From an informal textual lexicon to a well-structured lexical database: An experiment in data reverse engineering

Gérard Huet
INRIA-Rocquencourt
78150 Le Chesnay France

Abstract

We present an experiment in data reverse engineering in the field of computational linguistics. We explain a methodology which preserves to a great extent the original input format, allowing parallel acquisition/update of the data with processing at a more structured representation level. We motivate the use for such applications of Objective Caml, a functional programming language with strong static typing, parametric modules and meta-linguistic technology.

1 Introduction

This paper describes an experiment conducted by the author to turn a loosely structured textual Sanskrit to French lexicon into a comprehensive lexical data base, used as unique source for a variety of uses:

- a high-quality printable 320 pages Sanskrit-to-French dictionary, with entries listed both in *devanāgarī* script and in roman transcription with diacritics
- a Web site giving a fully tagged hypertext view of the same document
- various search tools of this web site, notably an index engine returning all entries (and the declensions thereof) matching a given prefix, a simplified index where diacritics may be omitted from the query, and a grammatical engine computing declensions of a given stem.

This data base, freely available on Internet[6], is now used as a testbench for computational linguistics[8].

In a first part of this paper we describe the methodology used. In a second part, we discuss the appropriateness of functional programming and of the W3C and other Internet standards for this type of application.

2 From loose text processing to context-free syntax analysis

The initial form of the lexicon was a source TEX document edited as a text file under the Emacs editor. It was thus a semi-structured document, in the sense that lexical entries were described using a combination of unstructured text with TEX macros. This effort was started from scratch in 1993, and had grown by middle 1996 to a text file of 1.2Mb, listing 8000 entries. Here is a sample entry of the document at that time:

```
\word{$kumAra$}{kum\=ara} m.
gar{\c c}on, jeune homme
\or prince; page; cavalier
\or myth. np. de Kum\=ara ``Prince'',
    \'epith. de Skanda
-- n. or pur
-- \fem{kum\=ar{\=\i}\/} adolescente, jeune
   fille, vierge.
```

Let us give a few comments on the above entry. The special string `$kumAra$` is actually a notation for the *devanāgarī* representation. It is interpreted by a specific processor `devnag`, originally due to Franz Velthuis[12], which in a first pass on the document analyses the Sanskrit words in syllables, combines their consonants in terms of *devanāgarī* ligatures, and compiles the corresponding sequences of ligatures in terms of sequences of glyphs in a low-level Metafont glyph alphabet. The corresponding C processor transforms thus `$kumAra$` into a cryptic `{\dn \7{k}mAr}` which can then be interpreted by TEX enriched with the `dn` font.

Despite the use of a few macros like `or`, `word` and `fem`, the source was extremely low level, there was no clear notion of scope of the various constituents, there were two distinct notations for diacritics, one using the `devnag` convention such as `kumAra`, the other one using TEX input conventions for roman alphabet with diacritics such as `kum\=ara` - used redundantly in the

two arguments of the word macro without mechanical check of consistency. Sanskrit words freely appeared in the French text, where accents were encoded in the TEX way. Structural markers such as the separator of grammatical roles used directly the typographical encoding -- instead of an explicit macro, and low-level typographical details such as the italic correction \/ appeared explicitly. It was hardly possible to debug typing mistakes, which could result in TEX errors showing up many pages after the typo. Needless to say, the document had become extremely hard to maintain. A more disciplined way of proceeding was clearly in order.

The first significant improvement was to isolate all the text proper from the TEX meta-notation, to use systematically macros for the structural units, and to terminate every entry and sub-entry with explicit closing markers. This first pass was effected mechanically by Emacs macros. Finally, French accents were coded in the ISO-LATIN1 standard 8-bit encoding. Here is the result on our sample entry:

```
\word{$kumAra$}{kum\=ara}
\sm \sem{garçon, jeune homme}
\or \sem{prince; page; cavalier}
\or \myth{np. de Kum\=ara ``Prince'',
          épith. de Skanda}
\role \sn \sem{or pur}
\role \fem{kum\=ar{\=\i}\/} \sem{adolescente,
      jeune fille, vierge}
\fin
```

We remark that we still have redundancy of diacritics encodings, and undisciplined intermixing of Sanskrit names such as Kum\=ara with French text. But at least the structure of the dictionary entries is implicit in the TEX macros.

