# The Zen Computational Linguistics Toolkit: Lexicon structures and Morphology computations Using a modular functional programming language

**Gérard Huet**

**INRIA**

LEC'2002, Hyderabad

# What the tutorial is about

- A computational platform for Sanskrit
- The ZEN computational morphology toolkit
- Pidgin ML
- The functional programming paradigm for CL
- Concrete programming issues in Objective Caml + Camlp4
- General architecture issues for a CL platform
- Cooperation on free CL resources

Three specific applicative technologies:

- Local processing of focused data
- Sharing
- Finite transducers as lexicon morphisms

## What shall not be discussed

- ML vs C++
- ML vs Java
- ML vs Prolog

# What shall not be discussed at length

- Objective CAML vs SML
- ML vs Haskell
- ML vs C
- Pidgin ML vs Objective CAML

# Basics: lists vs stacks

```
value l5 = [1; 2; 3; 4; 5];

value s5 = [5; 4; 3; 2; 1];

value rec unstack l s =
  match l with
    [ [] -> s
    | [h::t] -> unstack t [h::s]
    ];

value rev l = unstack l [];

value state3 = ([3; 2; 1],[4; 5]);
```

# Turing machines, Emacs, and Zippers

**Zippers.** First presentation at FLoC'96. Published as:
G. Huet. The Zipper. J. Functional Programming 7,5 (1997), 549-554.

Large scale implementations in syntax editors within computational linguistics platforms:

- G. Huet. Lexical morphisms with the Zen platform.
- A. Ranta. Grammatical frameworks.

# Contexts as zippers

```
type tree = [ Tree of forest ]
and forest = list tree;

type tree_zipper =
  [ Top
  | Zip of (forest * tree_zipper * forest)
  ];

type focused_tree = (tree_zipper * tree);
```

A focused tree is a tree with a focus point of interest, i.e. a tree and a stacked context.

# Operations on focused trees

```
value down (z,t) = match t with
   [ Tree(forest) -> match forest 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(unstack l [t::r]))
   ];
```

## More operations on focused trees

```
value left (z,t) = match z with
  [ Top -> raise (Failure "left")
  | Zip(l,u,r) -> match l with
    [ [] -> raise (Failure "left")
    | [elder::rest] -> (Zip(elders,u,[t::r]),rest)
    ]
  ];
value right (z,t) = match z with
  [ Top -> raise (Failure "right")
  | Zip(l,u,r) -> match r with
    [ [] -> raise (Failure "right")
    | [young::rest] -> (Zip([t::l],u,rest),young)
    ]
  ];
```

# Applicative updating

```
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 replace (z,_) t = (z,t);
```

# Points of view about focused structures

- Manipulation of focused data is local
- Redundant representation - efficiency
- The Interaction Combinators Paradigm

Remark. Zippers are linear contexts. They are superior to $\Omega$-terms, notably because the approximation ordering is substructural.

The Natural Transformation from tree functors to zipper functors is Differentiation; Zippers may also be seen as linear functions over trees.

# Back to linguistics

We want to process (parse and generate) natural language sentences, dialogues, corpuses of various kinds (oral, written, news, books, web sites, etc). We assume that the data is already digitalised and discretized as a stream of letters (phonemes for oral data, letters for written one).

A fundamental entity in this processing is the *word*. One traditionally distinguishes processing between streams of letters and words (morphology, lexical analysis) and processing between words and sentences (syntax, parsing).

# Words

Words are represented as list of positive integers.

```
type letter = int (* letters or phonemes *)
and word = list letter;
```

We provide coercions `encode : string -> word` and `decode : word -> string`. Here is lexicographic ordering.

```
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

```
type delta = (int * word);
```

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 tree, as a sequence of ups and downs: if $d = (n, u)$ we go up $n$ times and then down along word $u$.

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 $p$.

The converse of `diff : word -> word -> delta` is `patch : delta -> word -> word`: $w'$ may be retrieved from $w$ and $d = diff\ w\ w'$ as $w' = patch\ d\ w$.

# Tries

Tries, or laxical trees, store sparse sets of words sharing initial prefixes. They are due to René de la Briantais (1959). We use a very simple representation with lists of siblings.

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

Tries are managed (search, insertion, etc) using the zipper technology.

# Important remarks

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.

Such graphs are acyclic (trees). But more general finite state automata graphs may be represented as annotated trees. These annotations account for non-deterministic choice points, and for virtual pointers in the graph.

# Lexicon

Here is a simplistic lexicon compiler
`make_lex : list string -> trie`:

```
value make_lex =
  let enter1 lex c  = Trie.enter lex (Word.encode c)
  in List.fold_left enter1 Trie.empty;
```

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.

Tries share the words by there prefixes, but common suffixes account for a lot of redundancy in the structure. We shall eliminate this redundancy by sharing and get a minimal structure.

# The Share Functor

```
module Share : functor (Algebra:sig type domain = 'a;
                                    value size: int; end) ->
sig value share: Algebra.domain->int->Algebra.domain; end;
```

That is, *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 $k$.

# Memoizing

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

```
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`.

```
value search e = List.find (fun x -> x=e);
```

# The share function

```
value share element key =
  let bucket = memo.(key) in
  try search element bucket with
    [ Not_found ->
      do {memo.(key):=[element::bucket]; element}
    ];
```

Sharing is just recalling!

# Compressing trees as dags

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

```
module Dag = Share (struct type domain=tree;
                           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.

# Dynamic programming

Bottom-up traversing with inductive hash-code computation.

```
value hash1 key index sum = sum + index*key
and hash forest = forest mod hash_max;

value traverse lookup = travel
 where rec travel = fun
  [ Tree(forest) ->
    let f (tries,index,span) t =
       let (t0,k) = travel t
       in  ([t0::tries],index+1,hash1 k index span)
    in let (forest0,_,span) = List.fold_left f ([],1,1) forest
       in let key = hash span
          in (lookup (Tree(rev forest0)) key, key) ];
```

# Compressing a tree as a dag

Now, compressing a tree optimally as a minimal dag is simply effected by a sharing traversal:

```
value compress = traverse Dag.share;

value minimize tree = let (dag,_) = compress tree in dag;
```

# Advantages and extensions

Hashing keys and size is on the client side : we do not delegate hashing to Share, which is just an associative memory. This has two advantages:

- The computation is fully linear
- It is adapted to the statistics of the data

Extension : Auto-sharing types (controlled hash-consing). Suggests a monad of shared hashed structures accommodating entropy of the data.

# Dagified lexicons

We may dagify a lexicon *a posteriori* in one pass:

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

Or we may maintain a dagified structure by sharing dynamically when inserting words by appropriate modification of the zipper operations.

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.

# Advertisement

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).

# Variations

Many variations on tries exist. Optimisations of lexical analysers for programming languages are described in the Dragon book. But the dragon book of computational linguistics has not been written yet.

Variation with ternary trees. Ternary trees are inspired from Bentley and Sedgewick. 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. For our sanskrit lexicon index, the trie takes 221Kb and the tertree 180Kb. Shared as dags the trie takes 103Kb and the tertree 96Kb.

# Decos, Lexmaps, Autos

We understand the Trie structure of a set of Words as a special case of a finitely based mapping Deco = Word → Annotation in the case of Boolean annotations shared by prefix arguments (and by common subexpressions when shared).

We store morphology constructions as being of this type, and we investigate the reverse mapping by generalising them to relations, typically inductively defined through finite state machines.

The more sharing we get the better we optimise this data layout. It is thus of paramount importance that the annotations be local quasi-morphisms decorations.

# Decos

