The Sanskrit Platform Documentation
(Sanskrit Engine V3.23; Zen toolkit V4.1)

2020-05-04

Gérard Huet & Pawan Goyal

Abstract

This document is the commented code of the Sanskrit Engine software package. The programming language is Pidgin ML, implemented as Objective Caml (V4.07.1), under the revised syntax offered by the Camlp4 preprocessor. This documentation has been automatically generated by the Ocamlweb package of Jean-Christophe Filliâtre, using the LaTeX package, in the literate programming style pioneered by Don Knuth.

This program uses the Zen Computational Linguistics Toolkit. The present document is a companion volume to its documentation, available as `http://gallium.inria.fr/~huet/ZEN/zen.pdf` under the pdf format.

The Sanskrit Heritage site `http://sanskrit.inria.fr/` demonstrates various tools built with this package. An article describing Sanskrit segmentation and tagging based on this package is available as `http://gallium.inria.fr/~huet/PUBLIC/tagger.pdf` under the pdf format.

Module Control

Module Control contains exceptions of global scope

```
exception Anomaly of string (* deemed impossible by logic and ML semantics *)
;
exception Warning of string (* emits a warning *)
;
exception Fatal of string (* unrecoverable fatal error *)
;
(* error reporting *)
value report_mess = "- please report - "
;
value fatal_err_mess = "Fatal error "
```

```
and anomaly_err_mess = "Anomaly␣" ^ report_mess
and sys_err_mess = "System␣error␣" ^ report_mess
and stream_err_mess = "Stream␣error␣-␣wrong␣input␣?␣"
;
```

change if Morphology data type changes

```
value data_format_version = 1
;
```

Module Version

Generated by make version - see main Makefile

```
value version ="3.23" and version_date ="2020-05-04";
```

Module Date

Date utilities

```
value date_iso = Version.version_date (* "YYYY-MM-DD" *)
;
value version_id = Version.version
;
value version = "Version␣" ^ version_id ^ "␣[" ^ date_iso ^ "]"
;
value dico_date = (* for Sanskrit-French book form *)
  let year = String.sub date_iso 0 4
  and month = String.sub date_iso 5 2
  and day = String.sub date_iso 8 2 in
  (match int_of_string day with [ 1 → "1er" | n → string_of_int n ]) ^
  (match month with [ "01" → "␣Janvier␣"
                     | "02" → "␣F\\'evrier␣"
                     | "03" → "␣Mars␣"
                     | "04" → "␣Avril␣"
                     | "05" → "␣Mai␣"
                     | "06" → "␣Juin␣"
                     | "07" → "␣Juillet␣"
                     | "08" → "␣Ao\\^ut␣"
                     | "09" → "␣Septembre␣"
                     | "10" → "␣Octobre␣"
```

```
                    | "11" → "␣Novembre␣"
                    | "12" → "␣D\\'ecembre␣"
                    | _ → failwith "Invalid␣month␣code␣in␣date"
                    ]) ^ year
;
```

Module Canon

Inverse of *Transduction.code_raw* - word to VH transliteration
Except that .ll has no canonical code

```
value canon = fun
  [ 0 → "-" (* notation for suffixes and segmentation hint in compounds *)
  | 1 → "a"
  | 2 → "aa"
  | 3 → "i"
  | 4 → "ii"
  | 5 → "u"
  | 6 → "uu"
  | 7 → ".r"
  | 8 → ".rr"
  | 9 → ".l"
  | 10 → "e"
  | 11 → "ai"
  | 12 → "o"
  | 13 → "au"
  | 14 → ".m" (* anusvaara *)
  | 15 → "~~" (* anunāsika candrabindu *)
  | 16 → ".h"
  | 17 → "k"
  | 18 → "kh"
  | 19 → "g"
  | 20 → "gh"
  | 21 → "f" (* used to be "\"n" – fragile *)
  | 22 → "c"
  | 23 → "ch"
  | 24 → "j"
  | 25 → "jh"
  | 26 → "~n"
  | 27 → ".t"
```

```
| 28 → ".th"
| 29 → ".d"
| 30 → ".dh"
| 31 → ".n"
| 32 → "t"
| 33 → "th"
| 34 → "d"
| 35 → "dh"
| 36 → "n"
| 37 → "p"
| 38 → "ph"
| 39 → "b"
| 40 → "bh"
| 41 → "m"
| 42 → "y"
| 43 → "r"
| 44 → "l" (* Vedic l not accommodated *)
| 45 → "v"
| 46 → "z" (* used to be "\"s" – fragile *)
| 47 → ".s"
| 48 → "s"
| 49 → "h" (* h/.dh *)
| 50 → "_" (* hiatus *)
| −1 → "'" (* avagraha *)
| −2 → "[-]" (* amuissement - lopa of a or aa in preceding preverb *)
| −3 → "aa|a" (* sandhi of aa and a *a *)
| −4 → "aa|i" (* sandhi of aa and i *i *)
| −5 → "aa|u" (* sandhi of aa and u *u *)
| −6 → "aa|r" (* sandhi of aa and .r *r *)
| −7 → "aa|I" (* sandhi of aa and ii *I *)
| −8 → "aa|U" (* sandhi of aa and uu *U *)
| −9 → "aa|A" (* sandhi of aa and aa *A *)
| 123 → "aa|C" (* sandhi of aa and ch *C for ch gemination in cch *)
| 100 → "+" (* notation for segmentation hint *)
| 124 → failwith "Canon:␣Unrestored␣special␣phoneme␣j'" (* j/z *)
| 149 → failwith "Canon:␣Unrestored␣special␣phoneme␣h'" (* h/gh *)
| 249 → failwith "Canon:␣Unrestored␣special␣phoneme␣h''" (* h/dh *)
| n → if n < 0 ∨ n > 59 then failwith mess
          where mess = "Canon:␣Illegal␣char␣" ^ string_of_int n
      else "#" ^ Char.escaped (Char.chr (n − 2)) (* homo index 1 to 9 *)
```

```
                                       (* n-2 above since (ASCII) Char.chr 48 = '0' *)
  ]
;
(* Hiatus-conscious catenation b = True iff s starts with vowel *)
value catenate c (s, b) =
  let b' = c > 0 ∧ c < 14 (* Phonetics.vowel c *) in
  let protected = if b ∧ b' then "_" ^ s else s in
  (canon c ^ protected , b')
;
(* decode : word → string *)
value decode word =
  let (s, _) = List.fold_right catenate word ("",False) in s
;
value robust_decode c = (* used in Morpho_tex.print_inverse_map_txt *)
  let render n =
    try canon n with
    [ Failure _ → match n with
        [ 124 → "j'" | 149 → "h'" | 249 → "h''"
        | _ → string_of_int n
        ]
    ] in
  let conc s s' = render s ^ s' in
  List.fold_right conc c "" (* note no hiatus computation *)
;

value rdecode w = decode (Word.mirror w)
;
(*********************************************************************)
(* Important information for corpus processing *)
(*********************************************************************)
```

(∗ Beware. *decode* (*code_raw* *s*) is *s* with spaces removed but *code_raw* (*decode* *c*) may not be *c* because of VH ambiguities such as *decode* [1; 3] = *decode* [11] = "ai". Note that unsandhied text with spaces is wrongly parsed: *code_raw* "a␣i" = [11] and not [1; 50; 3]. Thus one should use underscore for hiatus in digitalised corpus: *code_raw* "a_i" = [1; 3]. The chunking of text by interpreting spaces is done in a preliminary pass by Sanskrit.padapatha. ∗)

Support for other translitteration schemes
Wax decoding - University of Hyderabad

```
value canon_WX = fun
```

```
[ 0  →  "-"
| 1  →  "a"
| 2  →  "A"
| 3  →  "i"
| 4  →  "I"
| 5  →  "u"
| 6  →  "U"
| 7  →  "q"
| 8  →  "Q"
| 9  →  "L"
| 10  →  "e"
| 11  →  "E"
| 12  →  "o"
| 13  →  "O"
| 14  →  "M"
| 15  →  "z"
| 16  →  "H"
| 17  →  "k"
| 18  →  "K"
| 19  →  "g"
| 20  →  "G"
| 21  →  "f"
| 22  →  "c"
| 23  →  "C"
| 24  →  "j"
| 25  →  "J"
| 26  →  "F"
| 27  →  "t"
| 28  →  "T"
| 29  →  "d"
| 30  →  "D"
| 31  →  "N"
| 32  →  "w"
| 33  →  "W"
| 34  →  "x"
| 35  →  "X"
| 36  →  "n"
| 37  →  "p"
| 38  →  "P"
| 39  →  "b"
```

```
  | 40 → "B"
  | 41 → "m"
  | 42 → "y"
  | 43 → "r"
  | 44 → "l" (* Vedic l not accommodated *)
  | 45 → "v"
  | 46 → "S"
  | 47 → "R"
  | 48 → "s"
  | 49 → "h"
  | 50 → "_" (* hiatus *)
  | −1 → "Z" (* avagraha *)
  | −2 → "[-]" (* amuissement - lopa of current aa- or preceding a- or aa- *)
  | −3 → "A|a" (* sandhi of aa and a *a *)
  | −4 → "A|i" (* sandhi of aa and i *e *)
  | −5 → "A|u" (* sandhi of aa and u *u *)
  | −6 → "A|r" (* sandhi of aa and .r *r *)
  | −7 → "aa|I" (* sandhi of aa and I *I *)
  | −8 → "aa|U" (* sandhi of aa and U *U *)
  | −9 → "aa|A" (* sandhi of aa and A *A *)
  | 123 → "aa|C" (* sandhi of aa and C *C for duplication *)
  | 100 → "+" (* explicit compound with no sandhi - experimental *)
  | n → if n < 0 ∨ n > 59 then failwith mess
            where mess = "Canon:␣Illegal␣char␣" ^ string_of_int n
         else "#" ^ Char.escaped (Char.chr (n − 2)) (* homo index 1 to 9 *)
  ]
;
value decode_WX word =
  List.fold_right (fun c s → (canon_WX c) ^ s) word ""
;
(* Sanskrit Library SLP1 decoding *)
value canon_SL = fun
  [ 0 → "-"
  | 100 → "+"
  | 1 → "a"
  | 2 → "A"
  | 3 → "i"
  | 4 → "I"
  | 5 → "u"
  | 6 → "U"
```

```
| 7 → "f"
| 8 → "F"
| 9 → "x"
| 10 → "e"
| 11 → "E"
| 12 → "o"
| 13 → "O"
| 14 → "M"
| 15 → "~"
| 16 → "H"
| 17 → "k"
| 18 → "K"
| 19 → "g"
| 20 → "G"
| 21 → "N"
| 22 → "c"
| 23 → "C"
| 24 → "j"
| 25 → "J"
| 26 → "Y"
| 27 → "w"
| 28 → "W"
| 29 → "q"
| 30 → "Q"
| 31 → "R"
| 32 → "t"
| 33 → "T"
| 34 → "d"
| 35 → "D"
| 36 → "n"
| 37 → "p"
| 38 → "P"
| 39 → "b"
| 40 → "B"
| 41 → "m"
| 42 → "y"
| 43 → "r"
| 44 → "l" (* Vedic l not accommodated *)
| 45 → "v"
| 46 → "S"
```

```
  | 47 → "z"
  | 48 → "s"
  | 49 → "h"
  | 50 → "_" (* hiatus *)
  | − 1 → "Z" (* avagraha *)
  | n → if n < 0 ∨ n > 59 then failwith mess
            where mess = "Canon:␣Illegal␣char␣" ^ string_of_int n
        else "#" ^ Char.escaped (Char.chr (n − 2)) (* homo index 1 to 9 *)
  ]
;
value decode_SL word =
  List.fold_right (fun c s → (canon_SL c) ^ s) word ""
;
(* Kyoto-Harvard decoding *)
value canon_KH = fun
  [ 0 → "-"
  | 100 → "+"
  | 1 → "a"
  | 2 → "A"
  | 3 → "i"
  | 4 → "I"
  | 5 → "u"
  | 6 → "U"
  | 7 → "R"
  | 8 → "RR"
  | 9 → "L"
  | 10 → "e"
  | 11 → "ai"
  | 12 → "o"
  | 13 → "au"
  | 14 → "M"
  | 15 → "M" (* candrabindu absent *)
  | 16 → "H"
  | 17 → "k"
  | 18 → "kh"
  | 19 → "g"
  | 20 → "gh"
  | 21 → "G"
  | 22 → "c"
  | 23 → "ch"
```

```
  | 24 → "j"
  | 25 → "jh"
  | 26 → "J"
  | 27 → "T"
  | 28 → "Th"
  | 29 → "D"
  | 30 → "Dh"
  | 31 → "N"
  | 32 → "t"
  | 33 → "th"
  | 34 → "d"
  | 35 → "dh"
  | 36 → "n"
  | 37 → "p"
  | 38 → "ph"
  | 39 → "b"
  | 40 → "bh"
  | 41 → "m"
  | 42 → "y"
  | 43 → "r"
  | 44 → "l" (* Vedic l not accommodated *)
  | 45 → "v"
  | 46 → "z"
  | 47 → "S"
  | 48 → "s"
  | 49 → "h"
  | 50 → "_" (* hiatus *)
  | -1 → "'" (* avagraha *)
  | n → if n < 0 ∨ n > 59 then failwith mess
              where mess = "Canon:␣Illegal␣char␣" ^ string_of_int n
          else "#" ^ Char.escaped (Char.chr (n - 2)) (* homo index 1 to 9 *)
  ]
;
value decode_KH word =
  List.fold_right (fun c s → (canon_KH c) ^ s) word ""
;
value switch_decode = fun (* normalizes anusvaara in its input *)
  [ "VH" → decode
  | "WX" → decode_WX
  | "KH" → decode_KH
```

```
  | "SL" → decode_SL
  | _ → failwith "Unexpected␣transliteration␣scheme"
  ]
;
(* Decoding without double quotes *)
value canon2 = fun
  [ 0 → "-"
  | 100 → "+"
  | 1 → "a"
  | 2 → "A"
  | 3 → "i"
  | 4 → "I"
  | 5 → "u"
  | 6 → "U"
  | 7 → ".r"
  | 8 → ".R"
  | 9 → ".l"
  | 10 → "e"
  | 11 → "E"
  | 12 → "o"
  | 13 → "O"
  | 14 → ".m"
  | 15 → "M"
  | 16 → ".h"
  | 17 → "k"
  | 18 → "K"
  | 19 → "g"
  | 20 → "G"
  | 21 → "N"
  | 22 → "c"
  | 23 → "C"
  | 24 → "j"
  | 25 → "J"
  | 26 → "~n"
  | 27 → ".t"
  | 28 → ".T"
  | 29 → ".d"
  | 30 → ".D"
  | 31 → ".n"
  | 32 → "t"
```

```
  | 33 → "T"
  | 34 → "d"
  | 35 → "D"
  | 36 → "n"
  | 37 → "p"
  | 38 → "P"
  | 39 → "b"
  | 40 → "B"
  | 41 → "m"
  | 42 → "y"
  | 43 → "r"
  | 44 → "l"
  | 45 → "v"
  | 46 → "z"
  | 47 → ".s"
  | 48 → "s"
  | 49 → "h"
  | 50 → "_" (* hiatus *)
  | − 1 → "'"
  | − 2 → "[-]"
  | − 3 → "aa|a" (* sandhi of A and a *a *)
  | − 4 → "aa|i" (* sandhi of A and i *i *)
  | − 5 → "aa|u" (* sandhi of A and u *u *)
  | − 6 → "aa|r" (* sandhi of A and .r *r *)
  | − 7 → "aa|I" (* sandhi of aa and I *I *)
  | − 8 → "aa|U" (* sandhi of aa and U *U *)
  | − 9 → "aa|A" (* sandhi of aa and A *A *)
  | 123 → "aa|C" (* sandhi of aa and C *C *)
  | n → if n < 0 ∨ n > 59 then failwith ("canon2: " ^ string_of_int n)
          else ("#" ^ Char.escaped (Char.chr (n − 2)))
  ]
;
(* hiatus-conscious catenation b = True iff s starts with vowel *)
value catenate2 c (s, b) =
  let b′ = c > 0 ∧ c < 14 (* Phonetics.vowel c *) in
  let protected = if b ∧ b′ then "_" ^ s else s in
  (canon2 c ^ protected , b′)
;
(* decode2 : word -¿ string (debug for Morpho_xml *)
value decode2 word =
```

```
  try let (s, _) = List.fold_right catenate2 word ("",False) in s
  with [ Failure _ → failwith ("decode2: " ^ robust_decode (Word.mirror word)) ]
;
value canon_upper = fun
  [ 101 → "A"
  | 102 → "AA"
  | 103 → "I"
  | 104 → "II"
  | 105 → "U"
  | 106 → "UU"
  | 107 → ".R"
  | 110 → "E"
  | 111 → "Ai"
  | 112 → "O"
  | 113 → "Au"
  | 117 → "K"
  | 118 → "Kh"
  | 119 → "G"
  | 120 → "Gh"
  | 122 → "C"
  | 123 → "Ch"
  | 124 → "J"
  | 125 → "Jh"
  | 127 → ".T"
  | 128 → ".Th"
  | 129 → ".D"
  | 130 → ".Dh"
  | 132 → "T"
  | 133 → "Th"
  | 134 → "D"
  | 135 → "Dh"
  | 136 → "N"
  | 137 → "P"
  | 138 → "Ph"
  | 139 → "B"
  | 140 → "Bh"
  | 141 → "M"
  | 142 → "Y"
  | 143 → "R"
  | 144 → "L"
```

```
  | 145 → "V"
  | 146 → "Z"
  | 147 → ".S"
  | 148 → "S"
  | 149 → "H"
  | n → failwith ("Illegal upper case code : " ^ string_of_int n)
  ]
;
(* decode_ref : word → string *)
value decode_ref word =
  let canon c = if c > 100 then canon_upper c else canon c in
  let canon_catenate c (s, b) =
       let b' = c > 0 ∧ c < 14 (* Phonetics.vowel c *) in
       let protected = if b ∧ b' then "_" ^ s else s in
       (canon c ^ protected , b') in
  let (s, _) = List.fold_right canon_catenate word ("",False) in s
;
value canon_html = fun
  [ 0 → "-"
  | 100 → "+"
  | 1 → "a"
  | 2 → "aa"
  | 3 → "i"
  | 4 → "ii"
  | 5 → "u"
  | 6 → "uu"
  | 7 → ".r"
  | 8 → ".rr"
  | 9 → ".l"
  | 10 → "e"
  | 11 → "ai"
  | 12 → "o"
  | 13 → "au"
  | 14 → ".m"
  | 15 → "~"
  | 16 → ".h"
  | 17 → "k"
  | 18 → "kh"
  | 19 → "g"
  | 20 → "gh"
```

```
  | 21 → "f"
  | 22 → "c"
  | 23 → "ch"
  | 24 → "j"
  | 25 → "jh"
  | 26 → "~n"
  | 27 → ".t"
  | 28 → ".th"
  | 29 → ".d"
  | 30 → ".dh"
  | 31 → ".n"
  | 32 → "t"
  | 33 → "th"
  | 34 → "d"
  | 35 → "dh"
  | 36 → "n"
  | 37 → "p"
  | 38 → "ph"
  | 39 → "b"
  | 40 → "bh"
  | 41 → "m"
  | 42 → "y"
  | 43 → "r"
  | 44 → "l"
  | 45 → "v"
  | 46 → "z"
  | 47 → ".s"
  | 48 → "s"
  | 49 → "h"
  | 50 → "_" (* hiatus *)
  | n → if n < 0 then
              failwith ("Illegal letter to canon_html : " ^ string_of_int n)
          else ("#" ^ Char.escaped (Char.chr (n - 2)))
  ]
;
value canon_upper_html = fun
  [ 101 → "Ua"
  | 102 → "Uaa"
  | 103 → "Ui"
  | 104 → "Uii"
```

```
  | 105 → "Uu"
  | 106 → "Uuu"
  | 107 → "U.r"
  | 110 → "Ue"
  | 111 → "Uai"
  | 112 → "Uo"
  | 113 → "Uau"
  | 117 → "Uk"
  | 118 → "Ukh"
  | 119 → "Ug"
  | 120 → "Ugh"
  | 122 → "Uc"
  | 123 → "Uch"
  | 124 → "Uj"
  | 125 → "Ujh"
  | 127 → "U.t"
  | 128 → "U.th"
  | 129 → "U.d"
  | 130 → "U.dh"
  | 132 → "Ut"
  | 133 → "Uth"
  | 134 → "Ud"
  | 135 → "Udh"
  | 136 → "Un"
  | 137 → "Up"
  | 138 → "Uph"
  | 139 → "Ub"
  | 140 → "Ubh"
  | 141 → "Um"
  | 142 → "Uy"
  | 143 → "Ur"
  | 144 → "Ul"
  | 145 → "Uv"
  | 146 → "Uz"
  | 147 → "U.s"
  | 148 → "Us"
  | 149 → "Uh"
  | n → failwith ("Illegal␣upper␣case␣code␣:␣" ^ string_of_int n)
  ]
;
```

```
(* Roman with diacritics Unicode - latin extended *)
value canon_uniromcode = fun
  [ 0 → "-"
  | 100 → "+"
  | 1 → "a"
  | 2 → "&#257;"
  | 3 → "i"
  | 4 → "&#299;"
  | 5 → "u"
  | 6 → "&#363;"
  | 7 → "&#7771;"
  | 8 → "&#7773;"
  | 9 → "&#7735;"
  | 10 → "e"
  | 11 → "ai"
  | 12 → "o"
  | 13 → "au"
  | 14 → "&#7747;" (* anusvaara as m with dot below *)
  | 15 → "&#7745;" (* candrabindu as m with dot above (?) *)
  | 16 → "&#7717;"
  | 17 → "k"
  | 18 → "kh"
  | 19 → "g"
  | 20 → "gh"
  | 21 → "&#7749;"
  | 22 → "c"
  | 23 → "ch"
  | 24 → "j"
  | 25 → "jh"
  | 26 → "&#241;"
  | 27 → "&#7789;"
  | 28 → "&#7789;h"
  | 29 → "&#7693;"
  | 30 → "&#7693;h"
  | 31 → "&#7751;"
  | 32 → "t"
  | 33 → "th"
  | 34 → "d"
  | 35 → "dh"
  | 36 → "n"
```

```
  | 37 → "p"
  | 38 → "ph"
  | 39 → "b"
  | 40 → "bh"
  | 41 → "m"
  | 42 → "y"
  | 43 → "r"
  | 44 → "l"
  | 45 → "v"
  | 46 → "&#347;"
  | 47 → "&#7779;"
  | 48 → "s"
  | 49 → "h"
  | 50 → "_"
  | − 1 → "'"
  | − 2 → "[-]" (* amuissement - lopa of current aa- or preceding a- or aa- *)
  | − 3 → "&#257;|a" (* sandhi of aa and (a,aa) *a *)
  | − 4 → "&#257;|i" (* sandhi of aa and (i,ii) *e *)
  | − 5 → "&#257;|u" (* sandhi of aa and (u,uu) *u *)
  | − 6 → "&#257;|r" (* sandhi of aa and .r *r *)
  | 124 → failwith "Canon:␣Unrestored␣special␣phoneme␣j'"
  | 149 → failwith "Canon:␣Unrestored␣special␣phoneme␣h'"
  | 249 → failwith "Canon:␣Unrestored␣special␣phoneme␣h''"
  | n → if n < 0 then
            failwith ("Illegal␣code␣to␣canon_unicode␣:␣" ^ string_of_int n)
          else ("_" ^ Char.escaped (Char.chr (n − 2)))
  ]
;
(* Gives the Unicode representation of the romanisation of word *)
(* unicode : word → string *)
value uniromcode word =
  let catenate c (s, b) =
      let b′ = c > 0 ∧ c < 14 (* Phonetics.vowel c *) in
      let protected = if b ∧ b′ then "␣" ^ s else s in
      (canon_uniromcode c ^ protected , b′) in
  let (s, _) = List.fold_right catenate word ("",False) in s
;
value halant = "&#x094D;"
(* and avagraha = "&#x093D;" and candrabindu = "&#x310;" *)
(* Numerals to come: 1="x0967;" ... 9="x0966F" *)
```

```
;
(* represents a stem word in romanization or VH transliteration *)
value stem_to_string html =
  if html then uniromcode (* UTF8 romanization with diacritics *)
          else decode (* VH *)
;
exception Hiatus
;
value indic_unicode_point = fun
  [ 0 | 100 → (* - *) "70"
  | 1 → (* a *) "05"
  | 2 → (* aa *) "06"
  | 3 → (* i *) "07"
  | 4 → (* ii *) "08"
  | 5 → (* u *) "09"
  | 6 → (* uu *) "0A"
  | 7 → (* .r *) "0B"
  | 8 → (* .rr *)"60"
  | 9 → (* .l *) "0C"
  | 10 → (* e *) "0F"
  | 11 → (* ai *) "10"
  | 12 → (* o *) "13"
  | 13 → (* au *) "14"
  | 14 → (* .m *) "02"
  | 15 → (*   *) "01"
  | 16 → (* .h *) "03"
  | 17 → (* k *) "15"
  | 18 → (* kh *) "16"
  | 19 → (* g *) "17"
  | 20 → (* gh *) "18"
  | 21 → (* 'n *) "19"
  | 22 → (* c *) "1A"
  | 23 → (* ch *) "1B"
  | 24 → (* j *) "1C"
  | 25 → (* jh *) "1D"
  | 26 → (*  n *) "1E"
  | 27 → (* .t *) "1F"
  | 28 → (* .th *)"20"
  | 29 → (* .d *) "21"
  | 30 → (* .dh *)"22"
```

```
  | 31 → (* .n *) "23"
  | 32 → (* t *) "24"
  | 33 → (* th *) "25"
  | 34 → (* d *) "26"
  | 35 → (* dh *) "27"
  | 36 → (* n *) "28"
  | 37 → (* p *) "2A"
  | 38 → (* ph *) "2B"
  | 39 → (* b *) "2C"
  | 40 → (* bh *) "2D"
  | 41 → (* m *) "2E"
  | 42 → (* y *) "2F"
  | 43 → (* r *) "30"
  | 44 → (* l *) "32"
  | 45 → (* v *) "35"
  | 46 → (* z *) "36"
  | 47 → (* .s *) "37"
  | 48 → (* s *) "38"
  | 49 → (* h *) "39"
  | 50 → (* underscore *) raise Hiatus
  | −1 → (* avagraha *) "3D"
  | −2 → "" (* amuissement *)
  | −3 → "06" (* "aa|a" sandhi of aa and (a,aa) *)
  | −4 → "0F" (* "aa|i" sandhi of aa and (i,ii) *)
  | −5 → "13" (* "aa|u" sandhi of aa and (u,uu) *)
  | −6 → "06" (* sandhi of aa and .r *)
  | c → if c < 0 ∨ c > 59
           then failwith ("Illegal␣code␣to␣dev_unicode:␣" ^ string_of_int c)
        else "" (* homo index dropped *)
  ]
and matra_indic_unicode_point = fun
  [ 100 (* + *) (* necessary for word form ending in consonant *)
  | 0 → (* - *) "70" (* id for iics *)
  | 1 → (* a *) "" (* default *)
  | 2 → (* aa *) "3E"
  | 3 → (* i *) "3F"
  | 4 → (* ii *) "40"
  | 5 → (* u *) "41"
  | 6 → (* uu *) "42"
  | 7 → (* .r *) "43"
```

```
  | 8  → (* .rr *)"44"
  | 9  → (* .l *) "62"
  | 10  → (* e *) "47"
  | 11  → (* ai *) "48"
  | 12  → (* o *) "4B"
  | 13  → (* au *) "4C"
  | 15  → (*    *) "01"
  | c  → failwith ("Illegal␣code␣to␣matra_unicode␣:␣" ^ string_of_int c)
  ]
;
(* om 50 udatta 51 anudatta 52 grave 53 acute 54 avagraha 3D .ll 61 danda 64 ddanda 65
0 66 1 67 2 68 3 69 4 6A 5 6B 6 6C 7 6D 8 6E 9 6F deg 70 *)
value inject_point s = "&#x09" ^ s ^ ";"
;
value deva_unicode c =
  let s = indic_unicode_point c in inject_point s
and matra_unicode c =
  if c = 1 then "" (* default *)
  else let s = matra_indic_unicode_point c in inject_point s
;
(* Gives the Unicode representation of devanagari form of word; *)
(* ligature construction is left to the font manager handling of halant. *)
(* Beware : word should not carry homophony index - use code_strip. *)
(* unidevcode : word → string *)
value unidevcode word =
  let ligature (s, b) c = (* b memorizes whether last char is consonant *)
      try let code = deva_unicode c in
           if c > 16 (* Phonetics.consonant c *) then
               if b (* add glyph *) then (s ^ halant ^ code, True)
               else (s ^ code, True)
           else if b then
                  if c = 0 (* - *) then (s ^ halant ^ code, False)
                  else (* add matra *) let m = matra_unicode c in (s ^ m, False)
           else (s ^ code, False)
      with (* hiatus represented by space in devanagarii output *)
             [ Hiatus → (s ^ "␣",False) ] in
  let (s, b) = List.fold_left ligature ("",False) word in
  if b then s ^ halant (* virama *) else s
;
```