The next step was to make explicit the lexicon structure by parsing the source as belonging to a context-free language identifying structural regularities. Actually 3 context-free grammars were identified, corresponding respectively to the structures of a generic lexicon, of Sanskrit quotations, and of French sentences. These 3 grammars are each defined with their own lexical analyser, so that diacritic encodings may appear only in the Sanskrit quotations, and accented letters may appear only in the French sentences. Before giving details on this grammatical apparatus, let us give the final source of our example entry:

```
\word{kumaara}
\sm
\sem{garçon, jeune homme}
\or \sem{prince; page; cavalier}
\or \myth{np. de \npd{Kumaara} "Prince",
          épith. de \np{Skanda}}
\role \sn \sem{or pur}
\role \fem{kumaarii} \sem{adolescente,
      jeune fille, vierge}
\fin
```

The transformation between the previous representation and the final one was effected semi-mechanically, by a mixture of mutual adjustment between the lexicon source and the grammar description. Some of these adjustments were aided by mechanical tools such as Emacs macros or other transducers. Others were hard to track. For instance, the explicit tagging of proper names in Sanskrit (with the label np for used occurrences and npd for defined occurrences) was easy for names where diacritics occur, such as Kum\=ara above, but hard for the others such as Skanda, for which only human intervention may decide whether the name belongs to Sanskrit or to French.

The technology used for the description of the lexical analysers being naturally fit to describing regular transducers, it was then easy to adapt it to the generation of the redundant diacritics encodings from a unique canonical description, which subsumes the two previous representations kumAra and kum\=ara into the unique kumaara, achieving one of the stated objectives.

We chose Objective Caml[14, 19, 1] as our programming language for this transduction activity. We shall justify this technological choice at the end of this article, after describing the rest of our processing tools. For the moment, we remark that the Ocaml preprocessor Camlp4[16] is ideally suited to the description of parsers in a high level notation smoothly integrated in the syntax of this variety of the ML programming language.

For instance, here is a portion of the source code of the transducer from diacritics encodings à la Velthuis into TEX notation:

```
value tex = Gram.Entry.create "skt to tex";
GEXTEND Gram
  tex:
    [ [ LETTER "a"; LETTER "a" -> "\\=a"
      | LETTER "a" -> "a"
      | LETTER "i"; LETTER "i" -> "{\\=\\i}"
      | LETTER "i" -> "i"
      | LETTER "u"; LETTER "u" -> "\\=u"
      | LETTER "u" -> "u"
```

```
      | "~"; LETTER "n" -> "\\~n"
      | "."; LETTER "t" -> "{\\d t}"
      | "."; LETTER "d" -> "{\\d d}"
      | "."; LETTER "s" -> "{\\d s}"
    ...
      | i = LETTER -> i
      | i = INT -> i
      | EOI -> raise Exit ] ];
END;
```

where `LETTER` and `INT` are defined as lexical tokens in our custom lexical analyser `Trans.lexer`, and `Gram` is a grammar module, initialised as follows:

```
module Gram = Grammar.Make
 (struct value lexer = Trans.lexer (); end);
```

That is, the Camlp4 standard library `Grammar` provides a generic functor (parametric module) `Make` which may be instanciated by a specific lexer structure in order to generate a module `Gram` which contains as components all the necessary parsing tools such as entry creation (`Gram.Entry.create`). One of these components is a parser `Gram.Entry.parse` which is a function from entries (such as `tex` defined above) to stream analysers. In our case, a stream analyser is a function from streams to strings. It is now easy to program a generic function `transducer`, which given an entry returns a function from character streams to strings, and the specific transducer from diacritics encodings to TEX encodings is simply obtained by partial application:

```
value skt_to_tex = transducer tex;
```

This very elegant implementation of the ideas exposed in the article by de Rauglaudre and Mauny[2] is furthermore quite efficient. Once the concepts are mastered, we are led to very smooth programs which are self documented and easy to maintain. Our reverse engineering exercise used dozens of such transducing engines, seamlessly embedded in the rest of our ML modules, and whose strong static typing guarantees correct execution.