```
type deco 'a = [ Deco of (list 'a * dforest 'a) ]
and dforest 'a = list (Word.letter * deco 'a);
```

We think of the decoration of type `list 'a` as an information associated with the word stored at that node.

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*.

Definition. A deco is a *tree morphism* if the information at every node is a function of the corresponding sub-tree. Such decos preserve the sharing of the trees they decorate.

# Encoding morphological parameters as decorations

We thus profit of the regularity of morphological transformations to have terse representations of the lexicon decorated by grammatical information. Thus if all plurals are obtained by adding ‘s’ to the singular stem except for a few exceptions, we do not pay any cost in encoding this plural information as an explicit instruction `[pl:suffix s]` decorating the stems, since it will not create any new node except for the few exceptions. As opposed to listing explicitly the plural form, which would undo all sharing.

In our sanskrit implementation, the various genders associated with a noun stem are defined in a deco used for producing the flexed forms. The flexed forms are then generated using an ad-hoc internal sandhi algorithm, difficult to encode as a finite-state process, and thus difficult to inverse.

# (Aside) The scoping structure of the lexicon

How to find the stem associated with a gender in the lexicon in one click so that morphology may be displayed - with no need of script or applet.

Simple distributed architecture - all the computation is done on the server side.

Maintaining computational invariants in the lexicon augments its robustness.

# Explicit morphology vs implicit morphology

By explicit morphology I mean listing explicitly the forms generated by morphology operations from root stems, prefixes and suffixes.

By implicit morphology I mean just having programs which will generate these flexed forms on demand.

Implicit morphology is not enough to recognize the segments of sentences identical with a flexed form: the morphological functions must be invertible.

# Compromise

On the other hand, the delimitation between implicit and explicit is blurred since e.g. a finite-state machine state graph may be both considered a program and a piece of data; for instance, a trie stores words, but actually the words are "recognized as being in the lexicon" by "running the lexicon over them as input data".

Thus we shall represent "explicitly" flexed forms and the information on how they are derived from root stems as a trie bearing as decorations instructions on how to "undo morphology" locally. For this purpose, we shall use the notion of *differential word* above. We may now store inverse maps of lexical relations (such as morphology derivations) using the Lexmap structure.

This way we bypass the (hard) problem of internal sandhi fsm axiomatisation.

# Lexmaps

```
type inverse 'a = (Word.delta * 'a)
and inverse_map 'a = list (inverse 'a);

type lexmap 'a = [ Map of (inverse_map 'a * mforest 'a) ]
and mforest 'a = list (Word.letter * lexmap 'a);
```

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 final substrings, and thus being amenable to sharing.

Example: cats and dogs sharing their 's' node while implicitly referring to their respective singular stem.

# 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.

# Example: Sanskrit

The main component in our tools is a structured lexical database. From this database, various hypertext documents may be produced mechanically. The index CGI engine searches for words by navigating in a persistent trie index of stem entries. The current database comprises 12000 items, and its index 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. At present, this deco records genders for 5700 nouns, and it has a size of 268KB.

We iterate on this genders structure a grammatical engine, which generates declined forms. This lexmap records about 120000 such flexed forms with associated grammatical information, and it has a size of 341KB. A companion trie, without the information, keeps the index of flexed words as a minimized structure of 140KB.

# Finite State Lore

Computational phonology are morphology use extensively finite state technology: rational languages and relations, transducers, bimachines, etc.

- Schützenberger
- Koskenniemi
- Kaplan and Kay

Finite state toolsets have been developed, where word transformations are systematically compiled in a low-level algebra of finite-state machines operators. Such toolsets have been developed at Xerox, Paris VII, Bell Labs, Mitsubishi Labs, etc. Compiling complex rewrite rules in rational transducers may be subtle. We depart from this fine-grained methodology and propose more direct translations preserving the structure of the lexicon.

## Finite State Machines as Lexicon Morphisms

We start with the remark that a lexicon represented as a 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. 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, which has computational and robustness advantages.

In the same spirit, we 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*.

# Unglueing

We 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). Consider for instance 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.

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

# Unglue

```
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;
```

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

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.

# The reactive engine code

```
value rec react input output back occ = fun
  [ Trie(b,forest) ->
      if b then let pushout = [occ::output] in
         if input=[] then (pushout,back)  (* solution found *)
         else let pushback = [(input,pushout)::back] in
              continue pushback
      else continue back
    where continue cont = match input with
       [ [] -> backtrack cont
       | [letter :: rest] ->
         try let next_state = List.assoc letter forest in
             react rest output cont [letter::occ] next_state
         with [ Not_found -> backtrack cont ]
       ] ]
```

# Backtrack