Module Min_lexer

A very simple lexer recognizing 1 character idents and integers and skipping spaces and comments between % and eol; used for various transduction tasks with Camlp4 Grammars. It is a copy of *ZEN/zen_lexer.ml* in order to simplify dependencies.

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

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

The string buffering machinery.

```
  value store buf c = do { Buffer.add_char buf c; buf }
  ;
  value rec number buf =
    parser
    [ [: `('0'..'9' as c); s :] → number (store buf c) s
    | [: :] → Buffer.contents buf
    ]
  ;
  value rec skip_to_eol =
    parser
```

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

Module Transduction

```
open Camlp4.PreCast; (* MakeGram Loc *)
module Gram = MakeGram Min_lexer
;
open Min_lexer.Token
;
value transducer trad t =
  try Gram.parse_string trad Loc.ghost t with
```

```
  [ Loc.Exc_located loc e -> do
      { Format.eprintf "In␣string␣\"%s\",␣at␣location␣%a:@." t Loc.print loc
      ; raise e
      }
  ]
;
```

Roman with diacritics, TeX encoding

```
value tex = Gram.Entry.mk "skt␣to␣tex"
and tex_word = Gram.Entry.mk "skt␣to␣tex␣word"
;
EXTEND Gram (* skt to tex *)
  tex :
    [ [ "\""; LETTER "n" -> "\\.n" (* deprecated *)
      | LETTER "f" -> "\\.n" (* recommended *)
      | LETTER "F" -> "f" (* patch for latin *)
      | "\""; LETTER "s" -> "\\'s" (* deprecated *)
      | LETTER "z" -> "\\'s" (* recommended *)
      | "\""; LETTER "S" -> "\\'S"
      | LETTER "Z" -> "\\'S"
      | "'"; LETTER "a" -> "\\'a"
      | "'"; LETTER "i" -> "{\\'\\i}"
      | "'"; LETTER "u" -> "\\'u"
      | "'"; LETTER "e" -> "\\'e"
      | "'"; LETTER "o" -> "\\'o"
      | LETTER "a"; LETTER "a"; "|"; LETTER "i" -> failwith "Unexpected␣phantom␣phoneme"
      | LETTER "a"; LETTER "a"; "|"; LETTER "u" -> failwith "Unexpected␣phantom␣phoneme"
      | LETTER "a"; LETTER "a"; "|"; LETTER "a" -> failwith "Unexpected␣phantom␣phoneme"
      | LETTER "a"; LETTER "a" -> "\\=a"
      | LETTER "a" -> "a"
      | LETTER "A"; LETTER "A" -> "\\=A"
      | LETTER "A" -> "A"
      | LETTER "i"; LETTER "i" -> "{\\=\\i}"
      | LETTER "i" -> "i"
      | LETTER "I"; LETTER "I" -> "\\=I"
      | LETTER "I" -> "I"
      | LETTER "u"; LETTER "u" -> "\\=u"
      | LETTER "u" -> "u"
      | LETTER "U"; LETTER "U" -> "\\=U"
      | LETTER "U" -> "U"
```

```
      | "~"; LETTER "n" → "\\~n"
      | LETTER "l"; "~"; "~" → "\\~l" (* candrabindu *)
      | LETTER "y"; "~"; "~" → "\\~y" (* candrabindu *)
      | LETTER "v"; "~"; "~" → "\\~v" (* candrabindu *)
      | "+" → "\\-" (* hyphenation hint *)
      | "$" → "\\_" (* pra-uga *)
      | "_" → "\\_" (* hiatus *)
      | "&" → "\\&" (* reserved *)
      | "-" → "-" (* prefix *)
      | "'" → "'" (* avagraha *)
      | "."; "."; "." → "..." (* ... *)
      | "."; LETTER "t" → "{\\d␣t}"
      | "."; LETTER "d" → "{\\d␣d}"
      | "."; LETTER "s" → "{\\d␣s}"
      | "."; LETTER "S" → "{\\d␣S}"
      | "."; LETTER "n" → "{\\d␣n}"
      | "."; LETTER "r"; LETTER "r" → "{\\RR}"
      | "."; LETTER "r" → "{\\d␣r}"
      | "."; LETTER "R" → "{\\d␣R}"
      | "."; LETTER "l"; LETTER "l" → "{\\LL}"
      | "."; LETTER "l" → "{\\d␣l}"
      | "."; LETTER "m" → "{\\d␣m}"
      | "."; LETTER "h" → "{\\d␣h}"
      | "."; LETTER "T" → "{\\d␣T}"
      | "."; LETTER "D" → "{\\d␣D}"
      | "#"; i = INT → "\\(_{" ^ i ^ "}\\)" (* homonyms *)
      | i = LETTER → i
      | i = INT → i
    ] ];
  tex_word :
    [ [ w = LIST0 tex; `EOI → String.concat "" w ] ];
END
;
value skt_to_tex = transducer tex_word
;
(****************************************************************)
(* Roman with diacritics, HTML decimal encoding for Unicode points *)
(****************************************************************)
value html_code = Gram.Entry.mk "skt␣to␣html␣code"
and html = Gram.Entry.mk "skt␣to␣html"
```

```
;
EXTEND Gram (* skt to HTML string *)
  html_code :
    [ [ "\""; LETTER "n" → "&#7749;"
      | LETTER "f" → "&#7749;"
      | LETTER "F" → "f" (* patch for latin *)
      | "\""; LETTER "s" → "&#347;"
      | LETTER "z" → "&#347;"
      | "\""; LETTER "S" → "&#346;"
      | LETTER "Z" → "&#346;"
      | "\""; LETTER "m" → "&#7745;" (* candrabindu as m with dot above *)
      | "'"; LETTER "a" → "a" (* we lose accents *)
      | "'"; LETTER "i" → "i"
      | "'"; LETTER "u" → "u"
      | "'"; LETTER "e" → "e"
      | "'"; LETTER "o" → "o"
      | LETTER "a"; LETTER "a" → "&#257;"
      | LETTER "a" → "a"
      | LETTER "A"; LETTER "A" → "&#256;"
      | LETTER "A" → "A"
      | LETTER "i"; LETTER "i" → "&#299;"
      | LETTER "i" → "i"
      | LETTER "I"; LETTER "I" → "&#298;"
      | LETTER "I" → "I"
      | LETTER "u"; LETTER "u" → "&#363;"
      | LETTER "u" → "u"
      | LETTER "U"; LETTER "U" → "&#362;"
      | LETTER "U" → "U"
      | "~"; LETTER "n" → "&#241;"
      | "~"; "~" → "&#7745;" (* candrabindu *)
      | "+" → "" (* "\&#173;" = &shy; cesure prints - *)
      | "$" → "_" (* pra-uga *)
      | "_" → "_" (* hiatus *)
      | "-" → "-" (* prefix *)
      | "&" → "&amp;" (* reserved *)
      | "'" → "'" (* avagraha *)
      | "."; "."; "." → "..." (* ... *)
      | "."; LETTER "t" → "&#7789;"
      | "."; LETTER "d" → "&#7693;"
      | "."; LETTER "s" → "&#7779;"
```

```
      | "."; LETTER "S" → "&#7778;"
      | "."; LETTER "n" → "&#7751;"
      | "."; LETTER "r"; LETTER "r" → "&#7773;"
      | "."; LETTER "r" → "&#7771;"
      | "."; LETTER "R" → "&#7770;"
      | "."; LETTER "l"; LETTER "l" → "&#7737;"
      | "."; LETTER "l" → "&#7735;"
      | "."; LETTER "m" → "&#7747;"
      | "."; LETTER "h" → "&#7717;"
      | "."; LETTER "T" → "&#7788;"
      | "."; LETTER "D" → "&#7692;"
      | "#"; i = INT → "_" ^ i (* homonymy index *)
      | "|"; LETTER "a" → "|a" (* phantom phoneme *a *)
      | "|"; LETTER "i" → "|i" (* phantom phoneme *i *)
      | "|"; LETTER "u" → "|u" (* phantom phoneme *u *)
      | "|"; LETTER "r" → "|&#7771;" (* phantom phoneme *r *)
      | "|"; LETTER "C" → "|ch" (* phantom phoneme *C *)
      | "["; "-"; "]" → "[-]" (* amuissement *)
      | i = LETTER → i
      | i = INT → i
    ] ];
  html :
    [ [ w = LIST0 html_code; `EOI → String.concat "" w ] ];
END
;
value skt_to_html = transducer html
;
```