3 From context-free concrete syntax to strongly typed abstract syntax

Let us state the requirements of our parser. First, it should preserve all the information contained in the data base, including typographical hints such as italic correction and hyphenation pragmas; it should also preserve comments, which contain valuable information about origin and degree of confidence of certain items, references to other sources of information, confirmation or absence in classical dictionaries, etc.

Another requirement is that it should as much as possible preserve the flexibility of the original format; the opposite requirement, that it should not be overly complex, defined a trade-off between preserving valuable notational flexibility and avoiding baroque and ad-hoc details.

This trade-off was made explicit by the requirement of designing an abstract syntax, defining an algebraic representation of the lexicon; this abstract syntax was made concrete as a set of recursive ML type definitions; to every non-terminal of the grammar ought to correspond a unique type, the various choices for the corresponding grammar entry corresponding to the various constructors of the relevant type; lists corresponded to the Kleene star operation, whereas option types corresponded to epsilon unions; modulo these regular operations, we aimed at isomorphism between abstract and concrete syntax; each grammar rule was associated with a semantic action, constructing a value of the appropriate type, usually the corresponding constructor, applied recursively to the values associated to the semantic actions of the non-terminals appearing in the production; the abstract syntax corresponding to a given input string was thus constructed as a well-typed ML value synthesized by bottom-up parsing.

The design of this abstract syntax was really the creative activity underlying this reverse engineering process, similar to the definition of the DTD[24] of our dictionary structure. It should be stressed at this point that an *a priori* design of this DTD from scratch would have been hopeless, in the sense that it would have had to be revised incessantly along with the dictionary growth, whereas starting from an 8000 entries existing dictionary gave us a much better insurance that most peculiarities and exceptions were already present - we indeed believe that the structure we came up with will scale up with minor adjustments to future growth of the lexical data base.

Let us give a concrete example of this abstract syntax, together with a typical set of grammar productions. There are two kinds of entries in our dictionary, genuine entries containing lemmatic information about a lexical item, and cross-references which point typically to an orthographic variant to its proper lemma. This dictinction corresponds to two productions in the dictionary grammar pertaining to the non-terminal `entry`:

```
entry:
  [ [ s = syntax; u = usage; oc = OPT cogs
```

```
      -> Entry(s,u,oc)
    | s = syntax; c = crossref
      -> Crossref(s,c)
  ] ] ;
```

these two productions produce both values of an ML type `entry`, constructed respectively with data constructors `Entry` and `Crossref`. Thus we find in the interface module `Dictionary` a datatype declaration:

```
type entry =
   [ Entry of (syntax * usage * option cogs)
   | Crossref of (syntax * crossref)
   ];
```

We can thus witness the expected isomorphism between the parse trees of the concrete syntax non-terminal `entry` and the values of the abstract syntax type `entry`. This parallel correspondance may be understood top-down across all the structure. Thus an `Entry` value has 3 sub-components, two mandatory ones, corresponding roughly to the syntactic information concerning the entry (orthographic stem for each grammatical rôle, etymological information, etc.) and to its usage information (meanings, expected valency of dependent items, etc.), the third one being optional (cognate etymologies in other Indo-European languages such as Greek, Latin, Germanic, English, French, Slavic, Hindi, etc.) We remark that the optional concrete syntax component `OPT cogs` has type `option cogs`, where the polymorphic type constructor `option` is the OCaml library sum constructor, defined as:

```
type option 'a = [ None | Some of 'a ];
```

Similarly, Ocaml grammars allow the Kleene star operator `LIST x` to stand for values of type `list t`, where the non-terminal `x` constructs values of type `list t`.

