The Zen Computational Linguistics Toolkit
ESSLLI 2002 Lectures

June 12, 2002

Gérard Huet

INRIA-Rocquencourt

Contents

Abstract

We present in this course 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 lasy 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 [14]. The Sanskrit Library Documentation, a companion to this document, is available at `http://pauillac.inria.fr/~huet/SKT/DOC/doc.ps` under format postscript, `doc.pdf` under format pdf, and `doc.html` under format html.

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. The Html version uses the Hevea Tex-to-Html translator of Luc Maranget.

1 Pidgin ML

We shall use as *meta language* for the description of our algorithms a pidgin version of the functional language ML [10, 8, 22, 30]. 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,`"CL"`) : (*nat* $\times$ *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:

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 fun $x\ \rightarrow\ x+x$, and it has type $int \rightarrow\ int$. It may be associated to the name *double* by declaring:

value $double\ =$ 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 rec $fact\ n\ =\ n \times (fact\ (n-1))$;

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

value $fst\ =$ 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 [[] $\rightarrow$ *True* | _ $\rightarrow$ *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 permit to sequentialize computation, and to share 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* $\rightarrow$ *e1* | *False* $\rightarrow$ *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* $\rightarrow$ ...];]
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 [20], under the so-called revised syntax [24]. 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 ];
```

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 function.
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

```
value union1 e list = if List.mem e list then list else [e ::list];
```

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

```
value set_of l = List.fold_left (fun acc x → if List.mem x acc then acc else [x :: acc]) [] 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 function *pr* for elements and function *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;
```

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 *)
```

```
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);
```

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.
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.
$diff : word \to word \to delta$

```
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 $w' = patch\ d\ w$.
$patch : delta \to word \to word$

```
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.

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 [11].

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 $List2.unstack$ 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 *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]
  ];
```

occurs occ t = *List.mem occ* (*paths t*). We assume *paths t* to be in *binlexico* order.
bintree_of1 : *binocc* → *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* → *bintree* → *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* → *binocc* → *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* → *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* | *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 = fun
     [ [] → match t with [ Trie(b, l) →
          if b then raise Redundancy
```

```
        else zip_up z (Trie(True, l)) ]
    | [n :: rest] → match t with
        [ Trie(b, l) → 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.

```
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 c = fun
  [ Trie(b, l) → match c with
    [ [] → b
    | [n :: r] → try let t = List.assoc n l
                     in mem r t
                 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
       [ [] → raise (Failure "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 raise (Failure "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 Implementing a lexicon as a trie

Now, assuming the coercion *encode* from strings to words, 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 (Word.encode c)) Trie.empty;
```

strings_of : *trie* → *list string*

```
value strings_of t = List.map Word.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"]);
```

4.3 Building a trie lexicon from an ASCII stream

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

Module Make_lex

```
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 (Word.encode str) }
        with [ End_of_file → output_value stdout lexicon.val ];

trie_of_strings ();
```

For instance, with `english.lst` storing a list of 173528 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 = α;
                                     value size : int; end)
→ sig value share : Algebra.domain → int → Algebra.domain;
       value memo : array (list Algebra.domain); (* for debug *)
   end;
```

Module Share

```
module Share (Algebra :sig type domain = α; 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);
```

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: $0 \leq k < Algebra.size$) 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 = 0 (* linear hash-code parameters *)
and hash1 letter key sum = sum + letter × key
and hash b arcs = (arcs + Gen.dirac b) mod hash_max;

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;
```

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;
```

And now 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 trie of 173528 English words is shrunk from 4.5Mb to 1Mb 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 [9].

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].