```
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,[])];
```

# Remark

Non-deterministic programming is no big deal. Why should you surrender control to a PROLOG blackbox ?

The three golden rules of non-deterministic programming:

- Identify well your search state space
- Represent states as non-mutable data
- Prove termination

The last point is essential for understanding the granularity and enforcing completeness.

# More on state space considerations

This non-deterministic process (recognizing $L^*$) uses the **same** state space as the lexicon/trie (recognizing $L$).

This corresponds to the fact that an automaton for $L^*$ may be obtained from the automaton for $L$ by inserting $\epsilon$-moves from accepting nodes to the initial node. But such transitions may be kept completely implicit. All you have to do is to manage the necessary non-determinism (continuing in $L$ which is not in general a prefix language (i.e. if may happen that both $w$ and $w \cdot s$ are in $L$) versus iterating) in the backtrack stack, but you do not have to modify at all the state space data structure. It is just a shift in point of view concerning this data.

# Still more on state space considerations

Remember that dagified tries define the minimal automaton of a finite language $L$.

But it is not the case that this automaton, completed with $\epsilon$ transitions, is minimal for $L*$. Consider for instance $L = \{a, aa\}$.

However, note that we are using it as a transducer computing *justifications* for a word in $L*$ to be a concatenation of precise words of $L$, and the minimal automaton does not keep enough information for that: distinct segmentations of a sentence must be separated.

# Childtalk

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

module Childish = Unglue(Childtalk);

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

We recover as expected: `pipi caca boudin`.

# Generating several solutions

We resume a resumption with

`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_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 () ];
```