Inverse to *Cgi.decode_url*

```
value url_letter = Gram.Entry.mk "skt␣to␣url␣letter"
and url = Gram.Entry.mk "skt␣to␣url"
;
(* Important: accents and avagraha are removed from the input stream *)
(* Should be isomorphic to code_rawu *)
EXTEND Gram (* skt to url *)
  url_letter :
    [ [ "\"" → "%22"
      | "~" → "%7E"
      | "#"; i = INT → "%23" ^ i
      | "'" → "" (* accents and avagraha hidden *)
```

```
(* — "'" → "%27" (* if preserved *) *)
        | "." → "."
        | "+" → "" (* "%2B" *)
        | "-" → "-"
        | "␣" → "+"
        | "_" → "_"
        | "$" → "%24"
        | i = LETTER → i
      ] ];
   url :
      [ [ w = LIST0 url_letter; `EOI → String.concat "" w ] ];
END
;
value encode_url = transducer url
;
(*********************************************)
(* Devanagari in Velthuis devnag transliteration *)
(*********************************************)
value dev = Gram.Entry.mk "dev␣symbol"
and dev_word = Gram.Entry.mk "dev␣word"
;
EXTEND Gram (* skt to devnag *)
   dev :
      [ [ "\""; LETTER "n" → "\"n"
        | LETTER "f" → "\"n"
        | "\""; LETTER "m" → "/" (* candrabindu *)
        | "\""; LETTER "s" → "\"s"
        | LETTER "z" → "\"s"
        | "'"; LETTER "a" → "a"
        | "'"; LETTER "i" → "i"
        | "'"; LETTER "u" → "u"
        | "'"; LETTER "e" → "e"
        | "'"; LETTER "o" → "o"
        | LETTER "a"; LETTER "a" → "aa"
        | LETTER "a" → "a"
        | LETTER "i"; LETTER "i" → "ii"
        | LETTER "i" → "i"
        | LETTER "u"; LETTER "u" → "uu"
        | LETTER "u" → "u"
        | "~"; LETTER "n" → "~n"
```

```
      | "~"; "~" → "/" (* candrabindu *)
      | "+" → ""
      | "$" → "$$" (* hiatus *) (* "{}" in devnag 1.6 *)
      | "-" → "@" (* suffix *)
      | "'" → ".a" (* avagraha *)
      | "."; LETTER "t" → ".t"
      | "."; LETTER "d" → ".d"
      | "."; LETTER "s" → ".s"
      | "."; LETTER "n" → ".n"
      | "."; LETTER "r"; LETTER "r" → ".R"
      | "."; LETTER "r" → ".r"
      | "."; LETTER "l" → ".l"
      | "."; LETTER "m" → ".m"
      | "."; LETTER "h" → ".h"
      | "#"; INT → "" (* homo index ignored *)
      | i = LETTER → i
    ] ];
  dev_word :
    [ [ w = LIST0 dev; `EOI → String.concat "" w ] ];
END
;
value skt_to_dev = transducer dev_word
;
(*************************************)
(* Greek and math symbols, TeX encoding *)
(*************************************)
value texmath = Gram.Entry.mk "math in tex"
and texmath_word = Gram.Entry.mk "math in tex word"
;
EXTEND Gram (* Greek and Math to TeX *)
  texmath :
    [ [ LETTER "a" → "\\alpha"
      | LETTER "b" → "\\beta"
      | LETTER "c" → "\\gamma"
      | LETTER "C" → "\\Gamma"
      | LETTER "d" → "\\delta"
      | LETTER "D" → "\\Delta"
      | LETTER "e" → "\\epsilon"
      | LETTER "f" → "\\phi"
      | LETTER "F" → "\\Phi"
```

```
| LETTER "g" → "\\psi"
| LETTER "G" → "\\Psi"
| LETTER "h" → "\\theta"
| LETTER "H" → "\\Theta"
| LETTER "i" → "\\iota"
| LETTER "k" → "\\kappa"
| LETTER "K" → "{\\rm␣K}"
| LETTER "l" → "\\lambda"
| LETTER "L" → "\\Lambda"
| LETTER "m" → "\\mu"
| LETTER "n" → "\\nu"
| LETTER "o" → "␣o"
| LETTER "O" → "{\\rm␣O}"
| LETTER "p" → "\\pi"
| LETTER "P" → "\\Pi"
| LETTER "q" → "\\chi"
| LETTER "r" → "\\rho"
| LETTER "s" → "\\sigma"
| LETTER "S" → "\\Sigma"
| LETTER "t" → "\\tau"
| LETTER "u" → "\\upsilon"
| LETTER "U" → "\\Upsilon"
| LETTER "v" → "\\varsigma"
| LETTER "w" → "\\omega"
| LETTER "W" → "\\Omega"
| LETTER "x" → "\\xi"
| LETTER "X" → "\\Xi"
| LETTER "y" → "\\eta"
| LETTER "z" → "\\zeta"
| LETTER "Z" → "{\\rm␣Z}"
| "*" → "{\\times}"
| "+" → "+"
| "@" → "{}^{\\circ}" (* degree *)
| "'" → "'"
| "|" → "{\\mid}"
| "!" → "\\!"
| "~" → "{\\sim}"
| "=" → "="
| "," → ",␣"
| i = INT → i
```

```
    ] ];
  texmath_word :
    [ [ w = LIST0 texmath; `EOI -> String.concat "" w ] ];
END
;
value math_to_tex = transducer texmath_word
;
(*****************************************)
(* Greek and math symbols, HTML encoding *)
(*****************************************)
value htmlmath = Gram.Entry.mk "math in html"
and htmlmath_word = Gram.Entry.mk "math in html word"
;
EXTEND Gram (* greek and math to html *)
  htmlmath :
    [ [ LETTER "a" -> "&#945;" (* "\&alpha;" *)
      | LETTER "b" -> "&#946;" (* "\&beta;" *)
      | LETTER "c" -> "&#947;" (* "\&gamma;" *)
      | LETTER "C" -> "&#915;" (* "\&Gamma;" *)
      | LETTER "d" -> "&#948;" (* "\&delta;" *)
      | LETTER "D" -> "&#916;" (* "\&Delta;" *)
      | LETTER "e" -> "&#949;" (* "\&epsilon;" *)
      | LETTER "f" -> "&#966;" (* "\&phi;" *)
      | LETTER "F" -> "&#934;" (* "\&Phi;" *)
      | LETTER "g" -> "&#968;" (* "\&psi;" *)
      | LETTER "G" -> "&#936;" (* "\&Psi;" *)
      | LETTER "h" -> "&#952;" (* "\&theta;" *)
      | LETTER "H" -> "&#920;" (* "\&Theta;" *)
      | LETTER "i" -> "&#953;" (* "\&iota;" *)
      | LETTER "k" -> "&#954;" (* "\&kappa;" *)
      | LETTER "K" -> "&#922;" (* "\&Kappa;" *)
      | LETTER "l" -> "&#955;" (* "\&lambda;" *)
      | LETTER "L" -> "&#923;" (* "\&Lambda;" *)
      | LETTER "m" -> "&#956;" (* "\&mu;" *)
      | LETTER "n" -> "&#957;" (* "\&nu;" *)
      | LETTER "o" -> "&#959;" (* "\&omicron;" *)
      | LETTER "O" -> "&#927;" (* "\&Omicron;" *)
      | LETTER "p" -> "&#960;" (* "\&pi;" *)
      | LETTER "P" -> "&#960;" (* "\&Pi;" *)
      | LETTER "q" -> "&#967;" (* "\&chi;" *)
```

```
      | LETTER "r" → "&#961;" (* "\&rho;" *)
      | LETTER "s" → "&#963;" (* "\&sigma;" *)
      | LETTER "S" → "&#931;" (* "\&Sigma;" *)
      | LETTER "t" → "&#964;" (* "\&tau;" *)
      | LETTER "u" → "&#965;" (* "\&upsilon;" *)
      | LETTER "U" → "&#933;" (* "\&Upsilon;" *)
      | LETTER "v" → "&#962;" (* "\&sigmaf" *)
      | LETTER "w" → "&#969;" (* "\&omega;" *)
      | LETTER "W" → "&#937;" (* "\&Omega;" *)
      | LETTER "x" → "&#958;" (* "\&xi;" *)
      | LETTER "X" → "&#926;" (* "\&Xi;" *)
      | LETTER "y" → "&#951;" (* "\&eta;" *)
      | LETTER "z" → "&#950;" (* "\&zeta;" *)
      | LETTER "Z" → "&#918;" (* "\&Zeta;" *)
      | "*" → "&#215;" (* "\&times;" *)
      | "+" → "+"
      | "@" → "&#176;" (* "\&deg;" *)
      | "'" → "&#8242;" (* "\&prime;" *)
      | "|" → "|"
      | "!" → "!"
      | "~" → "~"
      | "=" → "="
      | "," → ",␣"
      | i = INT → i
    ] ];
  htmlmath_word :
    [ [ w = LIST0 htmlmath; `EOI → String.concat "" w ] ];
END
;
value math_to_html = transducer htmlmath_word
;
(*****************************************************************)
(* Numeric code encoding, for devanagari sorting and other processing *)
(*****************************************************************)
value lower = Gram.Entry.mk "lower␣case␣as␣letter␣VH"
and word = Gram.Entry.mk "word␣VH"
and wx = Gram.Entry.mk "letter␣WX"
and wordwx = Gram.Entry.mk "word␣WX"
and kh = Gram.Entry.mk "letter␣KH"
and wordkh = Gram.Entry.mk "word␣KH"
```

```
and sl = Gram.Entry.mk "letter␣SL"
and wordsl = Gram.Entry.mk "word␣SL"
;
EXTEND Gram (* skt to nat *)
  lower : (* removes accents, keeps initial quote as avagraha *)
    [ [ LETTER "f" → 21
      | "\""; LETTER "n" → 21 (* compat Velthuis *)
      | LETTER "z" → 46 (* ziva *)
      | "\""; LETTER "s" → 46 (* compat Velthuis *)
      | LETTER "G" → 21 (* compat KH *)(* inconsistent with upper *)
      | LETTER "M" → 14
      | LETTER "H" → 16
      | LETTER "R" → 7
      | LETTER "S" → 47
      | "\""; LETTER "m" → 15 (* compat Velthuis *)
      | "~"; "~" → 15 (* candrabindu *)
      | "~"; LETTER "n" → 26
(* OBS — "+"; c=lower -¿ c (* prevent hyphenation in TeX *) *)
      | "-" → 0 (* notation for affixing *)
      | "+" → 100 (* notation for compounding *)
      | "&" → −1 (* & = alternate avagraha preserved - legacy *)
      | "_" → 50 (* sentential hiatus *)
      | "'"; LETTER "a"; LETTER "a" → 2 (* accented vowels - accent is lost *)
      | "'"; LETTER "a"; LETTER "i" → 11
      | "'"; LETTER "a"; LETTER "u" → 13
      | "'"; LETTER "a"; "$" → 1 (* pr'a-uga *)
      | "'"; LETTER "a" → 1
      | "'"; LETTER "i" → 3
      | "'"; LETTER "u" → 5
      | "'"; LETTER "e" → 10
      | "'"; LETTER "o"; "$" → 12 (* g'o-agra *)
      | "'"; LETTER "o" → 12
      | "'" → −1 (* avagraha *)
      | "."; "."; "."; c = lower → c
      | "."; LETTER "t"; LETTER "h" → 28
      | "."; LETTER "t" → 27
      | "."; LETTER "d"; LETTER "h" → 30
      | "."; LETTER "d" → 29
      | "."; LETTER "s" → 47
      | "."; LETTER "n" → 31
```

```
| "."; LETTER "r"; LETTER "r" → 8
| "."; LETTER "r" → 7
| "."; LETTER "l" → 9
| "."; LETTER "m" → 14
| "."; LETTER "h" → 16
| ":" → 16 (* alternate notation for vigraha *)
| LETTER "a"; LETTER "a"; "|"; LETTER "a" → −3 (* *a *)
| LETTER "a"; LETTER "a"; "|"; LETTER "i" → −4 (* *i *)
| LETTER "a"; LETTER "a"; "|"; LETTER "u" → −5 (* *u *)
| LETTER "a"; LETTER "a"; "|"; LETTER "A" → −9 (* *a *)
| LETTER "a"; LETTER "a"; "|"; LETTER "I" → −7 (* *i *)
| LETTER "a"; LETTER "a"; "|"; LETTER "U" → −8 (* *u *)
| LETTER "a"; LETTER "a"; "|"; LETTER "r" → −6 (* *r *)
| LETTER "a"; LETTER "a"; "|"; LETTER "C" → 123 (* *C *)
| LETTER "a"; LETTER "a" → 2
| LETTER "a"; LETTER "i" → 11
| LETTER "a"; LETTER "u" → 13
| LETTER "a"; "$" → 1 (* pra-ucya *)
| LETTER "a" → 1
| LETTER "i"; LETTER "i" → 4
| LETTER "i" → 3
| LETTER "u"; LETTER "u" → 6
| LETTER "u" → 5
| LETTER "e" → 10
| LETTER "o"; "$" → 12 (* go-agraa *)
| LETTER "o" → 12
| LETTER "k"; LETTER "h" → 18
| LETTER "k" → 17
| LETTER "g"; LETTER "h" → 20
| LETTER "g" → 19
| LETTER "c"; LETTER "h" → 23
| LETTER "c" → 22
| LETTER "j"; LETTER "h" → 25
| LETTER "j" → 24
| LETTER "t"; LETTER "h" → 33
| LETTER "t" → 32
| LETTER "d"; LETTER "h" → 35
| LETTER "d" → 34
| LETTER "p"; LETTER "h" → 38
| LETTER "p" → 37
```

| *LETTER* "b"; *LETTER* "h" → 40
| *LETTER* "b" → 39
| *LETTER* "n" → 36
| *LETTER* "m" → 41
| *LETTER* "y" → 42
| *LETTER* "r" → 43
| *LETTER* "l" → 44
| *LETTER* "v" → 45
| *LETTER* "w" → 45 (∗ alternate v rather than raising Stream error ∗)
| *LETTER* "s" → 48
| *LETTER* "h" → 49
| "#"; *i* = *INT* → 50 + *int_of_string* *i* (∗ 0 ∗)
| "["; "-"; "]" → − 2 (∗ amuissement ∗)

(∗ Special codes code 50 hiatus Canon.decode 50 = "_" codes 51 to 59 - 9 homonymy indexes code -1 -¿ "'" (∗ avagraha ∗) code -2 -¿ "[-]" (∗ amuissement ∗) code -3 -¿ "aa|a" (∗ sandhi of aa and a ∗) code -4 -¿ "aa|i" (∗ sandhi of aa and i ∗) code -5 -¿ "aa|u" (∗ sandhi of aa and u ∗) code -6 -¿ "aa|r" (∗ sandhi of aa and .r ∗) code -7 -¿ "aa|I" (∗ sandhi of aa and ii ∗) code -8 -¿ "aa|U" (∗ sandhi of aa and uu ∗) code -9 -¿ "aa|A" (∗ sandhi of aa and aa ∗) code 123 -¿ "aa|C" (∗ sandhi of aa and ch ∗) codes 101 to 149 reserved for upper case encodings in *Canon.decode_ref* codes 124, 149, 249 used for variants resp. j' of j 24 and h',h" of h 49 in *Int_sandhi* ∗)

]];
word :
[[*w* = *LIST0 lower*; ‘*EOI* → *w*]];
wx :
[[*LETTER* "a" → 1
| *LETTER* "A" → 2
| *LETTER* "i" → 3
| *LETTER* "I" → 4
| *LETTER* "u" → 5
| *LETTER* "U" → 6
| *LETTER* "q" → 7
| *LETTER* "Q" → 8
| *LETTER* "L" → 9
| *LETTER* "e" → 10
| *LETTER* "E" → 11
| *LETTER* "o" → 12
| *LETTER* "O" → 13
| *LETTER* "M" → 14
| *LETTER* "z" → 15 (∗ candrabindu ∗)

```
  | LETTER "H" → 16
  | LETTER "k" → 17
  | LETTER "K" → 18
  | LETTER "g" → 19
  | LETTER "G" → 20
  | LETTER "f" → 21
  | LETTER "c" → 22
  | LETTER "C" → 23
  | LETTER "j" → 24
  | LETTER "J" → 25
  | LETTER "F" → 26
  | LETTER "t" → 27
  | LETTER "T" → 28
  | LETTER "d" → 29
  | LETTER "D" → 30
  | LETTER "N" → 31
  | LETTER "w" → 32
  | LETTER "W" → 33
  | LETTER "x" → 34
  | LETTER "X" → 35
  | LETTER "n" → 36
  | LETTER "p" → 37
  | LETTER "P" → 38
  | LETTER "b" → 39
  | LETTER "B" → 40
  | LETTER "m" → 41
  | LETTER "y" → 42
  | LETTER "r" → 43
  | LETTER "l" → 44
  | LETTER "v" → 45
  | LETTER "S" → 46
  | LETTER "R" → 47
  | LETTER "s" → 48
  | LETTER "h" → 49
  | "-" → 0 (* notation for affixing *)
  | "+" → 100 (* notation for compounding *)
  | "_" → 50 (* sentential hiatus *)
  | LETTER "Z" → −1 (* avagraha *)
  | "#"; i = INT → 50 + int_of_string i (* 0 *)
] ];
```

```
wordwx :
  [ [ w = LIST0 wx; `EOI -> w ] ];
kh :
  [ [ LETTER "A" -> 2
    | LETTER "i" -> 3
    | LETTER "I" -> 4
    | LETTER "u" -> 5
    | LETTER "U" -> 6
    | LETTER "R"; LETTER "R" -> 8
    | LETTER "R" -> 7
    | LETTER "L" -> 9
    | LETTER "e" -> 10
    | LETTER "a"; LETTER "i" -> 11
    | LETTER "o" -> 12
    | LETTER "a"; LETTER "u" -> 13
    | LETTER "a" -> 1
    | LETTER "M" -> 14
      (* candrabindu absent *)
    | LETTER "H" -> 16
    | LETTER "k"; LETTER "h" -> 18
    | LETTER "k" -> 17
    | LETTER "g"; LETTER "h" -> 20
    | LETTER "g" -> 19
    | LETTER "G" -> 21
    | LETTER "c"; LETTER "h" -> 23
    | LETTER "c" -> 22
    | LETTER "j"; LETTER "h" -> 25
    | LETTER "j" -> 24
    | LETTER "J" -> 26
    | LETTER "T"; LETTER "h" -> 28
    | LETTER "T" -> 27
    | LETTER "D"; LETTER "h" -> 30
    | LETTER "D" -> 29
    | LETTER "N" -> 31
    | LETTER "t"; LETTER "h" -> 33
    | LETTER "t" -> 32
    | LETTER "d"; LETTER "h" -> 35
    | LETTER "d" -> 34
    | LETTER "n" -> 36
    | LETTER "p"; LETTER "h" -> 38
```

```
      | LETTER "p" → 37
      | LETTER "b"; LETTER "h" → 40
      | LETTER "b" → 39
      | LETTER "m" → 41
      | LETTER "y" → 42
      | LETTER "r" → 43
      | LETTER "l" → 44
      | LETTER "v" → 45
      | LETTER "z" → 46
      | LETTER "S" → 47
      | LETTER "s" → 48
      | LETTER "h" → 49
      | "'" → −1 (* avagraha *)
      | "-" → 0 (* notation for affixing *)
      | "+" → 100 (* notation for compounding *)
      | "_" → 50 (* sentential hiatus *)
      (* avagraha missing *)
      | "#"; i = INT → 50 + int_of_string i (* 0 *)
    ] ];
  wordkh :
    [ [ w = LIST0 kh; `EOI → w ] ];
  sl :
    [ [ LETTER "a" → 1
      | LETTER "A" → 2
      | LETTER "i" → 3
      | LETTER "I" → 4
      | LETTER "u" → 5
      | LETTER "U" → 6
      | LETTER "f" → 7
      | LETTER "F" → 8
      | LETTER "x" → 9
      | LETTER "e" → 10
      | LETTER "E" → 11
      | LETTER "o" → 12
      | LETTER "O" → 13
      | LETTER "M" → 14
      | "~" → 15
      | LETTER "H" → 16
      | LETTER "k" → 17
      | LETTER "K" → 18
```

```
    | LETTER "g" → 19
    | LETTER "G" → 20
    | LETTER "N" → 21
    | LETTER "c" → 22
    | LETTER "C" → 23
    | LETTER "j" → 24
    | LETTER "J" → 25
    | LETTER "Y" → 26
    | LETTER "w" → 27
    | LETTER "W" → 28
    | LETTER "q" → 29
    | LETTER "Q" → 30
    | LETTER "R" → 31
    | LETTER "t" → 32
    | LETTER "T" → 33
    | LETTER "d" → 34
    | LETTER "D" → 35
    | LETTER "n" → 36
    | LETTER "p" → 37
    | LETTER "P" → 38
    | LETTER "b" → 39
    | LETTER "B" → 40
    | LETTER "m" → 41
    | LETTER "y" → 42
    | LETTER "r" → 43
    | LETTER "l" → 44
    | LETTER "v" → 45
    | LETTER "S" → 46
    | LETTER "z" → 47
    | LETTER "s" → 48
    | LETTER "h" → 49
    | "'" → −1 (∗ avagraha ∗)
    | "-" → 0 (∗ notation for affixing ∗)
    | "+" → 100 (∗ notation for compounding ∗)
    | "_" → 50 (∗ sentential hiatus ∗)
    | "#"; i = INT → 50 + int_of_string i (∗ 0 ∗)
  ] ];
 wordsl :
  [ [ w = LIST0 sl; OPT "."; `EOI → w ] ];
END
```

```
;
value code_raw s = (* VH transliteration *)
  try Gram.parse_string word Loc.ghost s
  with
  [ Loc.Exc_located loc e → do
      { Format.eprintf "\nIn string \"%s\", at location %s :\n%!"
                       s (Loc.to_string loc)
      ; raise e
      }
  ]
and code_raw_WX s =
  try Gram.parse_string wordwx Loc.ghost s
  with
  [ Loc.Exc_located loc e → do
      { Format.eprintf "\nIn string \"%s\", at location %s :\n%!"
                       s (Loc.to_string loc)
      ; raise e
      }
  ]
and code_raw_KH s =
  try Gram.parse_string wordkh Loc.ghost s
  with
  [ Loc.Exc_located loc e → do
      { Format.eprintf "\nIn string \"%s\", at location %s :\n%!"
                       s (Loc.to_string loc)
      ; raise e
      }
  ]
and code_raw_SL s =
  try Gram.parse_string wordsl Loc.ghost s
  with
  [ Loc.Exc_located loc e → do
      { Format.eprintf "\nIn string \"%s\", at location %s :\n%!"
                       s (Loc.to_string loc)
      ; raise e
      }
  ]
;
(**********************************************************************)
(* The following gives codes to proper names, starting with upper letters *)
```

```
(************************************************************************)
value upper_lower = Gram.Entry.mk "upper␣case"
and wordu = Gram.Entry.mk "wordu"
;
EXTEND Gram (* skt to nat *)
  upper_lower :
    [ [ "\""; LETTER "S" → 146
      | LETTER "Z" → 146
      | LETTER "A"; LETTER "A" → 102
      | LETTER "A"; LETTER "i" → 111
      | LETTER "A"; LETTER "u" → 113
      | LETTER "A" → 101
      | LETTER "I"; LETTER "I" → 104
      | LETTER "I" → 103
      | LETTER "U"; LETTER "U" → 106
      | LETTER "U" → 105
      | "."; LETTER "S" → 147
      | "."; LETTER "R" → 107
      | "."; LETTER "T"; LETTER "h" → 128
      | "."; LETTER "T" → 127
      | "."; LETTER "D"; LETTER "h" → 130
      | "."; LETTER "D" → 129
      | LETTER "E" → 110
      | LETTER "O" → 112
      | LETTER "K"; LETTER "h" → 118
      | LETTER "K" → 117
      | LETTER "G"; LETTER "h" → 120
      | LETTER "G" → 119
      | LETTER "C"; LETTER "h" → 123
      | LETTER "C" → 122
      | LETTER "J"; LETTER "h" → 125
      | LETTER "J" → 124
      | LETTER "T"; LETTER "h" → 133
      | LETTER "T" → 132
      | LETTER "D"; LETTER "h" → 135
      | LETTER "D" → 134
      | LETTER "N" → 136
      | LETTER "P"; LETTER "h" → 138
      | LETTER "P" → 137
      | LETTER "B"; LETTER "h" → 140
```

| *LETTER* "B" → 139
| *LETTER* "M" → 141
| *LETTER* "Y" → 142
| *LETTER* "R" → 143
| *LETTER* "L" → 144
| *LETTER* "V" → 145
| *LETTER* "S" → 148
| *LETTER* "H" → 149
(∗ duplication with lower necessary in order to get proper sharing of prefix ∗)
| "\""; *LETTER* "n" → 21
| *LETTER* "f" → 21
| "\""; *LETTER* "s" → 46
| *LETTER* "z" → 46
| "~"; *LETTER* "n" → 26
| "~"; "~" → 15
| "+"; $c = upper\_lower$ → c
| "-" → 0
| "_" → 50 (∗ hiatus ∗)
| "$"; $c = upper\_lower$ → c (∗ word hiatus for VH trans pra-uga ∗)
| "'"; $c = upper\_lower$ → c
| "."; "."; "."; $c = upper\_lower$ → c
| "."; *LETTER* "t"; *LETTER* "h" → 28
| "."; *LETTER* "t" → 27
| "."; *LETTER* "d"; *LETTER* "h" → 30
| "."; *LETTER* "d" → 29
| "."; *LETTER* "s" → 47
| "."; *LETTER* "n" → 31
| "."; *LETTER* "r"; *LETTER* "r" → 8
| "."; *LETTER* "r" → 7
| "."; *LETTER* "l" → 9
| "."; *LETTER* "m" → 14
| "."; *LETTER* "h" → 16
| *LETTER* "a"; *LETTER* "a" → 2
| *LETTER* "a"; *LETTER* "i" → 11
| *LETTER* "a"; *LETTER* "u" → 13
| *LETTER* "a" → 1
| *LETTER* "i"; *LETTER* "i" → 4
| *LETTER* "i" → 3
| *LETTER* "u"; *LETTER* "u" → 6
| *LETTER* "u" → 5

```
    | LETTER "e" → 10
    | LETTER "o" → 12
    | LETTER "k"; LETTER "h" → 18
    | LETTER "k" → 17
    | LETTER "g"; LETTER "h" → 20
    | LETTER "g" → 19
    | LETTER "c"; LETTER "h" → 23
    | LETTER "c" → 22
    | LETTER "j"; LETTER "h" → 25
    | LETTER "j" → 24
    | LETTER "t"; LETTER "h" → 33
    | LETTER "t" → 32
    | LETTER "d"; LETTER "h" → 35
    | LETTER "d" → 34
    | LETTER "p"; LETTER "h" → 38
    | LETTER "p" → 37
    | LETTER "b"; LETTER "h" → 40
    | LETTER "b" → 39
    | LETTER "n" → 36
    | LETTER "m" → 41
    | LETTER "y" → 42
    | LETTER "r" → 43
    | LETTER "l" → 44
    | LETTER "v" → 45
    | LETTER "s" → 48
    | LETTER "h" → 49
    | "#"; i = INT → 50 + int_of_string i
  ] ];
 wordu :
  [ [ w = LIST0 upper_lower; `EOI → w ] ];
END
;
(* Similar to code_raw but accepts upper letters. *)
value code_rawu s =
 try Gram.parse_string wordu Loc.ghost s with
 [ Loc.Exc_located loc e → do
    { Format.eprintf "\nIn string \"%s\", at location %s :\n%!"
                    s (Loc.to_string loc)
    ; raise e
    }
```

```
  ]
;
```