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 : $tree \rightarrow zipper \rightarrow tree$

```
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.

$tree\_of$: $word \rightarrow trie$

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

mem_tree : $word \rightarrow tree \rightarrow bool$

```
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 = 0 (* 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;
```

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* → *list word*

```
value contents t = contents_prefix [ ] t
  where rec contents_prefix pref = fun
    [ Deco(i, l) →
      let down = let f l (n, t) = List.append 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 let t = List.assoc n arcs
                     in assoc r t
                 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
    [$Deco(i, arcs)\ \rightarrow$
      if $i = [\,]$ then match $arcs$ with
        [[ ] $\rightarrow$ raise ($Failure$ "next_deco")
        | $[(n, u) :: \_]\ \rightarrow\ next\_rec\ [n :: pref]\ u$
        ]
      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
    [$Deco(i, l)\ \rightarrow$ match l with
      [[ ] $\rightarrow\ List.rev\ acc$
      | _ $\rightarrow$ 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
  [$Deco(i, l)\ \rightarrow$
    $Trie.Trie(\neg\ (i = [\,]),\ List.map$ (fun $(n, t)\ \rightarrow\ (n, forget\_deco\ t))\ l)$
  ];

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

Similar to Deco, except that info is localised to the current word.
α is typically instantiated to a list of morphology functions.

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

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

```
type lexmap α = [ Map of (inverse_map α × marcs α) ]
and marcs α = list (Word.letter × lexmap α);
```

Typically, if word w is stored at a node $Map([...;\ (d,r);\ ...], ...)$, 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.
Invariant : letters are in increasing order in *marcs*, no repetition.

```
type zipm α =
  [ Top
  | Zip of ((inverse_map α) × (marcs α) × Word.letter × (marcs α) × (zipm α))
  ];
```

zip_up : (*zipm* α) → (*lexmap* α) → (*lexmap* α)

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

map_of i w builds the filiform *lexmap* containing w with info i.
map_of : (*inverse* α) → *word* → (*lexmap* α)

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

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]);

value addlex i l = [i :: l];
```

add_inverse : (*inverse* α) → (*inverse_map* (*list* α)) → (*inverse_map* (*list* α))

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

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

```
value add lexmap word i = enter_edit Top lexmap word
  where rec enter_edit z d = fun
     [ [ ] → match d with [ Map(j, l) → zip_up z (Map(add_inverse i j, l)) ]
     | [n :: rest] → match d with
         [ Map(j, l) → let (left, right) = List2.zip n l
                         in match right with
           [ [ ] → zip_up (Zip(j, left, n, [ ], z)) (map_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)) (map_of (single i) rest)
           ]
         ]
     ];

value empty = Map([ ], [ ]);
```

Invariant: *contents* returns words in lexicographic order.

contents : (*lexmap* α) $\rightarrow$ *list* (*word* $\times$ *inverse_map* α)

```
value contents t = contents_prefix [ ] t
  where rec contents_prefix pref = fun
    [ Map(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* $\rightarrow$ (*inverse* α) $\rightarrow$ *unit*) $\rightarrow$ (*lexmap* α) $\rightarrow$ *unit*

```
value iter f t = iter_prefix [ ] t
  where rec iter_prefix pref = fun
    [ Map(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 $\rightarrow$ *word* $\rightarrow$ (*inverse_map* β) $\rightarrow$ α) $\rightarrow$ α $\rightarrow$ (*lexmap* β) $\rightarrow$ α

```
value fold f x t = iter_prefix [ ] x t
  where rec iter_prefix pref x = fun
    [ Map(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* $\rightarrow$ (*lexmap* α) $\rightarrow$ (*inverse_map* α)

```
value rec assoc c = fun
```

```
  [ Map(i, arcs) → match c with
    [ [] → i
    | [n :: r] → try let t = List.assoc n arcs
                     in assoc r t
                 with [ Not_found → [] ]
    ]
  ];
```

The forgetful functor : $trie\_of$: $(lexmap\ \alpha) \rightarrow trie$.

```
value rec trie_of = fun
  [ Map(i, l) →
    Trie.Trie(¬ (i = []), List.map (fun (n, t) → (n, trie_of t)) l)];
```