# Solving a charade

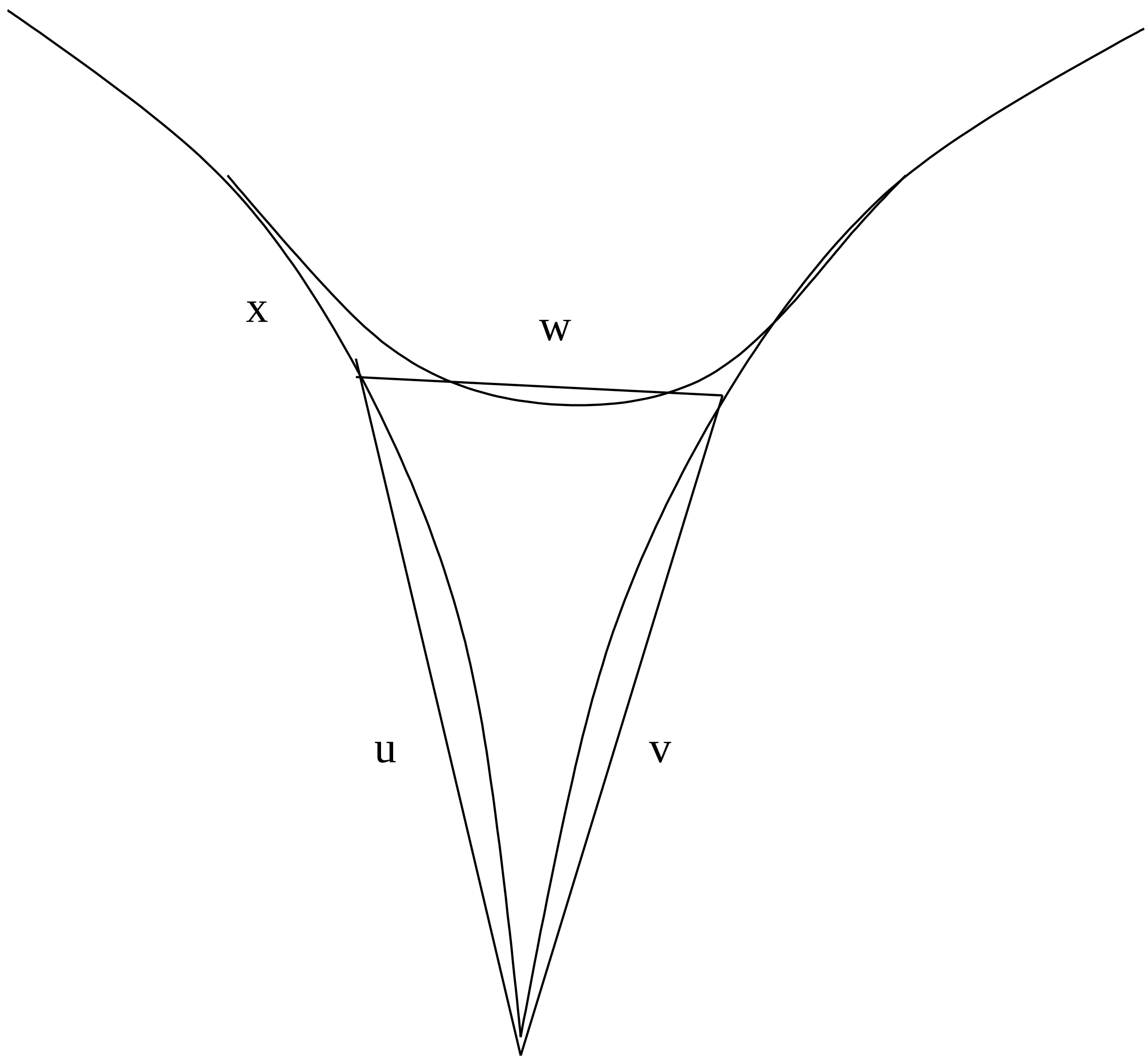
```
module Short = struct
value lexicon = Lexicon.make_lex
  ["able"; "am"; "amiable"; "get"; "her"; "i"; "to"; "together"];
end;

module Charade = Unglue(Short);

Charade.unglue_all (Word.encode "amiabletogether");

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

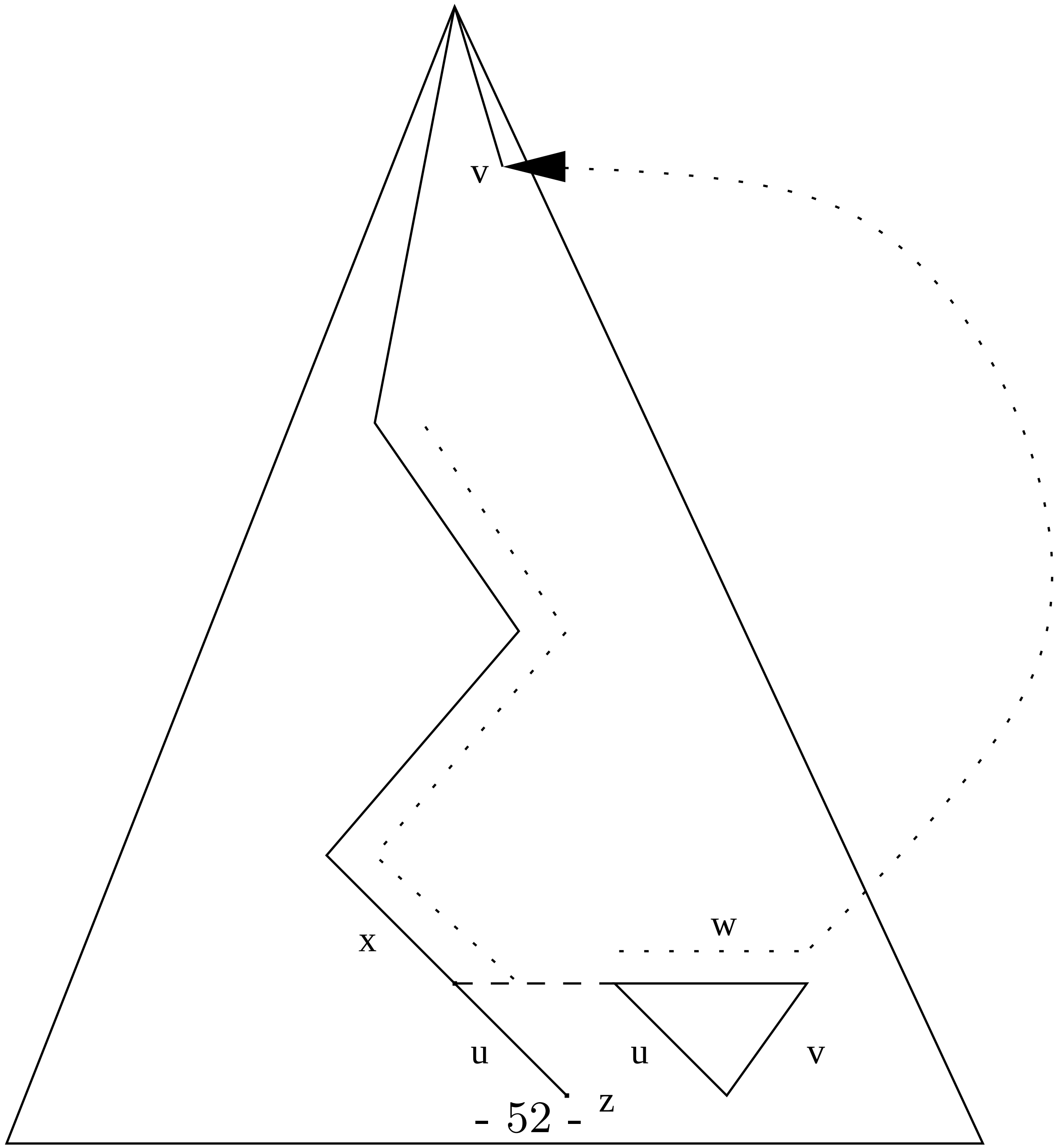
# Juncture euphony and its discretization

When successive words are uttered, the minimization of the energy necessary to reconfigurate the vocal organs at the juncture of the words provoques a euphony transformation, discretized at the level of phonemes by a contextual rewrite rule of the form:

$$[x]u|v \rightarrow w$$

This juncture euphony, or *external sandhi*, is actually recorded in sanskrit in the written rendering of the sentence. The first linguistic processing is therefore segmentation, which generalises unglueing into sandhi analysis.


x
w
u
v


v
x
w
u
u
v
z

# Auto