Simplified mapping for matching without diacritics

```
value simplified = Gram.Entry.mk "simplified"
and wordd = Gram.Entry.mk "wordd"
;
EXTEND Gram (* skt to nat *)
  simplified :
    [ [ "\""; LETTER "S" → 148
      | LETTER "Z" → 148
      | LETTER "A"; LETTER "A" → 101
      | LETTER "A"; LETTER "i" → 111
      | LETTER "A"; LETTER "u" → 113
      | LETTER "A" → 101
      | LETTER "I"; LETTER "I" → 103
      | LETTER "I" → 103
      | LETTER "U"; LETTER "U" → 105
      | LETTER "U" → 105
      | "."; LETTER "S" → 148
      | "."; LETTER "R" → 143
      | "."; LETTER "T"; LETTER "h" → 132
      | "."; LETTER "T" → 132
      | "."; LETTER "D"; LETTER "h" → 134
      | "."; LETTER "D" → 134
      | LETTER "E" → 110
      | LETTER "O" → 112
      | LETTER "K"; LETTER "h" → 117
      | LETTER "K" → 117
      | LETTER "G"; LETTER "h" → 119
      | LETTER "G" → 119
      | LETTER "C"; LETTER "h" → 122
      | LETTER "C" → 122
      | LETTER "J"; LETTER "h" → 124
      | LETTER "J" → 124
      | LETTER "T"; LETTER "h" → 132
      | LETTER "T" → 132
      | LETTER "D"; LETTER "h" → 134
      | LETTER "D" → 134
      | LETTER "N" → 136
```

```
| LETTER "P"; LETTER "h" → 137
| LETTER "P" → 137
| LETTER "B"; LETTER "h" → 139
| LETTER "B" → 139
| LETTER "M" → 141
| LETTER "Y" → 142
| LETTER "R" → 143
| LETTER "L" → 144
| LETTER "V" → 145
| LETTER "S"; LETTER "h" → 148
| LETTER "S" → 148
| LETTER "H" → 149
(* duplication with lower necessary in order to get proper sharing of prefix *)
| "\""; LETTER "m" → 15
| "\""; LETTER "n" → 36
| LETTER "f" → 36
| "\""; LETTER "s" → 48
| LETTER "z" → 48
| "~"; LETTER "n" → 36
| "~"; "~" → 15
| "+"; c = upper_lower → c
| "-" → 0
| "_" → 50 (* hiatus *)
| "$"; c = upper_lower → c (* word hiatus for VH trans pra-uga *)
| "'"; c = upper_lower → c
| "."; "."; "."; c = upper_lower → c
| "."; LETTER "t"; LETTER "h" → 32
| "."; LETTER "t" → 32
| "."; LETTER "d"; LETTER "h" → 34
| "."; LETTER "d" → 34
| "."; LETTER "s" → 48
| "."; LETTER "n" → 36
| "."; LETTER "r"; LETTER "r" → 43
| "."; LETTER "r" → 43
| "."; LETTER "l" → 44
| "."; LETTER "m" → 41
| "."; LETTER "h" → 49
| LETTER "a"; LETTER "a" → 1
| LETTER "a"; LETTER "i" → 11
| LETTER "a"; LETTER "u" → 13
```

```
      | LETTER "a" → 1
      | LETTER "i"; LETTER "i" → 3
      | LETTER "i" → 3
      | LETTER "u"; LETTER "u" → 5
      | LETTER "u" → 5
      | LETTER "e" → 10
      | LETTER "o"; LETTER "u" → 5 (∗ Vishnou ∗)
      | LETTER "o" → 12
      | LETTER "k"; LETTER "h" → 17
      | LETTER "k" → 17
      | LETTER "g"; LETTER "h" → 19
      | LETTER "g" → 19
      | LETTER "c"; LETTER "h" → 48 (∗ Vichnou , Krichna ∗)
      | LETTER "c" → 22
      | LETTER "j"; LETTER "h" → 24
      | LETTER "j" → 24
      | LETTER "t"; LETTER "h" → 32
      | LETTER "t" → 32
      | LETTER "d"; LETTER "h" → 34
      | LETTER "d" → 34
      | LETTER "p"; LETTER "h" → 37
      | LETTER "p" → 37
      | LETTER "b"; LETTER "h" → 39
      | LETTER "b" → 39
      | LETTER "n" → 36
      | LETTER "m" → 41
      | LETTER "y" → 42
      | LETTER "r"; LETTER "i"; LETTER "i" → 43 (∗ consistency with: ∗)
      | LETTER "r"; LETTER "i" → 43 (∗ Krishna ∗)
      | LETTER "r"; LETTER "u"; LETTER "u" → 43 (∗ consistency with: ∗)
      | LETTER "r"; LETTER "u" → 43 (∗ vikruti ∗)
      | LETTER "r" → 43
      | LETTER "l" → 44
      | LETTER "v" → 45
      | LETTER "s"; LETTER "h" → 48
      | LETTER "s" → 48
      | LETTER "h" → 49
    ] ];
  wordd :
    [ [ w = LIST0 simplified; `EOI → w
```

```
      | w = LIST0 simplified; "#"; INT; `EOI → w (* homo index ignored *)
    ] ];
END
;
(* Similar to code_skt_ref but simplified (no diacritics) *)
value code_rawd s =
  try Gram.parse_string wordd Loc.ghost s with
  [ Loc.Exc_located loc e → do
      { Format.eprintf "\nIn string \"%s\", at location %s :\n%!"
                        s (Loc.to_string loc)
      ; raise e
      }
  ]
;
```

Module Encode

Defines various encodings of transliterated strings into words as int lists

```
open Transduction; (* code_raw and similar *)
open Phonetics; (* homonasal vowel *)
exception In_error of string (* Error in user or corpus input *)
;
value is_vowel c = vowel c ∨ c > 100 ∧ c < 114 (* accounts for upper case *)
;
(* anusvara substituted by nasal or normalized to 14 when original *)
(* anunaasika before vowels treated as anusvaara *)
value rec normalize = normal_rec False
  where rec normal_rec after_vow = fun
  [ [] → []
  | [ 14 (* .m *) ] → [ 14 ] (* and NOT m *)
  | [ 14 (* .m *) :: [ c :: l ] ] →
    if after_vow then
        let c' = homonasal c in [ c' :: [ c :: normal_rec (is_vowel c) l ] ]
    else raise (In_error "Anusvaara should follow vowel")
  | [ 15 (*   *) :: [ c :: l ] ] → (* 31-08-19 anunaasika normalisation *)
    if after_vow then (* anunaasika assimilated to anasvaara *)
        let c' = homonasal c in [ c' :: [ c :: normal_rec (is_vowel c) l ] ]
    else [ 15 :: normal_rec False [ c :: l ] ]
  | [ 16 (* .h *) ] →
```

```
      if after_vow then [ 16 ]
      else raise (In_error "Visarga should follow vowel")
(* No change to visarga since eg praata.hsvasu.h comes from praatar—svasu.h and praatass-
vasu.h is not recognized. This is contrary to Henry§43 note 1. corresponding to the following
code: | [ 16 (× .h ×) :: [ c :: l ] ] → if after_vow then let c' = if sibilant c then c else 16 (× du.hkha
) in [ c' :: [ c :: normal_rec (is_vowel c) l ] ] else raise (In_error "Visarga should follow vowel")
*)
   | [ 50 :: l ] → [ 50 :: normal_rec False l ] (* hiatus *)
   | [ c :: l ] → [ c :: normal_rec (is_vowel c) l ]
   ]
;
value code_string str = normalize (code_raw str) (* standard VH *)
and code_string_WX str = normalize (code_raw_WX str)
and code_string_KH str = normalize (code_raw_KH str)
and code_string_SL str = normalize (code_raw_SL str)
and code_skt_ref str = normalize (code_rawu str)
and code_skt_ref_d str = normalize (code_rawd str)
;
(* Switching code function according to transliteration convention *)
value switch_code = fun (* normalizes anusvaara in its input *)
   [ "VH" → code_string (* Canon.decode *)
   | "WX" → code_string_WX (* Canon.decode_WX *)
   | "KH" → code_string_KH (* Canon.decode_KH *)
   | "SL" → code_string_SL (* Canon.decode_SL *)
   | _ → failwith "Unknown transliteration scheme"
   ]
;
value rev_code_string str = Word.mirror (code_string str)
;
(* anchor : string → string – used in Morpho_html.url and Sanskrit *)
value anchor t =
   let canon c = if c > 100 then Canon.canon_upper_html c
                                   else Canon.canon_html c in
   let catenate c (s, b) = (* similar to Canon.catenate *)
         let b' = c > 0 ∧ c < 14 (* Phonetics.vowel c *) in
         let hiatus = if b ∧ b' then "_" ^ s else s in
         (canon c ^ hiatus , b') in
   let word = code_skt_ref t in
   let (s, _) = List.fold_right catenate word ("",False) in s
;
```

```
(* strips from word stack (revcode) homonym index if any *)
value strip w = match w with
  [ [ last :: rest ] → if last > 50 then rest (* remove homonymy index *)
                       else w
  | [] → failwith "Empty␣stem␣to␣strip"
  ]
;
value rstem w = strip (Word.mirror w)
;
value rev_strip w = Word.mirror (rstem w) (* compute_mw_links *)
;
(* Builds revword normalised stem from entry string of root *)
(* Used by Verbs.revstem, Nouns.enter_iic, Print_dict *)
value rev_stem str = strip (rev_code_string str)
;
(* Takes a reversed word and returns its canonical name (homo,stem) *)
value decompose w = match w with
  [ [ last :: rest ] →
       if last > 50 then (last − 50, Word.mirror rest)
                    else (0, Word.mirror w)
  | [] → failwith "Empty␣stem␣to␣decompose"
  ]
;
(* Temporary - encoding of homo as last character of word *)
value decompose_str str =
  decompose (rev_code_string str) (* ugly multiple reversals *)
;
value normal_stem str = Word.mirror (rev_stem str)
;
value normal_stem_str str = Canon.decode (normal_stem str) (* horror *)
;
(* strips homonymy index of raw input - similar awful double reversal *)
value code_strip_raw s = rev_strip (code_raw s)
(* Hopefully used only for devanagari printing below *)
(* Same function, with skt input, is Subst.stripped_code_skt *)
(* A cleaner solution would be to have type lexeme = (word * int) and "x#5" represented
as (x,5) (0 if no homophone) *)
;
value skt_to_deva str = try Canon.unidevcode (code_string str) with
                                          [ Failure _ → raise (In_error str) ]
```

```
and skt_raw_to_deva str = try Canon.unidevcode (code_raw str) with
                              [ Failure _ → raise (In_error str) ]
and skt_raw_strip_to_deva str = try Canon.unidevcode (code_strip_raw str) with
                              [ Failure _ → raise (In_error str) ]
;
(* Following not needed since Transduction.skt_to_html is more direct value skt_to_roma str = Canon.
*)
```

diff with string in Velthuis transliteration - caution: argument swap

```
value diff_str str w = Word.diff w (code_string str)
;
```

Module Order

lexicographic comparison

```
value rec lexico l1 l2 = match l1 with
  [ [] → True
  | [ c1 :: r1 ] → if c1 = 50 (* hiatus *) then lexico r1 l2
                   else match l2 with
      [ [] → False
      | [ c2 :: r2 ] → if c2 = 50 (* hiatus *) then lexico l1 r2
                       else if c2 > 50 then c1 > 50 ∧ c1 < c2 (* homonym indexes *)
                       else if c1 > 50 then True
                       else if c2 < c1 then False
                       else if c2 = c1 then lexico r1 r2
                       else True
      ]
  ]
;
(* for use as argument to List.sort *)
value order w w' = if w = w' then 0 else if lexico w w' then -1 else 1
;
(* end; *)
```

Module Padapatha

```
value sanskrit_chunk encode s =
  match encode s with (* avagraha reverts to a *)
```

```
  [ [ −1 :: l ] → [ 1 :: l ] (* only initial avagraha reverts to a *)
  | x → x
  ]
;
(* Preprocessing of corpus to prepare padapatha form from list of chunks *)
(* This is extremely important from the segmenter complexity point of view *)
(* Since it takes hints at parallel treatment from non-ambiguous blanks. *)
exception Hiatus
;
exception Glue
;
(* We raise Glue below when there are multiple ways to obtain the current break, in which
case we do not profit of the sandhi hint. Furthermore, this is incomplete, notably when one
of the sandhied forms is a vocative. *)
(* Chunk w is adjusted for padapatha in view of next character c *)
(* No attempt is made to change c and thus tacchrutvaa is not chunkable. *)
(* This function defines the maximal separability of devanaagarii into chunks but is not
always able to go as far as creating the full padapaa.tha *)
value adjust c w = match Word.mirror w with
  [ [] → failwith "adjust"
  | [ last :: rest ] → match last with
        [ 14 (* .m *) → Word.mirror [ 41 (* m *) :: rest ] (* revert .m to m *)
              (* note: .m coming from sandhi of n is followed by sibilant and chunking is
allowed only after this sibilant *)
        | 11 (* ai *) when c = 43 (* r *) → raise Hiatus
              (* For ai.h+r -¿ ai r Whitney§179 en fait, toute voyelle longue *)
     (* — 10 — 13 No Hyatus: te rasasaarasafgrahavidhim but es-r -¿ er missed *)
        | 12 (* o *) → if rest = [ 40 ] (* bh from bhos -¿ bho *) then
                              Encode.code_string "bhos" (* "bho␣raama" "bho␣bhos" *)
                           else if rest = [ 49; 1 ] (* aho *) then
                              Encode.code_string "aho" (* "aho␣raama" *)
                           else if Phonetics.turns_visarg_to_o c ∨ c = 1
                                (* zivoham must be entered as zivo'ham (avagraha) *)
                                then Word.mirror [ 16 :: [ 1 :: rest ] ]
                                (* restore visarga, assuming original a.h form *)
                                (* This may miss hiatus os + rx -¿ o rx *)
                           else w
        | 1 (* a *) → if c = 1 then w else
                          if Phonetics.vowel c then raise Hiatus else w
        | 2 (* aa *) → if Phonetics.vowel c then raise Hiatus else
```

```
                    if Phonetics.elides_visarg_aa c then raise Hiatus else
                    w (* Hiatus except c surd unaspirate ? *)
  | 4 (* ii *) (* possible visarga vanishes *)
  | 6 (* uu *) → if c = 43 (* r *) then raise Hiatus else w
  (* next 4 rules attempt to revert last to 'd' in view of c *)
  | 34 (* d *) → if c = 35 (* dh *) then raise Glue else
                    if Phonetics.is_voiced c
                       then Word.mirror [ 32 :: rest ] (* d -¿ t *)
                       else w
  | 24 (* j *) → if Phonetics.turns_t_to_j c (* tat+jara -¿ tajjara *)
                       then Word.mirror [ 32 :: rest ] (* j -¿ t *)
                       else w
  | 26 (*  n *) → match rest with
       [ [ 26 (*  n *) :: ante ] → match ante with
            (* optional doubling of  n in front of vowel *)
            [ [ v :: _ ] → if Phonetics.short_vowel v ∧ Phonetics.vowel c
                                   then Word.mirror rest
                                   else failwith "padapatha"
            | _ → failwith "padapatha"
            ]
       | _ → if c = 23 (* ch could come from ch or z *)
                 then raise Glue
             else if Phonetics.turns_n_to_palatal c
                         (* taan+zaastravimukhaan -¿ taa nzaastravimukhaan *)
                  then Word.mirror [ 36 (* n *) :: rest ] (* n -¿  n *)
                  else w
       ]
  | 29 (* .d *) → if c = 30 (* .dh *) then raise Glue else
                     if Phonetics.is_voiced c
                       then Word.mirror [ 27 :: rest ] (* .d -¿ .t *)
                       else w
  | 39 (* b *) → if c = 40 (* bh *) then raise Glue else
                    if Phonetics.is_voiced c
                       then Word.mirror [ 37 :: rest ] (* b -¿ p *)
                       else w
  | 19 (* g *) → if c = 20 (* gh *) then raise Glue else
                    if Phonetics.is_voiced c (* vaak+vazya *)
                       then Word.mirror [ 17 :: rest ] (* g -¿ k *)
                       else w
  | 36 (* n *) → match rest with
```

```
          [ [ 36 (* n *) :: ante ] → match ante with
                (* optional doubling of n in front of vowel *)
             [ [ v :: _ ] → if Phonetics.short_vowel v ∧ Phonetics.vowel c
                                 then Word.mirror rest (* gacchann eva *)
                              else w
             | _ → failwith "padapatha"
             ]
          | _ → if c = 36 (* n *) ∨ c = 41 (* m *)
                   then raise Glue (* since d—m-¿nn and n—m -¿ nm *)
                     (* Word.mirror 32 :: rest (* n -¿ t *) *)
                     (* incompleteness: raajan naasiin vocatif raajan *)
                 else w
          ]
     | 22 (* c *) → if c = 22 then Word.mirror [ 32 :: rest ] (* c -¿ t *)
                     else if c = 23 (* ch could come from ch or z *)
                          then raise Glue else w
     | 44 (* l *) → if c = last
                       then Word.mirror [ 32 :: rest ] (* l -¿ t *)
                       else w
     | 21 (* f *) → match rest with
          [ [ 21 (* f *) :: ante ] → match ante with
                (* optional doubling of f in front of vowel *)
             [ [ v :: _ ] → if Phonetics.short_vowel v ∧ Phonetics.vowel c
                                 then Word.mirror rest
                              else failwith "padapatha"
             | _ → failwith "padapatha"
             ]
          | _ → if c = 41 (* m *) (* vaak+mayi *)
                      then Word.mirror [ 17 :: rest ] (* f -¿ k *)
                      else w
          ]
     (* NB if last is y, r or v and c is vowel, then it may come from resp. i,ii, .r,.rr, u,uu
and this choice means that we cannot make a chunk break here *)
     | 42 (* y *) | 45 (* v *) → if Phonetics.vowel c then raise Glue
                                    else w (* will fail *)
     | 43 (* r *) → if Phonetics.turns_visarg_to_o c ∨ Phonetics.vowel c
                       then Word.mirror [ 16 :: rest ] (* visarg restored *)
                     else w (* pb punar pitar etc *)
     | 46 (* z *) → match rest with
           [ [ 14 (* .m *) :: b ] → if c = 22 ∨ c = 23 (* c ch *) then
```

```
                                        Word.mirror [ 36 (* n *) :: b ]
                                      else w
          | [ 26 (*  n *) :: _ ] → if c = 46 (* z *) then
                                        Word.mirror [ 36 (* n *) :: rest ]
                                      else w
                                      (* c=23 (* ch *) could come from z *)
          | _ → if c = 22 ∨ c = 23 (* c ch *) then
                    Word.mirror [ 16 (* .h *) :: rest ] else w
          ]
      | 47 (* .s *) → match rest with
          [ [ 14 (* .m *) :: b ] → if c = 27 ∨ c = 28 (* .t .th *) then
                                      Word.mirror [ 36 (* n *) :: b ] else w
          | _ → w
          ]
      | 48 (* s *) → match rest with
          [ [ 14 (* .m *) :: b ] → if c = 32 ∨ c = 33 (* t th *) then
                                      Word.mirror [ 36 (* n *) :: b ] else w
          | _ → if c = 32 ∨ c = 33 (* t th *) then
                    raise Glue else w
          ]
      | _ → w
      ]
  ]
;
(* Called from Sanskrit.read_processed_skt_stream for use in read_sanskrit with argument
read_chunk = sanskrit_chunk encode *)
value padapatha read_chunk l = (* l is list of chunks separated by blanks *)
                              (* returns padapatha as list of forms in terminal sandhi *)
  let rec pad_rec = fun (* returns (c,l) with c first char of first pada in l *)
    [ [] → (−1, [])
    | [ chk :: chks ] →
      let (c, padas) = pad_rec chks
      and w = read_chunk chk (* initial avagraha reverts to a *) in
      (List.hd w (* next c *),
       try let pada = if c = (−1) then w (* last chunk *)
                      else adjust c w in
           [ pada :: padas ]
       with
        [ Hiatus → match padas with
          [ [] → failwith "padapatha"
```

```
        | [ p :: lp ] → let conc = w @ [ 50 :: p ] in (* w_p *)
                          [ conc :: lp ] (* hiatus indicates a word boundary *)
        ]
      | Glue → match padas with
        [ [] → failwith "padapatha"
        | [ p :: lp ] → let conc = w @ p in
                          [ conc :: lp ] (* we lose the boundary indication *)
        ]
      ])
  ] in
 let (_, padas) = pad_rec l in padas
;
```

Module Skt_lexer

A simple lexer recognizing idents, integers, punctuation symbols, and skipping spaces and comments between The transliteration scheme is Velthuis with aa for long a etc.

```
module Skt_lexer = struct

open Camlp4.PreCast;
open Format;

 module Loc = Loc; (* Using the PreCast Loc *)
 module Error = struct
   type t = string;
   exception E of t;
   value to_string x = x;
   value print = Format.pp_print_string;
 end;
 module Token = struct
   module Loc = Loc;
   type t =
     [ KEYWORD of string
     | IDENT of string
     | INT of int
     | EOI
     ]
   ;
   module Error = Error;
   module Filter = struct
```

```
    type token_filter = Camlp4.Sig.stream_filter t Loc.t
    ;
    type t = string → bool
    ;
    value mk is_kwd = is_kwd
    ;
    value rec filter is_kwd = parser
      [ [: `((KEYWORD s, loc) as p); strm :] →
            if is_kwd s ∨ s = "!" then [: `p; filter is_kwd strm :]
            else raise (Encode.In_error ("Undefined␣token␣:␣" ^ s))
      | [: `x; s :] → [: `x; filter is_kwd s :]
      | [: :] → [: :]
      ]
    ;
    value define_filter _ _ = ()
    ;
    value keyword_added _ _ _ = ()
    ;
    value keyword_removed _ _ = ()
    ;
  end
  ;
  value to_string = fun
    [ KEYWORD s → sprintf "KEYWORD␣%S" s
    | IDENT s → sprintf "IDENT␣%S" s
    | INT i → sprintf "INT␣%d" i
    | EOI → "EOI"
    ]
  ;
  value print ppf x = pp_print_string ppf (to_string x)
  ;
  value match_keyword kwd = fun
    [ KEYWORD kwd' when kwd' = kwd → True
    | _ → False
    ]
  ;
  value extract_string = fun
    [ INT i → string_of_int i
    | IDENT s | KEYWORD s → s
    | EOI → ""
```

```
      ]
    ;
  end
  ;
  open Token
  ;
```