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 flexed = α; end)
→ sig type flexed_map = Lexmap.lexmap (list Map.flexed);
      value minimize : flexed_map → flexed_map; end;
```

Module Minimap

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

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

```
type flexed_map = Lexmap.lexmap (list Map.flexed);
```

```
value hash_max = 9689; (* Mersenne 21 *)
```

```
module Memo = Share.Share (struct
    type domain = flexed_map;
    value size = hash_max; end);
```

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
  [ Lexmap.Map(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 (Lexmap.Map(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;

end;
```

7.4 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 [12, 13]. 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 future extension of this work will produce the flexed verbal forms as well, a still more productive process, the Sanskrit verbal system being complex indeed.

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 [21, 27], 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 [18] was put in a systematic framework of rational relations and transducers by the work of Kaplan and Kay [15] which is the basis for the Xerox morphology toolset [16, 17, 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). 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. 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 *)

exception Finished;
```

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] →
            try let next_state = List.assoc letter arcs
                in react rest output cont [letter :: occ] next_state
            with [ Not_found → backtrack cont ]
          ]
      in if b then
            let pushout = [occ :: output]
            in if input = [] then (pushout, back) (* solution found *)
               else let pushback = [(input, pushout) :: back]
                    (* we first try the longest possible matching word *)
                    in continue pushback
         else continue back
  ]
and backtrack = fun
  [ [] → raise Finished
  | [(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 ((Word.decode (List.rev word)) ^ "␣");

resume : (resumption → int → resumption)

value resume cont n =
  let (output, resumption) = backtrack cont in
  do { print_string "\n␣Solution␣"
     ; print_int n
     ; print_string "␣:\n"
     ; print_out output
     ; resumption
     };
```

```
value unglue_first sentence = (* similar to unglue *)
  resume [(sentence, [ ])] 1;

value unglue_all sentence = restore [(sentence, [ ])] 1
  where rec restore cont n =
  try let resumption = resume cont n
      in restore resumption (n + 1)
  with [ Finished ->
         if n = 1 then print_string "␣No␣solution␣found\n" else () ];

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"`].

```
let (sol, _) = Childish.unglue (Word.encode "pipicacaboudin")
in Childish.print_out sol;
```

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);

let (sol, cont) = Slang.unglue (Word.encode "fuckatruckshitabrick")
in Slang.print_out sol;
```

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 (Word.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 [28].

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 (Word.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 $Puzzle = Unglue(Phonetic)$;

$Puzzle.unglue\_all$ ($Word.encode$ `"galamandelarenealatourmagnanime"`);

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

```
gal aman de la rene ala tour magnanime
galaman de l arene a la tour magn a nime
```

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 [14] 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.

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

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

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

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

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

[9] 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.

[10] 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).

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

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

[13] 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.

[14] Gérard Huet. “Transducers as Lexicon Morphisms, Phonemic Segmentation by Euphony Analysis, Application to a Sanskrit Tagger.” Available from `http://pauillac.inria.fr/~huet/FREE/tagger.ps`.

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

[16] Lauri Karttunen. “Applications of Finite-State Transducers in Natural Language Processing.” In Proceedings of CIAA-2000.

[17] Lauri Karttunen. “The Replace Operator.” In Proceedings of ACL’95, Cambridge, MA, 1995. Extended version in [27].

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

[19] 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 [27].

[20] Xavier Leroy et al. “Objective Caml.” See: `http://caml.inria.fr/ocaml/index.html`.

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

[22] Larry C. Paulson. “ML for the Working Programmer.” Cambridge University Press, 1991.

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

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

[25] Dominique Revuz. “Dictionnaires et lexiques.” Thèse de doctorat, Université Paris VII, Feb. 1991.

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

[27] Emmanuel Roche and Yves Schabes, Eds. “Finite-State Language Processing.” MIT Press, 1997.

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

[29] 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.

[30] Pierre Weis and Xavier Leroy. “Le langage Caml.” 2ème édition, Dunod, Paris, 1999.

Index