-16-77, Department of Computer Science, University of Edinburgh (Sept. 1977).

[14] Gérard Huet. “The Zipper”. J. Functional Programming 7,5 (Sept. 1997), pp. 549–554.

[15] Gérard Huet. "Structure of a Sanskrit dictionary." INRIA Technical Report, Sept. 2000. Available as: http://pauillac.inria.fr/~huet/PUBLIC/Dicostruct.ps.

[16] Gérard Huet. "From an informal textual lexicon to a well-structured lexical database: An experiment in data reverse engineering." IEEE Working Conference on Reverse Engineering (WCRE'2001), Stuttgart, Oct. 2001.

[17] Gérard Huet. Automata Mista. In "Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday". Ed. Nachum Dershowitz, Springer-Verlag LNCS vol. 2772 (2004), pp. 359–372.

[18] Gérard Huet. A Functional Toolkit for Morphological and Phonological Processing, Application to a Sanskrit Tagger. J. Functional Programming, 15,4 (2005), pp. 573–614.

[19] Gérard Huet and Benoît Razet. The Reactive Engine for Modular Transducers. In "Algebra, Meaning and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday", Eds. Kokichi Futatsugi, Jean-Pierre Jouannaud and José Meseguer. Springer-Verlag LNCS vol. 4060 (2006), pp. 355–374

[20] Gérard Huet and Benoît Razet. Computing with Relational Machines. ICON'2008 tutorial. Preliminary version available at URL http://yquem.inria.fr/~huet/PUBLIC/Pune_tutorial.pdf.

[21] Ronald M. Kaplan and Martin Kay. "Regular Models of Phonological Rule Systems." Computational Linguistics (20,3), 1994, pp. 331–378.

[22] Lauri Karttunen. "Applications of Finite-State Transducers in Natural Language Processing." In Proceedings of CIAA-2000.

[23] Lauri Karttunen. "The Replace Operator." In Proceedings of ACL'95, Cambridge, MA, 1995. Extended version in [37].

[24] K. Koskenniemi. "A general computational model for word-form recognition and production." In Proceedings, 10th International Conference on Computational Linguistics, Stanford (1984).

[25] Eric Laporte. "Rational Transductions for Phonetic Conversion and Phonology." Report IGM 96-14, Institut Gaspard Monge, Université de Marne-la-Vallée, Aug. 1995. Also in [37].

[26] Xavier Leroy et al. "Objective Caml." See: http://caml.inria.fr/ocaml/index.html.

[27] Mehryar Mohri. "Finite-State Transducers in Language and Speech Processing." Computational Linguistics 23,2 (1997), pp. 269–311.

[28] Larry C. Paulson. "ML for the Working Programmer." Cambridge University Press, 1991.

[29] Aarne Ranta. "The GF Language: Syntax and Type System." See: `http://www.cs.chalmers.se/~aarne/GF/`.

[30] Daniel de Rauglaudre. "The Camlp4 preprocessor." See: `http://caml.inria.fr/camlp4/`.

[31] Benoît Razet. Automates modulaires. Mémoire de Master, Université Denis Diderot (Paris 7), 2005.

[32] Benoît Razet. Finite Eilenberg Machines. Proceedings of CIIA 2008, Eds. O.H. Ibarra and B. Ravikumar, Springer-Verlag LNCS vol. 5148 (2008), pp. 242–251.

[33] Benoît Razet. Simulating Finite Eilenberg Machines with a Reactive Engine. In Proceedings of MSFP 2008, Electric Notes in Theoretical Computer Science, `http://gallium.inria.fr/~razet/PDF/razet_msfp08.pdf`.

[34] Benoît Razet. Machines d'Eilenberg Effectives. Thèse de Doctorat, Université Denis Diderot (Paris 7), 2009.

[35] Dominique Revuz. "Dictionnaires et lexiques." Thèse de Doctorat, Université Paris VII, Feb. 1991.

[36] Emmanuel Roche and Yves Schabes. "Deterministic Part-of-Speech Tagging with Finite-State Transducers." Computational Linguistics 21,2 (1995), pp. 227-253.

[37] Emmanuel Roche and Yves Schabes, Eds. "Finite-State Language Processing." MIT Press, 1997.

[38] Richard Sproat. "Morphology and Computation." MIT Press, 1992.

[39] Richard Sproat, Chilin Shih, William Gale and Nancy Chang. "A Stochastic Finite-State Word-Segmentation Algorithm for Chinese." Computational Linguistics 22,3 (1996), pp. 377–408.

[40] Pierre Weis and Xavier Leroy. "Le langage Caml." 2ème édition, Dunod, Paris, 1999.

Index

Ascii (module), **25**, 25, 26
Aum0 (module), **57**
Aumt (module), **61**
Bintree (module), **19**
Dagify (module), **30**
Deco (module), **42**
Gen (module), **6**
Latin (module), **36**
Lexicon (module), **25**
Lexmap (module), **45**
List2 (module), **6**
Make_english_lexicon (module), **30**
Make_french_lexicon (module), **36**
Make_lex (module), **26**
Mini (module), **29**
Minimap (module), **48**
Minitertree (module), **40**
Pidgin (module), **4**
React0 (module), **59**
Reactt (module), **62**
Regular (module), **67**
Sanskrit_engine (module), **86**
Share (module), **27**
Tertree (module), **38**
Transducer (module), **35**
Trie (module), **21**, 25, 26
Unglue (module), **50**
Unglue_test (module), **53**
Word (module), **10**
Zen_lexer (module), **32**
Zipper (module), **13**