The string buffering machinery - ddr + np

```
value store buf c = do { Buffer.add_char buf c; buf }
;
value rec number buf =
  parser
  [ [: `('0'..'9' as c); s :] → number (store buf c) s
  | [: :] → Buffer.contents buf
  ]
;
value rec skip_to_eol =
  parser
  [ [: `'\n' | '\026' | '\012'; s :] → ()
  | [: `c ; s :] → skip_to_eol s
  ]
;
value ident_char =
  parser
  [ [: `('a'..'z' | 'A'..'Z' | '.' | ':' | '"' | '~' | '\'' | '+' | '-' | '$' as c) :]
    → c ]
;
value rec ident buff =
  parser
  [ [: c = ident_char; s :] → ident (store buff c) s
  | [: :] → Buffer.contents buff
  ]
;
value next_token_fun =
  let rec next_token buff =
      parser _bp
      [ [: c = ident_char; s = ident (store buff c) :] → IDENT s
      | [: `('0'..'9' as c); s = number (store buff c) :] → INT (int_of_string s)
      | [: `c :] _ep → KEYWORD (String.make 1 c)
      ] in
```

```
    let rec next_token_loc =
      parser bp
      [ [: ‘’%’; _ = skip_to_eol; s :] → next_token_loc s (* comments skipped *)
      | [: ‘’ ’ | ’\n’ | ’\r’ | ’\t’ | ’\026’ | ’\012’; s :] → next_token_loc s
      | [: tok = next_token (Buffer.create 80) :] ep → (tok, (bp, ep))
      | [: _ = Stream.empty :] → (EOI, (bp, succ bp))
      ] in
 next_token_loc
;
value mk () =
 let err loc msg = Loc.raise loc (Token.Error.E msg) in
 fun init_loc cstrm → Stream.from
     (fun _ → try let (tok, (bp, ep)) = next_token_fun cstrm in
                  let loc = Loc.move ‘start bp (Loc.move ‘stop ep init_loc) in
                  Some (tok, loc)
              with
              [ Stream.Error str →
                  let bp = Stream.count cstrm in
                  let loc = Loc.move ‘start bp (Loc.move ‘stop (bp + 1) init_loc) in
                  err loc str ])
;
end;
```

Interface for module Sanskrit

```
type skt (* abstract *)
;
type pada = list skt
and sloka = list pada
;
value string_of_skt : skt → string; (* input *)
value skt_of_string : string → skt; (* faking - debug and Subst.record_tad *)
value aa_preverb : skt;
value privative : skt → bool;
value i_root : skt;
value ita_part : skt;
value dagh_root : skt;
value daghna_part : skt;
value arcya_absolutive : skt;
```

value trad_skt : *string* → *skt*;
value trad_sanscrit : *string* → *sloka*;
value trad_skt_list : *string* → *list skt*;
value maha_epic : *skt*;
value rama_epic : *skt*;
value skt_to_tex : *skt* → *string*;
value skt_to_dev : *skt* → *string*;
value skt_to_html : *skt* → *string*;
value skt_raw_to_deva : *skt* → *string*;
value skt_raw_strip_to_deva : *skt* → *string*;
value skt_to_anchor : *skt* → *string*;
value raw_sanskrit_word : *skt* → *Word.word*;
value sanskrit_word : *skt* → *Word.word*;
value rev_stem_skt : *skt* → *Word.word*;
value normal_stem : *skt* → *Word.word*;
value clean_up : *skt* → *skt*;
value normal_stem_skt : *skt* → *string*;
value code_skt_ref : *skt* → *Word.word*;
value code_skt_ref_d : *skt* → *Word.word*;
value decode_skt : *Word.word* → *skt*;
value read_sanskrit : (*string* → *Word.word*) → *string* → *list Word.word*;
value read_raw_sanskrit : (*string* → *Word.word*) → *string* → *list Word.word*;

Module Sanskrit

The Sanskrit lexical processor

open *Skt_lexer*;

type *skt* = *string*
and *encoding* = *string* → *list int*
;

Recognize a Sanskrit sentence as either a pada or a sloka

type *pada* = *list skt*
and *sloka* = *list pada*
;
(∗ Dangerous - keeps the accent and chars + - dollar ∗)
value string_of_skt s = *s* (∗ coercion *skt* → *string* ∗)
;
(∗ Unsafe - debugging mostly, but also *Print_html.print_skt_px_ac* ∗)

```
value skt_of_string s = s (* coercion string → skt *)
;
value aa_preverb = "aa"
and privative p = List.mem p [ "a"; "an#1" ] (* privative prefixes *)
;
(* Sanskrit word used in computations *)
(* Fragile: assumes fixed entry in lexicon *)
value i_root = "i" (* Subst.record_ifc2 *)
and ita_part = "ita" (* id *)
and dagh_root = "dagh" (* id *)
and daghna_part = "daghna" (* id - accent needed *)
and arcya_absolutive = "arcya" (* Subst.record_noun_gen *)
;
module Gramskt = Camlp4.PreCast.MakeGram Skt_lexer
;
open Skt_lexer.Token
;
(* Entry points *)
value skt = Gramskt.Entry.mk "skt"
and skt1 = Gramskt.Entry.mk "skt1"
and pada = Gramskt.Entry.mk "pada"
and sloka_line = Gramskt.Entry.mk "sloka_line"
and sloka = Gramskt.Entry.mk "sloka"
and sanscrit = Gramskt.Entry.mk "sanscrit"
and prefix = Gramskt.Entry.mk "prefix"
and skt_list = Gramskt.Entry.mk "skt_list"
and prefix_list = Gramskt.Entry.mk "prefix_list"
;
EXTEND Gramskt
  skt : (* chunk of Sanskrit letters in Velthuis romanisation *)
    [ [ id = IDENT; "_"; s = skt → id ^ "_" ^ s (* hiatus (underscore) *)
      | id = IDENT; "#"; n = INT → id ^ "#" ^ n (* homonym index *)
      | id = IDENT → id (* possible avagraha is initial quote *)
      | n = INT → n (* numerals eg -tama *)
    ] ];
  skt1 :
    [ [ s = skt; `EOI → s ] ];
  pada : (* non-empty list of chunks separated by blanks *)
    [ [ el = LIST1 skt → el ] ];
  sanscrit :
```

```
    [ [ p = pada; "|"; "|" → [ p ]
      | p = pada; "|"; `EOI → [ p ]
      | p = pada; "|"; sl = sanskrit → [ p :: sl ]
      | p = pada; "!"; sl = sanskrit → [ p :: sl ] (* for voc and interj *)
      | p = pada; `EOI → [ p ]
      | `EOI → failwith "Empty sanskrit input"
    ] ] ;
  skt_list :
    [ [ el = LIST1 skt SEP ","; `EOI → el ] ] ;
END
;
value trad_string entry t =
  try Gramskt.parse_string entry Loc.ghost t with
  [ Loc.Exc_located loc e → do
      { Format.eprintf "\nIn string \"%s\", at location %s :\n%!"
                         t (Loc.to_string loc)
      ; raise e
      }
  ]
;
value trad_skt = trad_string skt1
  and trad_sanskrit = trad_string sanskrit
  and trad_skt_list = trad_string skt_list
;
value maha_epic = "Mahaabhaarata" (* for Print_html *)
  and rama_epic = "Raamaaya.na"
;
value skt_to_tex = Transduction.skt_to_tex; (* romanisation Tex diacritics *)
value skt_to_dev = Transduction.skt_to_dev; (* devanagari devnag *)
value skt_to_html = Transduction.skt_to_html; (* romanisation *)
```

Encoding functions skt -¿ word

```
value raw_sanskrit_word = Transduction.code_raw; (* no normalisation no accent*)
value sanskrit_word = Encode.code_string; (* normalisation *)
value skt_raw_to_deva = Encode.skt_raw_to_deva; (* devanagari unicode *)
value skt_raw_strip_to_deva = Encode.skt_raw_strip_to_deva; (* idem *)
value skt_to_anchor = Encode.anchor; (* hypertext anchor encoding *)
value rev_stem_skt = Encode.rev_stem; (* normalised revword *)
value normal_stem = Encode.normal_stem; (* normalised stem as word *)
```

Cleaning up by removing accents - used in *Print_dict*

```
value clean_up s = Canon.decode (Transduction.code_raw s)
;
(* Following used in Print_dict and Subst – ought to disappear *)
value normal_stem_skt = Encode.normal_stem_str; (* normalised stem as string *)
value code_skt_ref = Encode.code_skt_ref;
value code_skt_ref_d = Encode.code_skt_ref_d;
value decode_skt = Canon.decode
;
open Padapatha (* padapatha sanskrit_chunk *)
;
value sanskrit_sentence strm =
  try Gramskt.parse sanscrit Loc.ghost strm with
  [ Loc.Exc_located loc Exit -> raise (Encode.In_error "Exit")
  | Loc.Exc_located loc (Error.E msg)
    -> raise (Encode.In_error ("(Lexical) " ^ msg))
  | Loc.Exc_located loc (Stream.Error msg)
    -> raise (Encode.In_error ("(Stream) " ^ msg))
  | Loc.Exc_located loc (Failure s) -> raise (Encode.In_error s)
  | Loc.Exc_located loc ex -> raise ex
  ]
;
(* No padapatha processing, each chunk is assumed to be in terminal sandhi already. But
normalizes away anusvara, contrarily to its name *)
(* encode is raw_sanskrit_word, raw_sanskrit_word_KH, etc. *)
value read_raw_skt_stream encode strm =
  let process = List.map encode in
  match sanskrit_sentence strm with
  [ [ l ] -> process l
  | lines -> List.fold_right concat lines []
             where concat line lines = process line @ lines
  ]
;
value read_processed_skt_stream encode strm =
  let process = padapatha (sanskrit_chunk encode) in
  match sanskrit_sentence strm with
  [ [ l ] -> process l
  | lines -> List.fold_right concat lines []
             where concat line lines = process line @ lines
  ]
;
```

Now general readers with encoding parameter of type *string* → *word*
read_sanskrit : *encoding* → *string* → *list word*
Assumes sandhi is not undone between chunks - spaces are not significant
Generalizes *read_VH False* to all transliterations

```
value read_sanskrit encode str = (* encode : string → word *)
  read_processed_skt_stream encode (Stream.of_string str)
;
(* Assumes sandhi is undone between chunks (partial padapatha) *)
(* Generalizes read_VH True to all transliterations *)
value read_raw_sanskrit encode str = (* encode : string → word *)
  read_raw_skt_stream encode (Stream.of_string str)
;
```

Module Test_stamp

Tests consistency of data version of *Heritage_Resources*
with program version of *Heritage_Platform*

```
value check_data_version () =
  let resources_version_file = Paths.skt_resources_dir ^ "DATA/version.rem" in
  let (data_format_version, data_version) =
       (Gen.gobble resources_version_file : (int × string)) in
   if Control.data_format_version > data_format_version then do
       { print_string "Your linguistic data are stale\n"
       ; print_string "Your must install a recent version of Heritage_Resources\n"
       ; exit 1
       } else
   if Control.data_format_version < data_format_version then do
      { print_string "Your Heritage_Platform installation is too old\n"
      ; print_string "Your must re-install a more recent version\n"
      ; print_string "consistent with data format "
      ; print_int data_format_version
      ; print_string "\n"
      ; exit 1
      } else
   if data_version = Version.version then () else do
      { print_string "Warning: this platform release "
      ; print_string ("assumes version " ^ Version.version)
      ; print_string " of Heritage_Resources\n"
```

```
      ; print_string "while the currently installed Heritage_Resources at "
      ; print_string resources_version_file
      ; print_string (" has version " ^ data_version)
      ; print_string "\n You should consider updating to recent versions\n"
      }
;
try check_data_version () with
[ Sys_error m → failwith ("Wrong structure of Heritage_Resources " ^ m) ]
;
```

Interface for module Dir

Directory operations
subdirs dir returns the list of subdirectories of *dir*. The order of the returned list is unspecified. Raise *Sys_error* when an operating system error occurs.

```
value subdirs : string → list string
;
```

(∗ *files_with_ext ext dir* returns the list of files in *dir* with the extension *ext* (e.g. "txt"). The order of the returned list is unspecified. Raise *Sys_error* when an operating system error occurs. ∗)

```
value files_with_ext : string → string → list string
;
```

(∗ *split path* splits *path* into substrings corresponding to the subdirectories of *path*. ∗)

```
value split : string → list string
;
```

Module Dir

Return the list of files in the given directory with their absolute name.

```
value abs_files dir =
  let files = Array.to_list (Sys.readdir dir) in
  List.map (Filename.concat dir) files
;
value basenames files = List.map Filename.basename files
;
value subdirs dir =
  let subdirs = List.filter Sys.is_directory (abs_files dir) in
  subdirs |> basenames
```

```
;
value file_with_ext ext file =
  ¬ (Sys.is_directory file) ∧ Filename.check_suffix file ("." ^ ext)
;
value files_with_ext ext dir =
  let files = List.filter (file_with_ext ext) (abs_files dir) in
  files |> basenames
;
value split path = Str.split (Str.regexp_string Filename.dir_sep) path
;
```

Module Paths

Do not edit by hand - generated by configuration script - see main Makefile

```
value platform = "Station"
and default_transliteration = "VH"
and default_lexicon = "SH"
and default_display_font = "roma"
and zen_install_dir = "/Users/huet/ML/ZEN/Zen/ML/"
and skt_install_dir = "/Users/huet/Sanskrit/Heritage_Platform/"
and skt_resources_dir = "/Users/huet/Sanskrit/Heritage_Resources/"
and public_skt_dir = "/Library/WebServer/Documents/SKT/"
and skt_dir_url = "/SKT/"
and server_host = "127.0.0.1"
and remote_server_host = "https://sanskrit.inria.fr/"
and cgi_dir_url = "/cgi-bin/SKT/"
and cgi_index = "sktindex.cgi"
and cgi_indexd = "sktsearch.cgi"
and cgi_lemmatizer = "sktlemmatizer.cgi"
and cgi_reader = "sktreader.cgi"
and cgi_parser = "sktparser.cgi"
and cgi_tagger = "skttagger.cgi"
and cgi_decl = "sktdeclin.cgi"
and cgi_conj = "sktconjug.cgi"
and cgi_sandhier = "sktsandhier.cgi"
and cgi_graph = "sktgraph.cgi"
and cgi_user_aid = "sktuser.cgi"
and cgi_corpus_manager = "sktcorpus.cgi"
and cgi_save_corpus = "savecorpus.cgi"
```

```
and cgi_mkdir_corpus = "mkdircorpus.cgi"
and mouse_action = "CLICK";
```

Module Data

module Data html = struct
Absolute paths on development site

```
value resources name = Paths.skt_resources_dir ^ name ^ "/"
;
(* Read-only resources *)
value heritage_dir = resources "DICO"
and data_dir = resources "DATA"
;
(* Contains the locally computed transducers databases *)
value local_data_dir = "DATA/"
;
(* Local resources *)
value top_dev_dir name = Paths.skt_install_dir ^ name ^ "/"
;
value dico_dir = top_dev_dir "DICO" (* augments local copy of DICO dynamically *)
;
(* Absolute paths of target server *)
value top_site_dir name = Paths.public_skt_dir ^ name ^ "/"
;
value public_dico_dir = top_site_dir "DICO" (* hypertext dictionary *)
and public_data_dir = top_site_dir "DATA" (* linguistic data for cgis *)
and corpus_dir = top_site_dir "CORPUS" (* Corpus tree *)
;

value data name = data_dir ^ name
and local_data name = local_data_dir ^ name
and dico_page name = dico_dir ^ name
and public_data name = public_data_dir ^ name
and public_dico_page name = public_dico_dir ^ name
;
value public_entries_file = public_dico_page "entries.rem"
(* created by make releasedata, read by indexer *)
and public_dummies_file = public_dico_page "dummies.rem"
(* created by make releasedata, read by indexerd *)
```

```
;
value sandhis_file = data "sandhis.rem"
and sandhis_pv_file = data "sandhis_pv.rem"
and sandhis_ph_file = data "sandhis_ph.rem"
and public_sandhis_id_file = data "sandhis_id.rem"
and automaton_stats = data "automaton.txt"
    (* text file created by make_automaton - stats *)
;
value nouns_file = data "nouns.rem"
    (* created by make_nouns, read by Print_inflected.read_nouns, used by Make_transducers.make_tr
to generate the transducers, copied in public_nouns_file by make releasecgi for use by cgis
*)
and nouns2_file = data "nouns2.rem" (* same in mode Simple *)
and pronouns_file = data "pronouns.rem"
    (* created by make_nouns, read by Print_inflected.read_pronouns *)
and roots_infos_file = data "roots_infos.rem"
    (* created by Print_dict.postlude, read by Make_roots.make_roots *)
and roots_usage_file = data "roots_usage.rem"
    (* created by Print_html.postlude, read by Dispatcher.roots_usage *)
and verblinks_file = data "verblinks.rem"
    (* created by Print_dict.postlude calling Roots.collect_preverbs *)
    (* read by Print_html, Make_preverbs *)
    (* copied in public_verblinks_file *)
and lexical_kridantas_file = data "lexical_kridantas.rem"
    (* created by Print_dict.postlude, read by Make_roots.roots_to_conjugs *)
and unique_kridantas_file = data "unique_kridantas.rem"
    (* created by Make_roots.roots_to_conjugs *)
and roots_file = data "roots.rem"
    (* created by make_roots, read by reader, tagger & indexer *)
and peris_file = data "peris.rem"
and lopas_file = data "lopas.rem"
and parts_file = data "parts.rem"
and partvocs_file = data "partvocs.rem"
and lopaks_file = data "lopaks.rem"
and preverbs_file = data "preverbs.rem"
    (* created by make_preverbs, read by make_inflected *)
and preverbs_textfile trans = data (trans ^ "_preverbs.txt")
    (* created by make_preverbs for documentation *)
and iics_file = data "iics.rem"
    (* created by make_nouns, copied in public_iics_file by make install, read by make_automaton
```

invoked from DATA/Makefile ∗)
and *iics2_file* = *data* "iics2.rem" (∗ same in mode Simple ∗)
and *iifcs_file* = *data* "iifcs.rem" (∗ iic stems of ifc nouns ∗)
and *vocas_file* = *data* "voca.rem" (∗ created by *make_nouns* etc. ∗)
and *invs_file* = *data* "invs.rem" (∗ created by *make_nouns* etc. ∗)
and *piics_file* = *data* "piics.rem" (∗ created by *make_roots* etc. ∗)
and *ifcs_file* = *data* "ifcs.rem" (∗ created by *make_nouns* etc. ∗)
and *ifcs2_file* = *data* "ifcs2.rem" (∗ same in mode Simple ∗)
and *avyayais_file* = *data* "avyayais.rem" (∗ iic stems of avyayiibhava cpds ∗)
and *avyayafs_file* = *data* "avyayafs.rem" (∗ ifc stems of avyayiibhava cpds ∗)
and *iivs_file* = *data* "iivs.rem" (∗ created by *make_roots* etc. ∗)
and *auxis_file* = *data* "auxi.rem" (∗ created by *make_roots* etc. ∗)
and *auxiinvs_file* = *data* "auxiinv.rem" (∗ created by *make_roots* etc. ∗)
and *auxiks_file* = *data* "auxik.rem" (∗ created by *make_roots* etc. ∗)
and *auxiicks_file* = *data* "auxiick.rem" (∗ created by *make_roots* etc. ∗)
and *indecls_file* = *data* "indecls.rem" (∗ created by *make_roots* etc. ∗)
and *absya_file* = *data* "absya.rem" (∗ created by *make_roots* etc. ∗)
and *abstvaa_file* = *data* "abstvaa.rem" (∗ created by *make_roots* etc. ∗)
and *inftu_file* = *data* "inftu.rem" (∗ created by *make_roots* etc. ∗)
and *kama_file* = *data* "kama.rem" (∗ created by *make_nouns* etc. ∗)