We shall not detail further the abstract structure of our dictionary, which is fully explained in a technical report[7]; let us just remark that it is highly non trivial: the three mutually recursive grammars defining its syntax comprise a total of 321 productions, of which 14 pertain to Sanskrit quotations, 54 pertain to French sentences, and 253 pertain to the generic dictionary structure. It would have been hopeless to guess beforehand this structure, whereas it turned out to be relatively straightforward to induce this structure from the dictionary raw data itself, after a few iterations for smoothing out irregularities and mistakes.

We also remark that our lexical analysers are consistent with TEX lexical conventions, and that for instance we preserve comments in our source text, which contain valuable information about the original sources used in compiling the dictionary. But our parser enforces many more structural invariants than an arbitrary TEX document. Thus typos and other input mistakes are spotted by our parser exactly where these invariants are violated, with an explicit error message giving the precise location of the mistake, allowing continuous growth of the dictionary with reasonably good debugging facilities (mistakes indicate the precise line and character number where the syntax is violated).

We finish this section by pretty-printing our example entry above, as the ML representation of its abstract syntax:

```
(Entry
 (((Word "kumaara"), [], None),
  (Subst
   [(Meaning
     (Unpos (Genderorth (([Mas], None))),
   [([], (Sem
        [(Affirm [[(Unit "garçon")];
     [(Unit "jeune"); (Unit "homme")]])]));
    ([], (Sem
        [(Affirm [[(Unit "prince")]]);
         (Affirm [[(Unit "page")]]);
         (Affirm [[(Unit "cavalier")]])]));
    ([], (Myth
        [(Affirm [[(Unit "np."); (Unit "de");
                  (Npd "Kumaara");
 (Quotation (Affirm [[(Unit "Prince")]]))];
         [(Unit "épith."); (Unit "de");
          (Np "Skanda")]])]))]);
   (Meaning
    (Unpos (Genderorth (([Neu], None))),
     [([], (Sem [(Affirm [[(Unit "or");
                  (Unit "pur")]])]))]);
   (Meaning
    (Unpos (Genderorth (([Fem],
            Some "kumaarii"))),
     [([], (Sem [(Affirm
   [[(Unit "adolescente")]; [(Unit "jeune");
         (Unit "fille")];
        [(Unit "vierge")]])])]))]),
    None, true))
```

Alternatively, we could give an XML representation of the same:

```
<Entry>
 <Triple>
  <Word> "kumaara" </Word>
  ...
 </Triple></Entry>
```

4 From abstract syntax to low-level formating instructions

The main procedure of our dictionary processor is a read-eval-loop, which parses successive items, constructs their abstract syntax, and then proceeds in pretty-printing recursively this structure in terms of low-level typographical instructions.

Thus, in the case of our example entry above, the generated TEX file contains the following low-level code:

```
{\dn \7{k}mAr}~{\sl kum\=ara} m.
garçon, jeune homme
\(\mid\:\)prince; page; cavalier
\(\mid\:\)myth. np. de Kum\=ara ``Prince'',
     épith. de Skanda
--- n. or pur
--- f. {\sl kum\=ar{\=\i}} adolescente,
     jeune fille, vierge.
```

One may wonder at this point why we took all the trouble to compile high-level TEX text into a low-level typographically equivalent one! But now we are ready to reap the reward of our abstraction, by replacing easily the backend (TEX generator) by another formating generator, or by an arbitrary computation on the dictionary structure, for that matter. Furthermore, by proper use of the modularity constructs of OCaml, we may factor out the recursive traversal of the structure from the backend generator, reduced to low-level operations on the terminal strings. Let us explain a little this modular structure.