```
type lexicon = trie
and rule = (word * word * word);
```

The rule triple (`rev u, v, w`) represents the string rewrite $u|v \rightarrow w$.
Now for the transducer state space:

```
type auto = [ State of (bool * deter * choices) ]
and deter = list (letter * auto)
and choices = list rule;

module Auto = Share (struct type domain=auto;
                            value size=hash_max; end);
```

We assume linear hash functions `hash0`, `hash1`, `hash`.

# Compiling the lexicon to a minimal transducer

```
(* build_auto : word -> lexicon -> (auto * stack * int) *)
value rec build_auto occ = fun
  [ Trie(b,arcs) ->
     let local_stack = if b then get_sandhi occ else []
  in let f (deter,stack,span) (n,t) =
     let current = [n::occ]     (* current occurrence *)
     in let (auto,st,k) = build_auto current t
        in ([(n,auto)::deter],merge st stack,hash1 n k span)
  in let (deter,stack,span) = fold_left f ([],[],hash0) arcs
  in let (h,l) = match stack with
                 [[] -> ([],[]) | [h::l] -> (h,l)]
  in let key = hash b span h
  in let s = Auto.share (State(b,deter,h)) key
  in (s,merge local_stack l,key) ];
```

# Segmenting Transducer Data Structures

```
type transition =
  [ Euphony of rule (* (rev u,v,w) st u|v -> w *)
  | Id             (* identity or no sandhi *)
  ]
and output = list (word * transition);
type backtrack =
  [ Next of (input * output * word * choices)
  | Init of (input * output)
  ]
and resumption = list backtrack; (* coroutine resumptions *)

exception Finished;
```

## Running the Segmenting Transducer

```
value rec react input output back occ = fun
  [ State(b,det,choices) ->
    (* we try the deterministic space first *)
    let deter cont = match input with
         [ [] -> backtrack cont
         | [letter :: rest] ->
      try let next_state = List.assoc letter det
          in react rest output cont [letter::occ] next_state
      with [ Not_found -> backtrack cont ]
         ] in
    let nondets = if choices=[] then back
                  else [Next(input,output,occ,choices)::back]
    in if b then
          let out = [(occ,Id)::output] (* opt final sandhi *)
```

```
          in if input=[] then (out,nondets) (* solution *)
             else let alterns = [ Init(input,out) :: nondets ]
                  (* we first try the longest matching word *)
                  in deter alterns
       else deter nondets
  ]
and choose input output back occ = fun
  [ [] -> backtrack back
  | [((u,v,w) as rule)::others] ->
       let alterns = [ Next(input,output,occ,others) :: back ]
       in if prefix w input then
             let tape = advance (length w) input
             and out = [(u @ occ,Euphony(rule))::output]
             in if v=[] (* final sandhi *) then
                   if tape=[] then (out,alterns)
                   else backtrack alterns
```

```
                  else let next_state = access v
                       in react tape out alterns v next_state
            else backtrack alterns
  ]
and backtrack = fun
  [ [] -> raise Finished
  | [resume::back] -> match resume with
      [ Next(input,output,occ,choices) ->
            choose input output back occ choices
      | Init(input,output) ->
            react input output back [] automaton
      ]
  ];
```

# Example of Sanskrit Segmentation

```
process "tacchrutvaa";

Chunk: tacchrutvaa
may be segmented as:

 Solution 1 :
[  tad with sandhi d|"s -> cch]
[  "srutvaa with no sandhi]
```

# More examples

```
process "o.mnama.h\"sivaaya";