The transducers files, made by *make_transducers*

and *transducers_file* = *local_data* "transducers.rem" (∗ transducers Complete ∗)
and *transducers_file2* = *local_data* "transducers2.rem" (∗ transducers Simple ∗)

and *mw_exc_file* = *data* "mw_exceptions.rem" (∗ for MW indexing ∗)
and *mw_index_file* = *data* "mw_index.rem"
and *guess_auto* = *data* "guess_index.rem"
;
(∗ Next are the inflected forms banks, read at cgi time by *Lexer.load_morphs* ∗)
value *public_nouns_file* = *public_data* "nouns.rem"
and *public_nouns2_file* = *public_data* "nouns2.rem"
and *public_pronouns_file* = *public_data* "pronouns.rem"
and *public_preverbs_file* = *public_data* "preverbs.rem"
and *public_roots_file* = *public_data* "roots.rem"
and *public_peris_file* = *public_data* "peris.rem"
and *public_lopas_file* = *public_data* "lopas.rem"
and *public_lopaks_file* = *public_data* "lopaks.rem"
and *public_parts_file* = *public_data* "parts.rem"
and *public_partvocs_file* = *public_data* "partvocs.rem"
and *public_iics_file* = *public_data* "iics.rem"

```
and public_iics2_file = public_data "iics2.rem"
and public_piics_file = public_data "piics.rem"
and public_ifcs_file = public_data "ifcs.rem"
and public_ifcs2_file = public_data "ifcs2.rem"
and public_iivs_file = public_data "iivs.rem"
and public_avyayais_file = public_data "avyayais.rem" (* iic avyayiibhavas *)
and public_avyayafs_file = public_data "avyayafs.rem" (* ifc avyayiibhavas *)
and public_auxis_file = public_data "auxi.rem"
and public_auxiinvs_file = public_data "auxiinv.rem"
and public_auxiks_file = public_data "auxik.rem"
and public_auxiicks_file = public_data "auxiick.rem"
and public_iifcs_file = public_data "iifcs.rem"
and public_vocas_file = public_data "voca.rem"
and public_invs_file = public_data "invs.rem"
and public_inde_file = public_data "indecls.rem"
and public_absya_file = public_data "absya.rem"
and public_abstvaa_file = public_data "abstvaa.rem"
and public_inftu_file = public_data "inftu.rem"
and public_kama_file = public_data "kama.rem"
and public_stems_file = public_data "stems.rem"
and public_roots_infos_file = public_data "roots_infos.rem"
and public_roots_usage_file = public_data "roots_usage.rem"
and public_lexical_kridantas_file = public_data "lexical_kridantas.rem"
and public_unique_kridantas_file = public_data "unique_kridantas.rem"
and public_verblinks_file = public_data "verblinks.rem"
and public_mw_exc_file = public_data "mw_exceptions.rem"
and public_mw_index_file = public_data "mw_index.rem"
and public_guess_auto = public_data "guess_index.rem"
(* The segmenting transducers, read at cgi time by Load_transducers *)
and public_transducers_file = public_data "transducers.rem" (* Complete *)
and public_transducers_file2 = public_data "transducers2.rem" (* Simple *)
(* The cached supplementary nouns dictionary *)
and public_cache_file = public_data "cache.rem" (* cache genders *)
and public_cachei_file = public_data "cachei.rem" (* cache iics *)
and public_cache_txt_file = public_data "cache.txt" (* master cache *)
and public_trans_cache_file = public_data "transca.rem"
and public_trans_cachei_file = public_data "transcai.rem"
;
```

Module Index

Indexing utility

```
extract_zip : zipper → word

value extract_zip = extract_zip_acc []
where rec extract_zip_acc suff = fun
  [ Top → suff
  | Zip (_, _, n, _, up) → extract_zip_acc [ n :: suff ] up
  ]
;
exception Last of string
;
value rec previous b left z = match left with
  [ [] → if b then extract_zip z
         else match z with
              [ Top → failwith "entry␣'a'␣missing"
              | Zip (b', l', _, _, z') → previous b' l' z'
              ]
  | [ (n, t) :: _ ] → let w1 = extract_zip z
                      and w2 = last_trie t in
                      w1 @ [ n :: w2 ]
  ]
;
(* Vicious hack to return first homonym if it exists - ugly *)
value next_trie_homo = next_rec []
  where rec next_rec pref = fun
  [ Trie (b, l) →
    if b then List.rev pref
    else try let _ = List.assoc 51 l (* looking for homonym #1 *) in
             List.rev [ 51 :: pref ] (* found - we know it is accepting *)
         with (* no homonym - we keep looking for first accepting suffix *)
         [ Not_found → match l with
           [ [] → failwith "next" (* should not happen if trie in normal form *)
           | [ (n, u) :: _ ] → next_rec [ n :: pref ] u
           ]
         ]
  ]
;
value escape w = raise (Last (Canon.decode w))
;
```

(* *search* : (*w* : *word*) → (*t* : *trie*) → (*string* × *bool* × *bool*) *)
(* Assert : *t* is not *Empty* *)
(* *search w t* returns either the first member of *t* with *w* as initial substring with a boolean *exact* indicating if the match is exact and another one *homo* marking homonymy or else raises *Last s* with *s* the last member of *t* less than *w* in lexicographic order. Beware. Do not change this code if you do not understand fully the specs. *)

```
value search w t = access w t Trie.Top
  where rec access w t z = match w with
     [ [] → let w1 = extract_zip z
               and w2 = next_trie_homo t in
               let exact = w2 = []
               and homo = w2 = [ 51 ] in
               (Canon.decode (w1 @ w2), exact, homo)
     | [ n :: rest ] → match t with
        [ Trie (b, arcs) → match arcs with
          [ [] → if b then escape (extract_zip z)
                    else failwith "Empty␣trie"
          | _ → let (left, right) = List2.zip n arcs in
                   match right with
            [ [] → let w1 = extract_zip z and w2 = last_trie t in
                      escape (w1 @ w2)
            | [ (m, u) :: upper ] →
               if m = n then access rest u (Zip (b, left, m, upper, z))
               else escape (previous b left z)
            ]
          ]
        ]
     ]
;
value read_entries () =
  (Gen.gobble Data.public_entries_file : trie)
;
value is_in_lexicon word =
  (* Checks whether entry word actually appears in the lexicon, *)
  (* so that a reference URL is generated in the answers or not. *)
  (* NB: not indexed by lexical category *)
  let entries_trie = read_entries () in
  Trie.mem word entries_trie
;
```

Module Phonetics

Notation PrX gives X as the pratyaahaara notation of a set of phonemes
Sanskrit phonology

```
value vowel c = c > 0 ∧ c < 14 (∗ a aa i ii u uu .r .rr .l e ai o au ∗)(∗ Prhac ∗)
and anusvar c = c = 14 (∗ .m : anusvara standing for nasal ∗)
           (∗ —— c=15 candrabindu ∗)
and visarga c = c = 16 (∗ .h ∗)
and consonant c = c > 16 (∗ Prhal ∗)
and phantom c = c < (−1) (∗ -2 -3=*a -4=*i -5=*u -6=*r ∗)
;
(∗ final s assimilated to visarga ∗)
value visarg c = c = 48 (∗ s ∗) ∨ c = 16 (∗ .h ∗)
;
(∗ final r also assimilated to visarga ∗)
value visargor c = visarg c ∨ c = 43 (∗ r ∗)
;
value rec all_consonants = fun
  [ [ c :: rest ] → consonant c ∧ all_consonants rest
  | [] → True
  ]
;
value consonant_initial = fun
  [ [ c :: _ ] → consonant c
  | _ → False
  ]
;
value consonant_starts = fun
  [ [ chunk :: _ ] → consonant_initial chunk
  | _ → False
  ]
;
value monosyllabic = one_vowel
  where rec one_vowel = fun
    [ [] → True
    | [ c :: rest ] → if vowel c then all_consonants rest
                      else one_vowel rest
    ]
;
value short_vowel = fun
```

```
  [ 1 | 3 | 5 | 7 | 9 → True (* .l included *)
  | _ → False
  ]
and long_vowel = fun
  [ 2 | 4 | 6 | 8 → True
  | _ → False
  ]
;
value avarna c = c < 3 (* a aa *)
and ivarna c = c = 3 ∨ c = 4 (* i ii *)
and uvarna c = c = 5 ∨ c = 6 (* u uu *)
and rivarna c = c = 7 ∨ c = 8 (* .r .rr *)
;
value not_a_vowel c = vowel c ∧ ¬ (avarna c) (* c¿2 and c¡14 *)
and is_aa c = c = 2
and is_i_or_u c = c = 3 ∨ c = 5
and not_short_vowel c = vowel c ∧ ¬ (short_vowel c)
;
(* segments a word as a list of syllables - Unused *)
value syllables = syllables_rec [] []
  where rec syllables_rec accu_syl accu_pho = fun
  [ [ c :: rest ] →
      if vowel c then
         let new_syl = List.rev [ c :: accu_pho ] in
         syllables_rec [ new_syl :: accu_syl ] [] rest
      else syllables_rec accu_syl [ c :: accu_pho ] rest
  | [] → List.rev accu_syl
  ]
;
(* multi-consonant - used in Verbs for reduplicating aorist *)
(* we call (mult w) with w starting with a consonant *)
value mult = fun
  [ [ _ (* assumed consonant *) :: [ c :: _ ] ] → consonant c
  | _ → False
  ]
;
(* lengthens a vowel *)
value long c =
  if short_vowel c then
     if c = 9 then failwith "No long .l" else c + 1
```

```
  else if vowel c then c
       else failwith "Bad arg to long"
(* shortens a vowel *)
and short c =
  if long_vowel c then c - 1
  else if vowel c then c
       else failwith "Bad arg to short"
;
(* lengthens the final vowel of a (reverse) stem *)
value lengthen = fun
  [ [ v :: r ] → [ long v :: r ]
  | [] → failwith "Bad arg to lengthen"
  ]
;
(* homophonic vowels *)
value savarna v1 v2 = v1 < 9 ∧ v2 < 9 ∧ (long v1 = long v2)
;
(* special version where c may be a phantom for Sandhi *)
value savarna_ph v c = (vowel c ∧ savarna v c) ∨ (c = (-3) ∧ avarna v)
;
value velar c = c > 16 ∧ c < 22 (* gutturals : k kh g gh f *)
and palatal c = c > 21 ∧ c < 27 (* palatals : c ch j jh  n *)
and lingual c = c > 26 ∧ c < 32 (* cerebrals : .t .th .d .dh .n *)
and dental c = c > 31 ∧ c < 37 (* dentals : t th d dh n *)
and labial c = c > 36 ∧ c < 42 (* labials : p ph b bh m *)
and semivowel c = c > 41 ∧ c < 46 (* semi vowels : y r l v Prya.n *)
and sibilant c = c > 45 ∧ c < 49 (* sibilants : z .s s Przar *)
and aspirate c = c = 49 (* h *)
;
value stop c = c > 16 ∧ c < 42
;
value nasal c =
     c = 21 (* f *) ∨ c = 26 (*  n *) ∨ c = 31 (* .n *)
  ∨ c = 36 (* n *) ∨ c = 41 (* m *) ∨ anusvar c (* Pr nam *)
;
value n_or_f c = c = 21 (* f *) ∨ c = 36 (* n *)
;
value homonasal c = (* nasal homophonic to given consonant *)
  if consonant c then
  if velar c then 21 (* f *) else
```

```
  if palatal c then 26 (*  n *) else
  if lingual c then 31 (* .n *) else
  if dental c then 36 (* n *) else
  if labial c then 41 (* m *)
                else 14 (* .m *)
  else failwith "Non␣consonant␣arg␣to␣homonasal"
;
(* vowel modifiers = anusvaara 14, candrabindu 15 and visarga 16 *)
value vowel_mod c = c > 13 ∧ c < 17
;
(* eliminate duplicate consonant in test for prosodically long in Verbs *)
value contract = fun
  [ [ c :: r ] →
    let l = match r with
        [ [ c' :: r' ] → if c = c' then r' else r
        | [] → []
        ] in [ c :: l ]
  | [] → []
  ]
;
value voiced = fun (* voices previous phoneme with homophone *)
  [ 17 → 19 (* k -¿ g *)
  | 27 → 29 (* .t -¿ .d *)
  | 32 → 34 (* t -¿ d *)
  | 37 → 39 (* p -¿ b *)
  (* next 6 not used by sandhi *)
  | 18 → 20 (* kh -¿ gh *)
  | 22 → 24 (* c -¿ j *)
  | 23 → 25 (* ch -¿ jh *)
  | 28 → 30 (* .th -¿ .dh *)
  | 33 → 35 (* th -¿ dh *)
  | 38 → 40 (* ph -¿ bh *)
  | c → c
  ]
;
value voiced_consonant c = (* Prjhaz *)
  List.mem c [ 19; 20; 24; 25; 29; 30; 34; 35; 39; 40 ]
and mute_consonant c =(* Prkhay *)
  List.mem c [ 17; 27; 32; 37; 18; 22; 23; 28; 33; 38 ]
;
```

```
value is_voiced c = (* voiced phonemes *)
  vowel c ∨ voiced_consonant c ∨ List.mem c [ 42; 43; 45 ] (* y r v *)
;
(* Next 5 functions used in Sanskrit.adjust *)
value turns_t_to_j c = List.mem c [ 24; 25 ] (* j jh *)
;
value turns_n_to_palatal c = palatal c ∨ c = 46 (* z *)
;
value avagraha c = (c = −1) (* elided initial a after a.h which turns to o *)
;
value elides_visarg_aa c =
  voiced_consonant c ∨ nasal c ∨ semivowel c ∨ aspirate c
;
value turns_visarg_to_o c = elides_visarg_aa c ∨ avagraha c
;
value guna = fun
  [ 1 → [ 1 ] (* a is its own guna *)
  | 2 → [ 2 ] (* aa is its own guna and vriddhi *)
  | 3 | 4 → [ 10 ] (* e *)
  | 5 | 6 → [ 12 ] (* o *)
  | 7 | 8 → [ 1; 43 ] (* ar *)
  | 9 → [ 1; 44 ] (* al *)
  | c → [ c ]
  ]
;
value vriddhi = fun
  [ 1 | 2 → [ 2 ] (* aa *)
  | 3 | 4 | 10 | 11 → [ 11 ] (* ai *)
  | 5 | 6 | 12 | 13 → [ 13 ] (* au *)
  | 7 | 8 → [ 2; 43 ] (* aar *)
  | 9 → [ 2; 44 ] (* aal *)
  | c → [ c ]
  ]
;
```

Macdonnel §125 - condition for root of gana 1 to take guna of its stem

```
value gunify = fun (* arg word is reversed stem *)
  [ [ v :: _ ] when vowel v → True
  | [ _ :: [ v :: _ ] ] when short_vowel v → True
  | _ → False
```

```
  ]
;
(* Augment computation *)
value augment x = (* arg is first letter of root *)
  if vowel x then vriddhi x
  else if x = 23 (* ch *) then [ 1; 22; 23 ] (* cch *)
  else if x > 16 ∧ x < 50 then [ 1; x ] (* a prefix of consonant *)
  else failwith "Phonetics.augment"
;
value aug = fun (* augment last phoneme of word *)
  [ [ c :: word ] → augment c @ word
  | [] → failwith "Empty␣stem␣(aug)"
  ]
;
value light = fun (* light roots end in short vowel Pan6,1,69 *)
   [ [] → failwith "light"
   | [ c :: _ ] → short_vowel c
   ]
;
(* For absolutives of roots gana 10 *)
value light_10 = fun (* light roots end in short vowel Pan1,4,11 *)
   [ [] → failwith "light_10"
   | [ c :: r ] → if vowel c then False (* ? *) else match r with
        [ [] → failwith "light_10_1"
        | [ v :: _ ] → short_vowel v
        ]
   ]
;
(* Needed by Verbs.record_part_m_th for proper retroflexion of aatmanepada participles in
-maana - eg kriyamaa.na *)
(* all erase last phoneme - used in denominative verbs *)
value trunc_a = fun
  [ [ 1 :: w ] → w
  | _ → failwith "trunc_a"
  ]
and trunc_aa = fun
  [ [ 2 :: w ] → w
  | _ → failwith "trunc_aa"
  ]
and trunc_u = fun
```

```
  [ [ 5 :: w ] → w
  | _ → failwith "trunc_u"
  ]
;
value trunc = fun
  [ [ _ :: w ] → w
  | w → failwith ("trunc␣" ^ Canon.rdecode w)
  ]
;
(* Unused (* Stem has short vowel in last syllable *) value rec brief = fun [] → failwith "Stem␣with␣no␣
| [ c ] → if vowel c then short_vowel c else failwith "Stem␣with␣no␣vowel␣(brief)"
| [ c :: r ] → if vowel c then short_vowel c else brief r ; (* Sandhi of preverb aa- *) (*
Unused, but simulated by Inflected - related to asandhi below. *) value mkphantom = fun
(* arg is vowel not avarna and not .rr or .l *) 1 | 2 → [ −3 ] (× aa − a ×) | 3 | 4 →
[ −4 ] (× aa − i ×) | 5 | 6 → [ −5 ] (× aa − u ×) | 7 → [ −6 ] (× aar ×) | 10 |
11 → [ 11 ] (× ai ×) | 12 | 13 → [ 13 ] (× au ×) | _ → failwith "mkphantom" ; *)
(* Sandhi of a and aa with initial vowel (or phantom) (for Sandhi) *)
(* arg is (vowel not avarna and not .rr or .l) or -2,-4,-5,-6,-7,-8 *)
value asandhi = fun
  [ 3 | 4 | −4 | −7 → [ 10 ] (* e for i, ii and e-phantoms *i *I *)
  | 5 | 6 | −5 | −8 → [ 12 ] (* o for u, uu and o-phantoms *u *U *)
  | 7 → [ 1; 43 ] (* ar *)
  | −6 → [ 2; 43 ] (* aar *)
  | 123 → [ 2; 22; 23 ] (* aacch *)
  | 10 | 11 → [ 11 ] (* ai *)
  | 12 | 13 → [ 13 ] (* au *)
  | −2 → [] (* amuissement *)
  | _ → failwith "asandhi"
  ]
;
value vowel_or_phantom c = vowel c ∨ phantom c
;
(* Tests whether a word starts with a phantom phoneme (precooked aa-prefixed finite or
participial or infinitive or abs-ya root form) Used by Morpho, Inflected. Copied in Dispatcher.
*)
value phantomatic = fun
  [ [ c :: _ ] → c < (−2) ∨ c = 123
  | _ → False
  ]
(* Amuitic forms start with -2 = - which elides preceding -a or -aa from Pv *)
```

```
and amuitic = fun [ [ -2 :: _ ] → True | _ → False ]
;
```

Following 4 functions are used in stem computations in Verbs.
For m.rj-like verbs (Whitney§219-a) Panini8,2,36 "bhraaj" "m.rj" "yaj1" "raaj1" "vraj" "s.rj1" "bh.rjj" replace phoneme j=24 by j'=124 with sandhi j'+t = .s.t (j' is j going to z)

```
value mrijify stem = match stem with
  [ [ 24 :: r ] → [ 124 :: r ]
  | _ → failwith ("mrijify" ^ Canon.rdecode stem)
  ]
;
(* For "duh"-like verbs (Whitney§222) "dah" "dih" "duh1" Panini8,2,32 optionnellement "druh1" "muh" "snuh1" "snih1" Panini8,2,33 replace phoneme h=49 by h'=149 with sandhi h'+t = gdh (h' is h going to gh) ( whereas normal h goes to .dh like pp(lih)=lii.dha) *)
value duhify stem = match stem with
  [ [ 49 :: r ] → [ 149 :: r ]
  | _ → failwith ("duhify␣" ^ Canon.rdecode stem)
  ]
;
(* For "nah"-like verbs - h" is h going to dh. Replace phoneme h=49 by h"=249 with sandhi h"+t = ddh ) *)
value nahify stem = match stem with
  [ [ 49 :: r ] → [ 249 :: r ]
  | _ → failwith ("nahify␣" ^ Canon.rdecode stem)
  ]
;
```

Aspiration of initial consonant of root stems ending in aspirate. The syllabic loop is necessary for e.g. druh -¿ dhruk. See Whitney§155.

```
value syll_decomp = fun
  [ [ c :: rest ] → decomp_rec [] c rest
      where rec decomp_rec cs c w = match w with
        [ [ c' :: rest' ] → if consonant c' then decomp_rec [ c :: cs ] c' rest'
                                 else (cs, c, w)
        | [] → (cs, c, w)
        ]
  | [] → failwith "syll_decomp"
  ]
;
value mk_aspirate w = (* c-cs-vow is the syllable ending in vow *)
  let (cs, c, rest) = syll_decomp w in
```

```
  let aspc = match c with
        [ 19 (* g *) → 20 (* gh *)
        | 34 (* d *) → 35 (* dh *) (* e.g. duh → dhuk *)
        | 39 (* b *) → 40 (* bh *) (* e.g. budh → bhut *)
        | _ → c (* e.g. vṛdh samidh *)
        ] in
  List2.unstack cs [ aspc :: rest ]
;
value asp = fun
  [ [ vow :: rest ] when vowel vow → [ vow :: mk_aspirate rest ]
  | _ → failwith "Penultimate␣not␣vowel"
  ]
;
(* final form of a pada *)
(* Warning - finalize does NOT replace s or r by visarga, and fails on visarga *)
value finalize rstem = match rstem with
  [ [] → []
  | [ c :: rest ] → match c with
          [ 17 (* k *) (* first permitted finals *)
          | 18 (* kh *)
          | 21 (* ṅ *)
          | 27 (* ṭ *)
          | 28 (* ṭh *)
          | 31 (* ṇ *)
          | 32 (* t *) (* e.g. marut, viśvajit *)
          | 33 (* th *)
          | 36 (* n *)
          | 37 (* p *)
          | 38 (* ph *)
          | 41 (* m *)
          | 44 (* l *) (* l needed for pratyāhāra hal *)
          | 45 (* v *) (* diiv2 *)
          | 43 (* r *) (* no visarga to keep distinction r/s for segmentation *)
          | 48 (* s *) → rstem (* but sras -¿ srat ? *)
          | 19 (* g *)
          | 22 (* c *)
          | 23 (* ch *)
          | 24 (* j *) (* e.g. bhiṣaj; bhuj; asṛj -yuj *)
          | 25 (* jh *) → match rest with
              [ [ 26 (* ñ *) :: ante ] → [ 21 (* ṅ *) :: ante ]
```

```
          | [ 24 (* j *) :: ante ] | [ 22 (* c *) :: ante ]
               → [ 27 (* ṭ *) :: ante ] (* majj bh.rjj pracch *)
          | [ 21 (* ṅ *) :: _ ] → rest
          | _ → [ 17 (* k *) :: rest ] (* but sometimes ṭ - eg devej *)
          ]
     | 20 (* gh *) → [ 17 (* k *) :: asp rest ]
     | 26 (* ñ *) → [ 21 (* ṅ *) :: rest ]
     | 29 (* ḍ *)
     | 30 (* ḍh *) (* e. g. vriḍh *) (* asp? *)
     | 124 (* j' *) → [ 27 (* ṭ *) :: rest ] (* e.g. rāṭ *)
     | 34 (* d *) → [ 32 (* t *) :: rest ] (* e.g. suhṛd *)
     | 35 (* dh *) → [ 32 (* t *) :: asp rest ] (* e.g. budh, vṛdh *)
     | 39 (* b *) → [ 37 (* p *) :: rest ]
     | 40 (* bh *) → [ 37 (* p *) :: asp rest ] (* e.g. kakubh *)
     | 46 (* ś *) → match rest with
         (* .t is default and k exception (Henry, Whitney§145,218) *)
         [ [ 3 :: [ 34 :: _ ] ] (* -diś → -dik *)
         | [ 7 :: [ 34 :: _ ] ] (* -dṛś → -dṛk *)
         | [ 7 :: [ 37 :: [ 48 :: _ ] ] ] (* -spṛś → -spṛk *)
              → [ 17 (* k *) :: rest ]
         | _ → [ 27 (* ṭ *) :: rest ] (* default *)
         (* NB optionally nak Whitney§218a *)
         ]
     | 47 (* ṣ *) → match rest with
         [ [ 7 :: [ 35 :: _ ] ] (* -dhṛṣ → -dhṛk *)
              → [ 17 (* k *) :: rest ] (* Kane §97 *)
         | [ 17 :: ante ] (* -kṣ → -k *)
              → [ 17 (* k *) :: ante ] (* vivik.s Kane §97 but MW: vivi.t *)
         | _ → [ 27 (* ṭ *) :: rest ] (* e.g. dviṣ → dviṭ *)
         ]
     | 49 (* h *) → [ 27 (* ṭ *) :: asp rest ] (* e.g. lih → liṭ *)
     | 149 (* h' *) → [ 17 (* k *) :: asp rest ] (* -duh → -dhuk , impft doh adhok, etc.
*)
     | 249 (* h" *) → [ 32 (* t *) :: asp rest ]
     | c → if vowel c then rstem
            else let s = Canon.rdecode rstem in
                 failwith ("Illegal stem " ^ s ^ " (finalize)")
     ]
  ]
;
```

```
value finalizer root = match root with
  [ [] → []
  | [ c :: rest ] → match c with
       [ 41 (* m *) → [ 36 (* n *) :: rest ] (* Whitney §143a *)
       | _ → finalize root
       ]
  ]
;
(* Used in Nouns.build_root *)
value finalize_r stem = match stem with
  [ [] → []
  | [ c :: rest ] → match c with
       [ 43 (* r *) → match rest with
           [ [ c :: l ] → if short_vowel c (* giir puurbhyas Whitney §245b *)
                          then [ 43 :: [ long c :: l ] ]
                          else stem
           | [] → failwith "Illegal arg r to finalize_r"
           ]
       | 48 (* s *) → match rest with
           [ [ 1 :: [ 45 :: [ 35 :: _ ] ] ] → [ 34 (* t *) :: rest ] (* dhvas *)
           | [ 1 :: [ 45 :: _ ] ] → stem (* suvas *)
           | _ → [ 34 (* t *) :: rest ] (* sras *)
           ]
       | _ → finalize stem
       ]
  ]
;
(* Used in Nouns *)
value bi_consonant rstem = match rstem with
  [ [ c1 :: [ c2 :: _ ] ] → consonant c1 ∧ consonant c2
  | _ → False
  ]
;
```

Caution. Phantom phonemes *a (-3), *i (-4), *u (-5) and *r (-6) are NOT vowels, you should use *vowel_or_phantom* function. Extra fine-grained phonemes j' (124) h' (149) and h" (249) are consonants.