The main component `grind` is a parametric module, or *functor* in ML terminology. Here is exactly the module type `grind.mli` in the syntax of Ocaml:

```
module Grind : functor
 (Process:Proc.Process_signature) -> sig end;
```

with interface module `Proc` specifying the expected signature of a `Process`:

```
module type Process_signature = sig
value process_header :
   (Sanskrit.skt * Sanskrit.skt) -> unit;
value process_entry :
   Dictionary.entry -> unit;
value prelude : unit -> unit;
value postlude : unit -> unit;
end;
```

That is, there are two sorts of items in the data base, namely headers and entries. The grinder will start by calling the process `prelude`, will process every header with routine `process_header` and every entry with routine `process_entry`, and will conclude by calling the process `postlude`. The module interface `Dictionary` describes the dictionary abstract syntax as a set of mutually recursive ML datatypes, whereas module `Sanskrit` holds the private representation structures of Sanskrit nouns (seen from `Dictionary` as an abstract type, insuring that only the Sanskrit lexical analyser may construct values of type `skt`).

A typical process is the printing process `Print_dict`, itself a functor. Here is its interface:

```
module Print_dict : functor
  (Printer:Print.Printer_signature)
   -> Proc.Process_signature;
```

It takes as argument a `Printer` module, which specifies low-level printing primitives for a given medium, and defines the printing of entries as a generic recursion over its abstract syntax. Thus we may define the typesetting primitives to generate a TEX source in module `Print_tex`, and obtain a TEX processor by a simple instanciation:

```
module Process_tex = Print_dict(Print_tex);
```

But similarly, we may now define the primitives to generate the HTML source of a Web document, and obtain an HTML processor `Process_html` as:

```
module Process_html = Print_dict(Print_html);
```

It is very satisfying indeed to have such sharing in the tools that build two ultimately very different objects, a book with professional typographical quality on one hand, and a Web site fit for hypertext navigation and search on the other hand. But maybe a few words are in order to explain how indeed the hypertext structure is implicit in the abstract syntax representation.

5 Scoping and name management

There are several notions of scoping in the lexical database. They emerged progressively, first from the need to provide correct hypertext reference to Sanskrit quotations, and thus complete decoration by definition anchors. Since we want direct access through the whole document, this gives a notion of global scope of dictionary entries and proper name definitions and other explicitly binding names. The important meta-notion which emerged was that certain slots of the abstract structure algebra constructors where *binding* over their (name) arguments - in our case it requires

the corresponding type of this argument to be `skt`, the sort of Sanskrit references.

Thus a number of such "binding constructors" were identified. Conversely, two unary reference constructors were identified, corresponding in the French text to the quotations to Sanskrit common and proper names respectively.

The processing of such binding slots stays transparent in the TEX printing; in the HTML version, it generates appropriate URL anchors and references respectively. The generating phase constructs a trie index of the binding references, warning for duplicated bindings. A second checking phase accesses this persistent structure at every reference for warning of missing anchors. We thus enforce in this second step the overall construction of a consistent Web site.

At this point we reach the concept of a module type signature, corresponding roughly to a DTD in XML parlance, with constructors binding in certain arguments. An abstract syntax value, with names in binding and reference slots, may be considered as a kind of graph representation.

The next notion was induced from a further extension of our computational linguistics tools, namely a declension grammatical engine. The problem was to generate all flexed forms corresponding to the basic stems in the lexicon. In Sanskrit, the declension of a substantive (or adjective, pronoun, and number) depends on the terminal ending of its stem string and on its gender; in order to automate this computation by a simple traversal of the dictionary structure, a new scope constraint arose: each stem possibly associated with an entry had to be associated with its possible genders. Here the scoping constraint is more complex than statically determining global binding slots, and it corresponds to a dynamic top-down traversal of the abstract syntax tree. During this traversal the current stem is remembered, so that each gender declaration may register the proper information in a trie associating with each entry all such pairs (stem,gender). A final pass on the trie calls a declension engine, in order to build a big trie of all flexed words. Furthermore, the gender declarations generate references to the declension engine seen as a CGI binary, associated to the appropriate arguments. This way the declensions of a given entry are dynamically computable from within the dictionary hyper-structure.