 Solution 1 :
[  om with sandhi m|n -> .mn]
[  namas with sandhi s|"s -> .h"s]
[  "sivaaya with no sandhi]

process "sugandhi.mpu.s.tivardhanam";

 Solution 1 :
[  sugandhim with sandhi m|p -> .mp]
[  pu.s.ti with no sandhi]
[  vardhanam with no sandhi]
```

# Sanskrit Tagging

```
process "sugandhi.mpu.s.tivardhanam";

 Solution 1 :
[  sugandhim
 < { acc. sg. m. }[sugandhi] >  with sandhi m|p -> .mp]
[  pu.s.ti
 < { iic. }[pu.s.ti] >  with no sandhi]
[  vardhanam
 < { acc. sg. m. | acc. sg. n. | nom. sg. n.
   | voc. sg. n. }[vardhana] >  with no sandhi]
```

# Statistics

The complete automaton construction from the flexed forms lexicon takes only 9s on a 864MHz PC. We get a very compact automaton, with only 7337 states, 1438 of which accepting states, fitting in 746KB of memory. Without the sharing, we would have generated about 200000 states for a size of 6MB!

The total number of sandhi rules is 2802, of which 2411 are contextual. While 4150 states have no choice points, the remaining 3187 have a non-deterministic component, with a fan-out reaching 164 in the worst situation. However in practice there are never more than 2 choices for a given input, and segmentation is extremely fast.

## Overgeneration Problems

Very short particles have to be treated differently, or otherwise there would be intolerable overgeneration. Probably prosody will have to come to the rescue. The case of vedic “u”.

Compounds. The *bahuvrīhi* problem.

Intrinsic overgeneration. *a+a=a+ā=ā+a=ā+ā=ā* Most s.m. end with *a*, many s.f. end with *ā*, the preverb *ā* (towards) is frequent, the prefix *a* is common (negation). So there is often room for interpretation !

E.g. *na asato vidyate bhāvo na abhāvo vidyate sataḥ*
vs *na asato vidyate abhāvo na abhāvo vidyate sataḥ*

Double entendre poetry.

# Soundness and Completeness of the Algorithms

**Theorem**. If the lexical system $(L, R)$ is strict and weakly non-overlapping $s$ is an *(L,R)-sentence* iff the algorithm ($segment\_all\ s$) returns a solution; conversely, the (finite) set of all such solutions exhibits all the proofs for $s$ to be an *(L,R)-sentence*.

**Fact**. In classical Sanskrit, external sandhi is strongly non-overlapping.

Cf. `http://pauillac.inria.fr/~huet/FREE/tagger.ps`

# Where is the information?

Mel'cuk says "Everything is in the lexicon".

The key concept is lexicon directed. So most of the information is indeed in the lexicon. But a lot of phonological information (sandhi rules) and grammatical knowledge is in the code.

If time permits. A tour of the dictionary structures.

# Enjoy!

- Sanskrit site: http://pauillac.inria.fr/~huet/SKT/
- Sandhi Analysis paper:
  http://pauillac.inria.fr/~huet/FREE/tagger.ps
- Course notes:
  http://pauillac.inria.fr/~huet/ZEN/esslli.ps
- Course slides:
  http://pauillac.inria.fr/~huet/ZEN/Trento.ps
- Tutorial slides:
  http://pauillac.inria.fr/~huet/ZEN/Hyderabad.ps
- ZEN library: http://pauillac.inria.fr/~huet/ZEN/zen.tar
- Objective Caml: http://caml.inria.fr/ocaml/

## What next (on the Sanskrit front)

- Sanskrit 1 Verb morphology, Corpus testing, Lexicon acquisition mode, Segmentation training, Philology assistant (Scharf, Smith)
- Sanskrit 2 Sentinels, Prosody, Valency checking, Dependency synthesis
- Sanskrit 3 Discourse analysis: Reference, Scope, Theme, Focus, Anaphora resolution, Extra-linguistic information
- Sanskrit $\infty$ Distributed development of multilingual tools, Saving the Pune dictionary project

# What next (on the Zen front)

- Zen maintenance Distribution, Hotline, Users' club, Coordination of extensions
- Zen immediate extensions Grafting of regular relations, Rules compiler
- Towards a more comprehensive generic platform for computational linguistics, accommodating the levels of Syntax, Semantics, and Discourse Information Dynamics