Module Int_sandhi

This module defines internal sandhi operations used in morphology computations The code is complex - do not change without extensive tests.

```
open Phonetics; (* asp finalize visarg *)
open Canon; (* decode rdecode *)

value code str = Encode.code_string str
and mirror = Word.mirror
;
(* Retroflexion of s: for all s in w : l = w1 s w2 with w2 not empty and not starting with r,
look back in w1 skipping c such that retrokeeps(c); if retroacts(c) found then s → ṣ and if
w2 starts with (t, th, n) then this letter becomes retroflex too. *)

value retroacts c =
  c = 17 (* k *) ∨ c = 43 (* r *) ∨ (vowel c ∧ c > 2 ∧ ¬ (c = 9 (* ḷ *)))
;
value retrokeeps c = anusvar c ∨ visarga c (* ḥ *)
;
value rec retros = fun
  [ [] → False
  | [ c :: l ] → retroacts c ∨ (retrokeeps c ∧ retros l)
  ]
;
value rec inspects accu = fun
  [ [] → mirror accu
  | [ c ] → mirror [ c :: accu ]
  | [ 48 (* s *) :: [ 43 (* r *) :: l ] ] → inspects [ 43 :: [ 48 :: accu ] ] l
  | [ 48 (* s *) :: l ] →
    if retros accu then match l with
       [ [] → failwith "Illegal␣arg␣to␣accu"
       | [ 32 (* t *) :: r ] →
             inspects [ 27 (* ṭ *) :: [ 47 (* ṣ *) :: accu ] ] r
       | [ 33 (* th *) :: r ] →
             inspects [ 28 (* ṭh *) :: [ 47 (* ṣ *) :: accu ] ] r
       | [ 36 (* n *) :: r ] →
             inspects [ 31 (* ṇ *) :: [ 47 (* ṣ *) :: accu ] ] r
       | l → inspects [ 47 (* ṣ *) :: accu ] l
       ]
    else inspects [ 48 (* s *) :: accu ] l
  | [ c :: l ] → inspects [ c :: accu ] l
```

```
  ]
;
value retroflexs l = inspects [] l
;
```

(∗ Retroflexion of n: for all n in w : l = w1 n w2 with w2 not empty and starting with *enabling*(*c*), look back in w1 skipping *c*; if *retrokeepn*(*c*) and if *retroactn*(*c*) found then n → ṇ and if w2 starts with n if becomes ṇ too. ∗)

```
value retroactn c = rivarna c ∨ c = 43 (∗ r ∗) ∨ c = 47 (∗ ṣ ∗)
;
value retrokeepn c =
  velar c ∨ labial c ∨ vowel c ∨ anusvar c
            ∨ c = 42 (∗ y ∗) ∨ c = 45 (∗ v ∗) ∨ c = 49 (∗ h ∗)
;
value rec retron = fun
  [ [] → False
  | [ c :: rest ] → retroactn c ∨ (retrokeepn c ∧ retron rest)
  ]
;
```

(∗ uses **P**{8,3,24} ∗)

```
value enabling c = vowel c ∨ c = 36 ∨ c = 41 ∨ c = 42 ∨ c = 45 (∗ n m y v ∗)
;
value retrn c = if c = 36 then 31 (∗ n → ṇ ∗) else c
;
value rec inspectn accu = fun
  [ [] → mirror accu
  | [ c ] → mirror [ c :: accu ]
  | [ 36 (∗ n ∗) :: [ c :: l ] ] →
       if enabling c ∧ retron accu then
           inspectn [ retrn c :: [ 31 (∗ ṇ ∗) :: accu ] ] l
       else inspectn [ 36 :: accu ] [ c :: l ]
  | [ c :: l ] → inspectn [ c :: accu ] l
  ]
;
value retroflexn w = inspectn [] w
;
value ortho_code w = retroflexn (retroflexs w)
;
value ortho s = decode (ortho_code (code s))
;
```

(∗ Test examples ∗)

```
assert (ortho "nisanna" = "ni.sa.n.na");
assert (ortho "pranamati" = "pra.namati");
assert (ortho "parinindati" = "pari.nindati"); (* could be "parinindati" *)
assert (ortho "gurusu" = "guru.su");
```

Exceptions: padas not ortho

```
assert (ortho "visarpati" = "vi.sarpati"); (* should be "visarpati" *)
(* Following due to non-IE origin of stem ? *)
assert (ortho "kusuma" = "ku.suma"); (* but "kusuma" correct *)
assert (ortho "pustaka" = "pu.s.taka"); (* but "pustaka" correct *)
```

Note ortho does not transform final ”s” or ”r” into visarga
Homonasification necessary for present class 7 nk-¿fk
Also (very rare) normalisation of anusvara

```
value homonase c l = match l with
  [ [ 14 (* .m *) :: r ] when stop c → [ c :: [ homonasal c :: r ] ]
  | [ 26 (*  n *) :: r ] when velar c → [ c :: [ 21 (* f *) :: r ] ]
  | _ → [ c :: l ]
  ]
;
(* Local combination of retron and retros, together with homonasification *)
value rec retro_join left = fun
  [ [] → mirror left
  | [ c ] → mirror (homonase c left)
  | [ 36 (* n *) :: [ c :: l ] ] →
      if enabling c ∧ retron left then
           retro_join [ retrn c :: [ 31 (* ṇ *) :: left ] ] l
      else retro_join [ 36 :: left ] [ c :: l ]
  | [ 48 (* s *) :: [ 43 (* r *) :: l ] ] →
           retro_join [ 43 :: [ 48 :: left ] ] l
  | [ 48 (* s *) :: l ] →
     if retros left then match l with
       [ [] → failwith "Illegal␣arg␣to␣retro_join"
       | [ 32 (* t *) :: r ] →
             retro_join [ 27 (* ṭ *) :: [ 47 (* ṣ *) :: left ] ] r
       | [ 33 (* th *) :: r ] →
             retro_join [ 28 (* ṭh *) :: [ 47 (* ṣ *) :: left ] ] r
       | [ 36 (* n *) :: r ] →
             retro_join [ 31 (* ṇ *) :: [ 47 (* ṣ *) :: left ] ] r
       | l → retro_join [ 47 (* ṣ *) :: left ] l
       ]
```

```
    else retro_join [ 48 :: left ] l
  | [ c :: l ] → retro_join (homonase c left) l
  ]
;
(* sandhi of -s and -.h *)
value sglue first = fun
  [ [ 1 :: _ ] → [ −1; 12; first ] (* as -¿ o *)
  | _ → [ 48; first ] (* keep s *)
  ]
and sglue1 first _ = [ 48; first ] (* keep s *)
;
(* Restore main phoneme from finer distinction. *)
(* We unprime a primed phoneme by modulo 100 *)
(* Codes 124, 149 and 249 ought to disappear if phonemic features introduced *)
value restore = fun
  [ 124 → 24 (* restores j' → j *)
  | 149 | 249 → 49 (* restores h' → h and idem h" *)
  | c → c
  ]
;
(* Its extension to (reversed) words *)
value restore_stem = fun
  [ [ c :: r ] → [ restore c :: r ]
  | [] → []
  ]
;
(* Change of final consonant before consonant in internal sandhi *)
(* Gonda §19-II is not quite clear, so we keep a minimum rewrite set. *)
(* What is missing is the removal of all final consonants but one - eg vrazc *)
value cons_cons = fun
  [ 22 (* c *) | 23 (* ch *) | 24 (* j *) | 25 (* jh *) | 46 (* ś *)
                    → 17 (* k *) (* but sometimes ṭ like in finalize *)
  | 124 (* j' *) → 47 (* ṣ *)
  | 149 (* h' *) → 49 (* h *)
  | 26 (* ñ *) → 21 (* ṅ *)
  | 34 (* d *) → 32 (* t *)
  | 35 (* dh *) | 249 (* h" *) → 33 (* th *)
  | c → c
  ]
;
```

```
(* Error messages *)
value illegal_left w =
  let mess = "Left␣arg␣of␣sandhi␣end␣illegal␣in␣" ^ (rdecode w) in
  failwith mess
and illegal_right w =
  let mess = "Right␣arg␣of␣sandhi␣beginning␣illegal␣in␣" ^ (decode w) in
  failwith mess
and too_short () = failwith "Left␣arg␣of␣int_sandhi␣too␣short"
;
(* Internal sandhi - wl is mirror of code of left string, wr is code of right string. Result is
code after internal sandhi at their junction. This is a deterministic function. Optional rules
have to be encoded elsewhere. *)
value int_sandhi wl wr = try
  match wl with
    [ [] → (* eg "ap" *) wr
    | [ last :: before ] → match wr with
        [ [] → mirror (finalize wl)
        | [ first :: after ] →
          if vowel last then
              if vowel first then
                  let glue =
(* glue is the string replacing last; first with a special convention: when it starts with -1, it
means the last letter (an "a") of before is erased, and when it starts with -2, it means the
last letter (a vowel) of before is lengthened *)
          if savarna last first then [ long last ]
          else if avarna last then
                    if ivarna first then [ 10 ] (* e *)
                    else if uvarna first then [ 12 ] (* o *)
                    else match first with
                          [ 7 → [ 1; 43 ] (* ar *)
                          | 10 | 11 → [ 11 ] (* ai *)
                          | 12 | 13 → [ 13 ] (* au *)
                          | _ → failwith ".rr␣or␣.l␣initial"
                          ]
                else if ivarna last then [ 42; first ] (* y *)
                      (* but zrii+as=zriyas P{6,4,77} *)
                else if uvarna last then [ 45; first ] (* v *)
                      (* but bhuu+aadi=bhuuvaadi not bhvaadi irregular? *)
                else if last = 7 ∨ last = 8 (* .r .rr *) then [ 43; first ] (* r *)
                else (* last diphthong *)
```

```
                              match last with
                              [ 10 (* e *) → [ 1; 42; (* ay *) first ]
                              | 11 (* ai *) → [ 2; 42; (* āy *) first ]
                              | 12 (* o *) → [ 1; 45; (* av *) first ]
                              | 13 (* au *) → [ 2; 45; (* āv *) first ]
                              | _ → illegal_left wl
                              ]
                   (* let glue ... *) in
                      retro_join before (glue @ after)
               else (* first consonant last vowel *) match first with
                    [ 23 (* ch *) when short_vowel last →
                        (mirror wl) @ [ 22 :: wr ] (* cch *)
                    | 42 (* y *) →
                        let split = match last with (* P{6,1,79} *)
                                    [ 12 (* o *) → [ 45; 1 ] (* av *)
                                    | 13 (* au *) → [ 45; 2 ] (* aav *)
                                    | c → [ c ] (* e or ai included *)
                                    ] in
                        retro_join (split @ before) wr
                    | _ → retro_join wl wr
                    ]
               else (* consonant last *) (* should be analysed further *)
            if wr = [ 32 ] (* t *) then (* ad hoc *)
              let wl' = if visarg last (* s ḥ *) then
                             [ 32 :: before ] (* aśāt impft śās *) (* *azaa.h *)
                        else finalizer wl in
              mirror wl'
            else if all_consonants wr then mirror (finalizer wl)
            else if vowel first then retro_join [ restore last :: before ] wr
                                             (* j' → j & h' → h *)
                       (* no doubling of ṅ or n for internal sandhi no change of consonants
before vowels, even ch/cch *)
            else (* both consonant *) let glue = match first with
        [ 17 | 18 (* k kh *) →
                    match cons_cons last with
                  [ 41 → [ 36; first ] (* m+k → nk *) (* Gonda §19-VIII *)
                  | 48 → [ 16; first ] (* s+k → .hk could also be .sk 47; first *)
                  | 39 | 40 → [ 37; first ] (* b bh → p *)
                  | 33 → [ 32; first ] (* th → t *)
                  | c → [ c; first ]
```

```
               ]
    | 19 | 20 (* g gh *) →
                    if visarg last then sglue first before
                    else match cons_cons last with
               [ 41 → [ 36; first ] (* m+g → ng *) (* Gonda §19-VIII *)
               | c → [ voiced c; first ]
               ]
    | 22 | 23 (* c ch *) → match cons_cons last with
               [ 41 → [ 36; first ] (* m+c → nc *) (* Gonda §19-VIII *)
               | 32 | 34 → [ 22; first ] (* t+c → cc, d+c → cc *)
               | 33 → [ 32; first ] (* th → t *)
               | 36 → [ 14; 46; first ] (* n+c → ṃśc *)
               | 39 | 40 → [ 37; first ] (* b bh → p *)
               | c → [ if visarg c then 46 (* ś *) else c; first ]
               ]
    | 24 | 25 (* j jh *) →
                  if visarg last then sglue first before
                  else match cons_cons last with
               [ 41 → [ 36; first ] (* m+j → nj *) (* Gonda §19-VIII *)
               | 32 → [ 24; first ] (* t+j → jj *)
               | 36 → [ 26; first ] (* n+j → ñj *)
               | c → [ voiced c; first ] (* k+j → gj ? *)
               ]
    | 36 (* n *) → if visarg last then sglue1 first before (* ḥn → rn → rṇ *)
                      else match last with
               [ 41 → [ 36; 36 ] (* m+n → nn *) (* Gonda §19-VIII *)
               | 22 → [ 22; 26 ] (* c+n → cñ *) (* Gonda §19-IX *)
               | 24 | 124 → [ 24; 26 ] (* j+n → jñ *) (* Gonda §19-IX *)
               | 149 | 249 → [ 49; 36 ] (* h'+n → h+n same h" *)
               | c → [ c; 36 ] (* no other change - Gonda §19-I (except retroflexion e.g.
v.rk.na) *)
               ]
    | 37 | 38 (* p ph *) →
                match cons_cons last with
               [ 33 → [ 32; first ] (* th → t *)
               | 39 | 40 → [ 37; first ] (* b bh → p *)
               | c → [ if visarg c then 16 else c; first ]
               ]
    | 39 | 40 (* b bh *) → if visarg last then sglue first before
                              else match cons_cons last with
```

```
          [ c → [ voiced c; first ] ]
  | 41 (* m *) → if visarg last then sglue1 first before (* ḥm → rm *)
                 else match last with
          [ 41 → [ 36; 41 ] (* m+m → nm *) (* Gonda §19-VIII *)
          | _ → [ restore last; first ] (* no change Gonda §19-I *)
          ]
  | 42 (* y *) → if visarg last then sglue1 first before (* ḥy → ry *)
                 else [ restore last; first ]
  | 43 (* r *) → if visarg last then match before with
                     [ [ 3 :: [ 36 :: _ ] ] (* nis-r *) → [ − 1 ; 4 ; 43 ] (* nīr *)
                     | _ → [ restore last; first ]
                     ]
                 else match last with
          [ 41 → [ 36; 43 ] (* m+r → nr *) (* Gonda §19-VIII *)
          | _ → [ restore last; first ] (* no other change Gonda §19-I *)
          ]
  | 44 (* l *) → if visarg last then sglue1 first before (* ḥl → rl *)
                 else match last with
          [ 41 → [ 36; 44 ] (* m+l → nl *) (* Gonda §19-VIII *)
          | _ → [ restore last; first ] (* no other change Gonda §19-I *)
          ]
  | 45 (* v *) → if visarg last then sglue1 first before (* ḥv → rv *)
                 else match last with
          [ 41 → [ 36; 45 ] (* m+v → nv *) (* Gonda §19-VIII *)
          | _ → [ restore last; first ] (* no other change Gonda §19-I *)
          ]
  | 46 (* ś *) → match cons_cons last with
          [ 32 | 33 → [ 22; 23 ] (* t+ś → cch *)
          | 36 | 41 → [ 14; 46 ] (* n,m+ś → ṃś *) (* Gonda §19-VIII *)
          | 39 | 40 → [ 37; 46 ] (* b bh → p *)
          | 48 → [ 16; 46 ] (* s+ś → ḥś *)
          | c → [ c; first ]
          ]
  | 47 (* ṣ *) →
             match cons_cons last with
          [ 36 | 41 → [ 14; 47 ] (* n,m+ṣ → ṃṣ *) (* Gonda §19-VIII *)
          | 48 → [ 16; 47 ] (* s+ṣ → ḥṣ *)
          | 33 → [ 32; 47 ] (* th → t *)
          | 39 | 40 → [ 37; 47 ] (* b bh → p *)
          | 24 → [ 17; 47 ] (* j+ṣ → kṣ *)
```

```
            | c → [ c; first ]
            ]
    | 48 (* s *) →
              match cons_cons last with
            [ 36 | 41 → [ 14; 48 ] (* n,m+s → ṃs *) (* Gonda §19-VIII *)
            | 47 → match before with
              [ [ 17 :: _ ] → [ 47 ] (* kṣ+s → kṣ *)
              | _ → [ 17; 47 ] (* ṣ+s → kṣ *) (* Gonda §19-VI *)
              ]
            | 48 → match before with (* horrible glitch *)
              [ [] → [ 48 ] (* se 2 sg pm as#1 *)
              | [ 2 ] → [ 48; 48 ] (* āsse 2 sg pm ās#2 *)
              | _ → [ 16; 48 ] (* ḥs *)
              ]
            | 19 | 20 | 49 → [ 17; 47 ] (* g,h+s → kṣ : lekṣi dhokṣi *)
            | 249
            | 33 → [ 32; 48 ] (* th → t h"+s → ts natsyati*)
            | 39 | 40 → [ 37; 48 ] (* b bh → p *)
            | 17 → [ 17; 47 ] (* yuj yuñk+se → yuṅkṣe *)
            | c → [ c; first ]
            ]
    | 29 | 30 (* ḍ ḍh *) →
                  if visarg last then sglue first before
                  else match cons_cons last with
            [ 41 → [ 36; first ] (* m+ḍ → nḍ *) (* Gonda §19-VIII *)
            | 32 → [ 29; first ] (* t+ḍ → ḍḍ *)
            | 36 → [ 31; first ] (* n+ḍ → ṇḍ *)
            | c → [ voiced c; first ]
            ]
    | 34 (* d *) → if visarg last then sglue first before
                     else match cons_cons last with
            [ 41 → [ 36; first ] (* m+d → nd *) (* Gonda §19-VIII *)
            | 47 → [ 29; 29 ] (* ṣ+d → ḍḍ ? *)
            | c → [ voiced c; if lingual c then 29 (* ḍ *) else 34 ]
            ]
    | 35 (* dh *) → if visarg last then sglue first before
                      else match last with
            [ 32 | 33 | 35 → [ 34; 35 ] (* dh+dh → ddh *)(* Gonda §19-III *)
            | 41 → [ 36; 35 ] (* m+dh → ndh *) (* Gonda §19-VIII *)
            | 49 → [ -2; 30 ] (* h+dh → ḍh *) (* Gonda §19-VII *)
```

```
              | 22 | 23 | 149 → [ 19; 35 ] (* c+dh → gdh - dugdhve, vagdhi *)
              | 249 → [ 34; 35 ] (* h"+dh → ddh - naddhaa *)
              | 24 → [ 19; 35 ] (* j+dh → gdh *)(* yuṅgdhi *)
              | 47 → match before with
                [ [ 17 :: _ ] → [ −1; 29; 30 ] (* kṣ+dh → ḍḍh - caḍḍhve *)
                | _ → [ 29; 30 ] (* ṣ+dh → ḍḍh *)
                ]
              | 46 | 124 → [ 29; 30 ] (* ś+dh → ḍḍh id. j' *)
              | c → [ voiced c ; if lingual c then 30 (* ḍh *) else 35 ]
              ]
    | 32 (* t *) → match last with
              [ 41 → [ 36; 32 ] (* m+t → ṃt = nt *) (* Gonda §19-VIII *)
              | 20 | 149 → [ 19; 35 ] (* gh+t → gdh *) (* Gonda §19-III *)
              | 19 | 22 | 24 → [ 17; 32 ] (* g+t → kt *) (* P{8,4,54} *)
                      (* id c+t → kt *) (* Gonda §19-V ? *)
                      (* id j+t → kt *) (* yukta anakti bhunakti *)
              | 23 → match before with
                [ [ 22 :: _ ] → [ −1; 47; 27 ] (* cch+t → ṣṭ eg p.rṣṭa *)
                | _ → [ 47; 27 ] (* ch+t → ṣṭ *) (* ? *)
                ]
              | 25 → [ 24; 35 ] (* jh+t → jdh *) (* Gonda §19-III *)
              | 27 | 29 → [ 27; 27 ] (* ṭ+t → ṭṭ ḍ+t → ṭṭ *)
              | 28 → [ 27; 28 ] (* ṭh+t → ṭṭh *)
              | 30 → [ 29; 30 ] (* ḍh+t → ḍḍh *) (* Gonda §19-III ? *)
              | 33 → [ 32; 33 ] (* th+t → tth *)
              | 34 → [ 32; 32 ] (* d+t → tt *)
              | 35 | 249 → [ 34; 35 ] (* dh+t → ddh *) (* Gonda §19-III *)
              | 38 → [ 37; 33 ] (* ph+t → pth *)
              | 39 → [ 37; 32 ] (* b → p *)
              | 40 → [ 39; 35 ] (* bh+t → bdh *) (* Gonda §19-III *)
              | 46 (* ś+t → ṣṭ *)
              | 124 → [ 47; 27 ] (* j'+t → ṣṭ eg mṛj → mārṣṭi *)
              | 47 → match before with
                [ [ 17 :: _ ] → [ −1; 47; 27 ] (* kṣ+t → ṣṭ eg caṣṭe *)
                | _ → [ 47; 27 ] (* ṣ+t → ṣṭ *) (* Gonda §19-V *)
                ]
              | 49 → [ −2; 30 ] (* h+t → ḍh *) (* Gonda §19-VII *)
              | c → [ if visarg c then 48 (* s *) else c; first ]
              ]
    | 33 (* th *) → match last with
```

```
            [ 41 → [ 36; first ] (* m+th → ṃth = nth *)(* Gonda §19-VIII *)
            | 149 (* h'+t → gdh *)
            | 20 → [ 19; 35 ] (* gh+th → gdh *) (* Gonda §19-III *)
            | 22 | 23 → [ 17; 33 ] (* c+th → kth *) (* Gonda §19-V *)
            | 24 → [ 17; 33 ] (* j+th → kth *)
            | 25 → [ 24; 35 ] (* jh+th → jdh *) (* Gonda §19-III *)
            | 27 | 28 | 29 → [ 27; 28 ] (* ṭ(h)+th → ṭṭh ḍ+th → ṭṭh *)
            | 30 → [ 29; 30 ] (* ḍh+th → ḍḍh *) (* Gonda §19-III ? *)
            | 33 (* th+th → tth *)
            | 34 → [ 32; 33 ] (* d+th → tth *) (* ? *)
            | 35 | 249 → [ 34; 35 ] (* dh+th → ddh *) (* Gonda §19-III *)
            | 39 → [ 37; 33 ] (* b → p *)
            | 40 → [ 39; 35 ] (* bh+th → bdh *) (* Gonda §19-III *)
            | 124 (* j'+th → ṣṭh eg iyaṣṭha *)
            | 46 → [ 47; 28 ] (* ś+th → ṣṭh *)
            | 47 → match before with
              [ [ 17 :: _ ] → [ −1; 47; 28 ] (* kṣ+th → ṣṭh *)
              | _ → [ 47; 28 ] (* ṣ+th → ṣṭh *) (* Gonda §19-V *)
              ]
            | 49 → [ −2; 30 ] (* h+th → ḍh *) (* Gonda §19-VII *)
            | c → [ if visarg c then 48 else restore c; first ]
            ]
    | 27 | 28 (* ṭ ṭh *) → match cons_cons last with
            [ 41 → [ 36; first ] (* m+ṭ → nṭ *) (* Gonda §19-VIII *)
            | 32 | 33 → [ 27; first ] (* t+ṭ → ṭṭ d+ṭ → ṭṭ *)
            | 36 → [ 14; 47; first ] (* n+ṭ → ṃṣṭ *)
            | 39 | 40 → [ 37; first ] (* b bh → p *)
            | c → [ if visarg c then 47 else c; first ]
            ]
    | 49 (* h *) →
        if visarg last then sglue first before
        else match cons_cons last with
            [ 17 → [ 19; 20 ] (* k+h → ggh *)
            | 27 → [ 29; 30 ] (* ṭ+h → ḍḍh *)
            | 32 | 33 → [ 34; 35 ] (* t+h → ddh d+h → ddh *)
            | 37 → [ 39; 40 ] (* p+h → bbh *)
            | 41 → [ 36; 49 ] (* m+h → nh *) (* Gonda §19-VIII *)
            | c → [ c; 49 ]
            ]
    | _ → illegal_right wr
```

```
      ] (* let glue *) in
          let (w1, w2) = match glue with
            [ [] → failwith "empty␣glue"
            | [ −1 :: rest ] → match before with
                [ [] → too_short ()
                | [ _ (* a *) :: init ] → (init, rest @ after)
                ] (* as → o *)
            | [ −2 :: rest ] → match before with
                [ [] → too_short ()
                | [ 7 (* ṛ *) :: init ] → (before, rest @ after)
                | [ c :: init ] → (w, rest @ after)
                      where w = if vowel c then [ long c :: init ] (* guu.dha *)
                                 else before (* raraṃh+tha → raraṃḍha *)
                ] (* Gonda §19-VII *)
            | _ → (before, glue @ after)
            ] in retro_join w1 w2
          ] (* match wr *)
     ] (* match wl *)
with [ Failure s → failwith mess
        where mess = s ^ "␣in␣int_sandhi␣of␣" ^ (rdecode wl) ^ "&" ^ (decode wr) ]
;
value internal_sandhi left right =
  decode (int_sandhi (mirror (code left)) (code right))
;
```