We shall not give full details on such tools, which are more fully documented in reference[8], but the basic idea is to use the abstract syntax of the lexicon entries to make all kinds of computations, such as the verification of consistency constraints. One simple such constraint concerns the proper alphabetic ordering of the entries, a not entirely trivial check in Sanskrit, where for instance a generic nasalisation notation (the so-called *anunāsika*) stands for the nasal consonant which is homophonic (*savarṇa*) to the following consonant.

We end this section by giving the HTML representation obtained for our example entry:

```
<P CLASS="dico">
 <A CLASS="defn"; NAME="kum=ara">
    <I>kum&#257;ra</I></A>
 <A CLASS="defr";
  HREF="http:XX/dicdecl?query=kumaara &
       gender=Mas">m.</A>
gar&#231;on, jeune homme; fils
| prince; page; cavalier
| myth. np. de <A CLASS="defn";
   NAME="*kum=ara">Kum&#257;ra</A>
&#171;Prince&#187;, &#233;pith. de
  <A HREF="s.html#*skanda">Skanda</A> &#8212;
  <A CLASS="defr";
   HREF="http:XX/dicdecl?query=kumaara &
        gender=Neu">n.</A>
or pur &#8212;
  <A CLASS="defr";
   HREF="http:XX/dicdecl?query=kumaarii &
        gender=Fem">f.</A>
  <A CLASS="defn"; NAME="kum=ar=i">
     <I>kum&#257;r&#299;</I></A>
adolescente, jeune fille, vierge.
```

The string `XX` above stands for the URL of the CGI binary directory of the intended HTTP server, and is installation dependent. Note how invocations of the grammatical engine `dicdecl` are precomputed as HTML references triggered by clicking on the gender specifications. Thus all possible declensions of the stems associated with an entry are just one click away from the dictionary navigator.

6 The compiling chains

The overall architecture of our software platform is a collection of processes, called as parameters to the generic `Grind` functor, in order to make some computation during the recursive traversal of the various entries, computed on the fly while parsing the data base; global information is kept in tries structure, which maintain a maximally shared representation. It is to be remarked that the progressive construction of these tries is completely applicative, being implemented as functional zippers[5].

Two such processes compute respectively the TEX source of the paper dictionary, available in PS or PDF format, and a consistent Web site which implements a sort of hypertext encyclopedia, with various executable families of links, permitting easy navigation to quotations, to etymological parents, to synonyms and antonyms, etc, and also specific grammatical tabulations such as declension tables. We finally stress that these two 'prettyprinters', which deliver such different final products, are actually two instanciations of a generic pretty-printing functor with backend modules giving the representations in respectively TEX and HTML of low level typographical primitives (such as blank, line, print_ident, etc). We thus fully use the module constructions of Objective Caml.

It is to be noted that at some point the encoding of French sentences, which originally was just an encapsulation of ISO-LATIN1 strings, had to be dropped in favor of Unicode UTF-8 encodings, in order to represent diacritics, for which we used a font designed by Chris Fynn[4]. But the corresponding modification consisted in just piping the 8-bits strings into an ISO-LATIN1 to UTF-8 transducer, a matter of 15 minutes programming with our generic transduction function. Thus we had the best of both worlds - Unicode encoding of the Web pages, allowing arbitrary fonts in the dictionary Web site, while preserving ISO-LATIN acquisition with AZERTY (french) keyboards in the database source.

7 Implementation notes

7.1 ML as a software engineering workhorse

We believe that Objective Caml, together with the syntax facilities of its companion macro-processor Camlp4, is ideally suited to such processing. Its stream library is extremely efficient, and the whole 1.7Mb of source database is completely processed into say the web site in less than 1mn on a 500MHz PC. We recall that Objective Caml is available both in bytecode version, for easy debugging, and in optimised native code competitive with C programming. The kernel of Objective Caml being itself written in ANSI C, it is available on all current platforms.

One of the main strengths of OCaml is the versatility of its programming paradigms: one may program in a purely applicative functional style, or one may mix applicative and imperative features. For instance, we made an extensive use of mutable data structures for the representation of our indexes (tries), although these tries were implemented in a completely applicative way as functional zippers[5].

OCaml may indeed be considered as the modern successor of both LISP-Scheme and Pascal-Modula. It benefits from the safety of strong static typing, but polymorphism together with the synthesis of principal types by the compiler relieves the programmer from cumbersome explicit type declarations. Its module system is a specially powerful abstraction mechanism for the design of complex systems. It has an elegant object model, well integrated with the rest of the language, although we did not use object orientation in our application. It ought to be remarked that the mechanisms available for object oriented programming do not incur any penalty if one does not use them.

The total code written so far for our platform comprises 45 modules, totaling about 10000 lines of code. This whole system was developed solely by the author in about 6 months as a part-time activity. The programs were developed directly with the native code compiler, with only occasional use of bytecode execution and virtually no debugging: the whole development cycle consisted in iterating source edition, compilation until syntax errors and type inconsistencies were fixed, then native code execution. The dependency analysis tool of OCaml, coupled with the Unix make command, proved to be invaluable.

The only other software packages used for the development of our platform are the Emacs text editor (with its OCaml mode Tuareg) and the TEX-Metafont-LATEX 2_ε text formatter (with the devnag preprocessor[12] for displaying *devanāgarī* text under the Babel package). This demands an appropriate font server for the TrueType font format, a difficulty for old versions of X windows on UNIX/Linux platforms.

7.2 Web and other standards

For our Web tools we used the Web standard HTML 4.0[13] with Cascading Style Sheets in order to separate the formating instructions from the structuring constructs. We used UTF-8 encodings corresponding to the Unicode[23] standard 2.1, for displaying diacritics in the roman indic font[4].

A general remark concerning the W3C standards is that they are evolving too fast, and that interoperability across platforms is far from satisfactory. Since a large percentage of the Web users use browsers in the original version which was installed at the time of purchase of their platform, they will not be able to correctly interpret more recent versions of the various standards. Thus the designer of scripting tools has to

make various trade-offs between generating obsolete features (the so-called deprecated options in W3C jargon) which every browser will accept, and newer functionalities which may be unknown to older platforms. Even mature HTML features may have quite different look-and-feel under say Netscape Navigator and Internet Explorer browsers in their various versions. In this compromise we refrained from using ad-hoc browser scripting languages such as Javascript, mobile code programming languages such as Java, and ad-hoc features such as cookies, and we limited ourselves in the dynamic capabilities of our Web interface to remote procedure calls generating HTML pages (the so-called CGI-bins).

A related remark concerns XML[24], which is rapidly imposing itself as the standard data interchange format, especially in the area of linguistic and philology resources such as digital dictionaries and corpuses[22]. This means that XML versions of digital lexicons such as our sanskrit dictionay will have to be provided in the future in order to be usable in more generic linguistic platforms. Such documents will be compliant with precise DTDs explaining their logical structure, which will be themselves instances of generic standard formats. However, we are not convinced at all that the production of these documents, as well as computations performed on them, will need to use all the layers of resource description and other frameworks which are weekly churned out by the W3 Consortium and the various vendors. We rather believe that general programming languages such as OCaml are a more solid, more uniform, and more efficient alternative to such specific software engineering tools.

8 Conclusion

Reverse engineering is often referred to in the context of transferring some legacy system to a new programming language, system or platform, or of porting by hook or crook some obsolete database to an up-to-date format. Here we show an application where the reverse engineering operation leads to a translator (actually a family of translators and processing tools) which allows the development of a natural language processing platform centered on a lexical database whose format is only slightly modified. This means that the development of this platform may process asynchronously from the subsequent updates to the database, whose development may continue, without changing too much its input conventions. Furthermore, the reverse engineering operation may be seen as a continuous process, where more and more structure is progressively extracted from the data. For instance, comments in the original source may progressively find their way in the structure, as citation indexes for instance. This is possible because, in our approach, even though all computations are done on the abstract syntax tree representation, we still retain the original concrete syntax text file as the database source.

There is a general need for such adaptors in the field of computational linguistics, where resources such as lexicons and digitalised corpuses are long-term investments often developed by independent teams. Rather than attempting a complete standardization of their formats, an unending quest anyway in a world of fast-changing technologies, our approach permits parallel developments without changing the editing tools and acquisition processes. It is of upmost importance however that these translation processes be clean, well documented programs which may be maintained and adapted over long evolution cycles. We have shown that a general-purpose language such as Objective Caml, with its extensive meta-syntactic features, its powerful type discipline and module system, and its efficient and portable backend, is an ideal tool for such engineering tasks. We see little justification indeed in the use of less general and poorly efficient computing paradigms, such as the so-called "scripting languages".

References

[1] Guy Cousineau and Michel Mauny. "The Functional Approach to Programming". Cambridge University Press, 1998.

[2] Daniel de Rauglaudre and Michel Mauny. "Parsers in ML". Proceedings of Conference on LISP and Functional Programming (1992), pp. 76–85.

[3] Christiane Fellbaum, Ed. "Wordnet, an Electronic Lexical Database." MIT Press, 1998. See also: `http://www.cogsci.princeton.edu/~wn`.

[4] Christopher J. Fynn. "How to use Diacritics for Romanised Indic Text on the WWW". See: `http://www.user.dicon.co.uk/~cfynn/sanskrit.htm`.

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

[6] Gérard Huet. "Sanskrit to French dictionary". Available from the Sanskrit Web Site: http://pauillac.inria.fr/~huet/SKT/.

[7] Gérard Huet. "Structure of a Sanskrit Dictionary." INRIA Research Report to appear, 2001.

[8] Gérard Huet. "Computational Linguistics for Sanskrit: a Software Engineering Approach". To appear, anniversary volume in honor of Rod Burstall, 2001.

[9] Donald Knuth. The TEXbook. Addison-Wesley, 1984.

[10] Leslie Lamport. LATEX - A Document Preparation System. Addison-Wesley, 1986.

[11] Leslie Lamport et al. LATEX2ε - The macro package for TEX. Edition 1.6, Dec. 1994. Available from CTAN archive sites, such as ftp://ftp.tex.ac.uk/.

[12] Anshuman Pandey. Devanāgarī for TEX, version 1.6, Aug. 1998. Available from CTAN archive sites, such as ftp://ftp.tex.ac.uk/.

[13] Dave Raggett, Arnaud Le Hors and Ian Jacobs, Eds. HTML 4.0 Specification. W3C Recommendation, April 24, 1998. http://www.w3.org/TR/REC-html40.

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

[15] Acrobat Reader, an Adobe product, freely downloadable from: http://www.adobe.com/products/acrobat/readstep.html.

[16] The Camlp4 preprocessor. Available from: http://caml.inria.fr/camlp4/.

[17] The Common Gateway Interface. See: http://www.w3.org/CGI/Overview.html.

[18] Ghostscript. See: http://www.cs.wisc.edu/~ghost/index.html.

[19] Objective Caml. Available from: http://caml.inria.fr/ocaml/index.html.

[20] PDF, the portable document format, an Adobe product. See: http://www.adobe.com/products/acrobat/adobepdf.html.

[21] Postscript, an Adobe product. See: http://www.adobe.com/print/postscript/main.html.

[22] The Text Encoding Initiative. See: http://www.uic.edu/orgs/tei/.

[23] The Unicode standard 2.1. See: http://charts.unicode.org/.

[24] The XML norm, version 1.1, W3C consortium. See: http://www.w3c.org/documents/XML/.