tests

```
assert (internal_sandhi "ne" "ati" = "nayati");
assert (internal_sandhi "budh" "ta" = "buddha");
assert (internal_sandhi "rundh" "dhve" = "runddhve");
assert (internal_sandhi "d.rz" "ta" = "d.r.s.ta");
assert (internal_sandhi "dvi.s" "ta" = "dvi.s.ta");
assert (internal_sandhi "dvi.s" "dhvam" = "dvi.d.dhvam");
assert (internal_sandhi "han" "si" = "ha.msi");
assert (internal_sandhi "labh" "sye" = "lapsye"); (* I will take *)
assert (internal_sandhi "yaj" "na" = "yaj~na");
assert (internal_sandhi "han" "ka" = "hanka");
assert (internal_sandhi "gam" "va" = "ganva");
assert (internal_sandhi "lih" "ta" = "lii.dha");
assert (internal_sandhi "manas" "su" = "mana.hsu");
assert (internal_sandhi "jyotis" "stoma" = "jyoti.h.s.toma");
```

```
assert (internal_sandhi "manas" "bhis" = "manobhis");
assert (internal_sandhi "bhas" "ya" = "bhasya");
assert (internal_sandhi "bho" "ya" = "bhavya");
assert (internal_sandhi "sraj" "su" = "srak.su");
assert (internal_sandhi "yuj" "ta" = "yukta");
assert (internal_sandhi "yu~nj" "te" = "yufkte");
assert (internal_sandhi "tad" "" = "tat");
assert (internal_sandhi "vid" "aam" = "vidaam");
assert (internal_sandhi "nis" "rasa" = "niirasa");
assert (internal_sandhi "hi.ms" "aa" = "hi.msaa"); (* not hi.m.saa *)
assert (internal_sandhi "praa~nc" "s" = "praaf");
let adoh = duhify (Encode.rev_code_string "adoh") in
assert (decode (int_sandhi adoh (code "t")) = "adhok"); (* she milked - not "adho.t" *)
```

Not fully correct - still to be improved Special cases - to be accommodated at proper point in the derivation cf. Macdonell §60 footnote 1 p 26 d is assimilated before primary suffix -na: ad+na -¿ anna t and d are assimilated before secondary suffixes -mat and -maya: vidyunmat m.rnmaya

Interface for module Skt_morph

Sanskrit morphology interface

```
type deictic = [ Speaker | Listener | Self | Numeral ]
;
(* Deictics have their gender determined from the context for pronouns of 1st and 2nd person, or the reflexive pronoun "aatman", or numerals over 4 *)
type gender = [ Mas | Neu | Fem | Deictic of deictic ]
and genders = list gender
;
type number = [ Singular | Dual | Plural ]
;
type case = [ Nom (* nominatif *)
            | Acc (* accusatif *)
            | Ins (* instrumental *) (* comitatif (Henry) *)
            | Dat (* datif *)
            | Abl (* ablatif *)
            | Gen (* génitif *)
            | Loc (* locatif *)
            | Voc (* vocatif *)
            ]
```

```
;
(* The verb system *)
type gana = int (* present class: 1 to 10, plus 11 for denominatives *)
and aor_class = int (* aorist class: 1 to 7 *)
;
type person = [ First | Second | Third ] (* Indian Third, Second and First *)
;
type conjugation = [ Primary | Causative | Desiderative | Intensive ]
;
type finite = (conjugation × paradigm) (* finite forms of verbs *)
and paradigm =
  [ Presenta of gana and pr_mode (* parasmaipade *)
  | Presentm of gana and pr_mode (* aatmanepade *)
  | Presentp of pr_mode (* passive of present system *)
  | Conjug of tense and voice (* other tenses/modes/aspects *)
  | Perfut of voice (* periphrastic futur (lu.t) - always active *)
  ]
and voice = [ Active | Middle | Passive ] (* diathesis (pada: Para Atma Ubha) *)
and pr_mode =
  [ Present (* Indicative (la.t) *)
  | Imperfect (* Preterit (laf) *)
  | Imperative (* (lo.t) *)
  | Optative (* Potential (lif) *)
  ]
and tense =
  [ Future (* (l.r.t) *)
  | Perfect (* Remote past - resultative aspect (li.t) *)
  | Aorist of aor_class (* Immediate past or future with perfective aspect (luf) *)
  | Injunctive of aor_class (* (le.t) - injunctions also Prohibitive with maa *)
  | Benedictive (* Precative: optative aorist (aazirlif) *)
  | Conditional (* Preterit of future (l.rf) *)
  | Subjunctive (* le.t *) (* Rare subjunctive, intermediate between Optative and Imperative
*)
  ]
;
(* NB from Indo-European: the present stem has the imperfective aspect, the aorist one the
perfective aspect, and the perfect one the resultative. *)
(* Vedic Subjunctive and Pluperfect are not yet taken into account. The only non-present
passive forms are some passive aorist forms in 3rd sg. Future, Perfect and Aorist use Midddle
forms for Passive. *)
```

Verbal adjectives

```
type kritya = int (* shades of intention of passive future/potential participle: 1 -ya (obligation, necessity or possibility, potentiality) (yat kyap .nyat) 2 -aniiya (fitness, desirability, effectivity) (aniiyar) 3 -tavya (necessity, unavoidability) (tavyat) *)
;
type verbal = (conjugation × participle)
and participle = (* participles *)
(* These are the kridanta stems (primary verbal derivatives) with participial value. They act as adjectives or gendered nouns. But Ppra does not qualify as a noun, but as an adverb, signifying simultaneous action. *)
  [ Ppp (* passive past participle *)
  | Pppa (* active past participle *)
  | Ppra of gana (* active present participle *)
  | Pprm of gana (* middle present participle *)
  | Pprp (* passive present participle *)
  | Ppfta (* active perfect participle *)
  | Ppftm (* middle perfect participle *) (* no passive *)
  | Pfuta (* active future participle *)
  | Pfutm (* middle future participle *)
  | Pfutp of kritya (* passive future/potential participle/gerundive 3 forms *)
  | Action_noun (* generative only for auxiliaries, for cvi compounds *)
(*— Agent_noun, etc. – non generative, must be lexicalized; see nominal *)
  ]
;
(* Invariable verbal forms. Such forms are indeclinable and have their own inflected forms constructors. Infinitives are similar to dative substantival forms, periphrastic perfect forms are associated with an auxiliary verb in the perfect. Absolutives split into root absolutives in -tvaa and absolutives in -ya that must be prefixed with a preverb. Absolutives in -aam (.namul) are in both. *)
type modal = (conjugation × invar)
and invar =
  [ Infi (* infinitive (tumun) *)
  | Absoya (* absolutive (gerund, invariable participle) (lyap) *)
  | Perpft (* periphrastic perfect (li.t) *)
  ]
;
(* Varieties of na n-samaasas *)
type nan_kind =
  [ Neg (* logical negation: adj -¿ adj *)
  | Not (* sentential negation: adv -¿ adv *)
```

```
  | Opp (* opposite notion: subst -¿ subst preserving gender *)
  | Priv (* bahuvrihi: noun -¿ adj with gender-raising *)
  | Abse (* noun -¿ noun in n. *)
  ]
;
type sadhana = (* karaka, action or absolutive - coarser than krit *)
  [ Agent
  | Action
  | Object
  | Instr
  | Orig (* unused *)
  | Loca
  | Absolu
  | Nan of nan_kind
  ]
;
(* Primary nominal formations (k.rdantas) *)
type nominal = (conjugation × krit)
and krit = (* coarser than Paninian krit suffixes *)
  [ Agent_aka (* .nvul P{3,1,133} P{3,3,108-109} -aka -ikaa v.rddhi .svun P{3,1,145} trade gu.na f. -akii vu n P{3,1,146-147} vun P{3,1,149-150} repeated action, benediction *)
  | Agent_in (* .nini P{3,1,134} P{3,2,78-86} -in -inii v.rddhi ghinu.n P{3,2,141-145} ini P{3,2,93} ifc. -vikrayin past *)
  | Agent_tri (* t.rc P{3,1,133} t.rn P{3,2,135} habit -t.r gu.na *)
  | Agent_ana (* lyu P{3,1,134} yuc P{3,2,148} -ana a. .nyu.t P{3,1,147-148} profession f. -anii *)
  | Agent_root (* kvip P{3,2,61} ifc + P{3,2,76} adja ifc. mnf. P{6,1,67} amuis de v P{3,2,76} root autonomous mnf. + .tak P{3,2,8} root ifc (f. -ii) + .ta P{3,2,20} -kara ifc (f. -ii) habitual, enjoy + ka P{3,2,3} root -aa, amuie, ifcno (no preverb) f. ii *)
  | Agent_a (* ac P{3,1,134} gu.na m. -a f. -aa .na P{3,1,140-143} v.rddhi (f. -aa) ka P{3,1,135-136;144} -gu.na P{3,2,3-7} m. -a (f. -aa) metaphoric use za P{3,1,137-139} idem ka but (f. -aa) nb present stem a.n P{3,2,1} vriddhi ifc (iic obj) (f. -ii) -kaara *)
  | Agent_nu (* i.s.nu P{3,2,136} i.s.nuc P{3,2,136-138} -i.s.nu gu.na (habit) khi.s.nuc P{3,2,57} -i.s.n'u gu.na knu P{3,2,140} ksnu P{3,2,139} -nu -gu.na *)
  | Action_ana (* lyu.t P{3,3,115-117} -ana n. *)
  | Action_na (* naf P{3,3,90} nan P{3,3,91} -na m. -naa f. *)
  | Action_a (* gha n P{3,3,18-} -a m. v.rddhi *)
  | Action_ya (* kyap P{3,1,107} -ya n. -yaa f. *)
  | Action_ti (* ktin P{3,3,94} -ti f. *)
  | Action_i (* ki P{3,3,92-93} -i f. *)
```

| *Action_root* (∗ unknown krit of non-agent noun ∗)
| *Object_root* (∗ we should probably lump action and object in *Non_agent* ∗)
| *Object_a* (∗ ka -a n. ∗)
| *Instrument* (∗ ka **P**{3,1,136} 0/amui n. ∗)
| *Instra* (∗ .s.tran -tra n. -trii f. traa f. ∗)
| *Orig_root* (∗ sruc srut sruva ∗)
| *Agent_u* (∗ san+u -u on des stem ∗)
| *Action_aa* (∗ san+a+.taap **P**{3,3,102} -aa on des stem ∗)
| *Abstract* (∗ abstract nouns n. -as u.naadi suffix ∗)
]
;

type *ind_kind* =
[*Adv* (∗ adverb ∗)
| *Avya* (∗ turned into an adverb by avyayiibhaava compounding ∗)
| *Abs* (∗ root absolutive in -tvaa ∗)
| *Tas* (∗ tasil taddhita ∗)
| *Part* (∗ particule ∗)
| *Prep* (∗ preposition ∗)
| *Conj* (∗ conjunction ∗)
| *Nota* (∗ notation ∗)
| *Infl* (∗ inflected form ∗)
| *Interj* (∗ interjection ∗)
| *Default* (∗ default - inherits its role ∗)
]
;

Interface for module Morphology

Morphology interface
Used by *Inflected* for inflective morphology generation, and by *Morpho* for further treatment

open *Skt_morph*;

module Morphology : sig

type *inflexion_tag* = (∗ vibhakti ∗)
[*Noun_form* of *gender* and *number* and *case* (∗ declined nominal ∗)
| *Part_form* of *verbal* and *gender* and *number* and *case* (∗ declined participle ∗)
| *Bare_stem* (∗ iic forms ∗)
| *Avyayai_form* (∗ iic forms of avyayiibhaava cpds ∗)
| *Avyayaf_form* (∗ ifc forms of avyayiibhaava cpds ∗)

```
  | Verb_form of finite and number and person (* finite conjugated root forms *)
  | Ind_form of ind_kind (* indeclinable forms: prep, adv, etc *)
  | Ind_verb of modal (* indeclinable inf abs-ya and perpft *)
  | Abs_root of conjugation (* abs-tvaa *)
  | Gati (* iiv verbal auxiliaries forms *)
  | Unanalysed (* un-analysable segments *)
  | PV of list string (* Preverb sequences *)
  (* NB preverb sequences are collated separately by Roots module, and they do not appear
in solutions, they are removed by compression of Dispatcher.validate. *)
  ]
and inflexions = list inflexion_tag
;
type inflected_map = Lexmap.lexmap inflexions
and lemma = Lexmap.inverse inflexions
and lemmas = list lemma
;
type unitag = (Word.delta × inflexions)
and multitag = list unitag
;
type morphology =
  { nouns : inflected_map
  ; nouns2 : inflected_map
  ; prons : inflected_map
  ; roots : inflected_map
  ; krids : inflected_map
  ; voks : inflected_map
  ; lopas : inflected_map
  ; lopaks : inflected_map
  ; indes : inflected_map
  ; absya : inflected_map
  ; abstvaa : inflected_map
  ; iics2 : inflected_map
  ; iics : inflected_map
  ; iifs : inflected_map
  ; iiks : inflected_map
  ; iivs : inflected_map
  ; peris : inflected_map
  ; auxis : inflected_map
  ; auxiinvs : inflected_map
  ; auxiks : inflected_map
```

```
  ; auxiicks : inflected_map
  ; vocas : inflected_map
  ; invs : inflected_map
  ; ifcs : inflected_map
  ; ifcs2 : inflected_map
  ; inftu : inflected_map
  ; kama : inflected_map
  ; iiys : inflected_map
  ; avys : inflected_map
  ; caches : inflected_map
  ; cacheis : inflected_map
  }
;
end;
```

Module Naming

Unique naming mechanism.
Kridanta names management: namespace data structures
The problem is to find the lexical entry, if any, that matches a stem and an etymology, corresponding to the morphological structure of a generated stem. For instance entry "k.rta" has etymology pp(k.r#1). It does not produce forms, and is skipped by the morphology generator, since the pp participal stem is a productive taddhita construction, that will indeed generate stem k.rta from its root k.r#1. The problem for the morphology generator is to display forms of k.rta with a link to k.rta in the hypertext lexicon. It is non-trivial, since homonymies occur. Thus homophony indexes associated with generators and consistent with possible lexicalizations must be registered. A first pass of recording builds *lexical_kridantas* as a *deco_krid* deco indexing the stems with a pair (morphology,homo). Then the morphology generator from Inflected extends it as *unique_kridantas*, accessed as *Inflected.acccess_krid* and *Inflected.register_krid*, and used by *Parts.gen_stem*.
Unique naming of kridantas
associates to a pair (verbal,root) a homophony index for unique naming

```
type homo_krid = ((Skt_morph.verbal × Word.word) × int)
and deco_krid = Deco.deco homo_krid
;
value homo_undo w = Encode.decompose (Word.mirror w)
;
value look_up_homo homo = look_rec
  where rec look_rec = fun
```

```
  [ [] → failwith "look_up_homo"
  | [ (morpho, n) :: rest ] → if n = homo then morpho else look_rec rest
  ]
;
value unique_kridantas =
  try (Gen.gobble Data.public_unique_kridantas_file : deco_krid)
  with [ _ → failwith "unique_kridantas" ]
and lexical_kridantas =
  try (Gen.gobble Data.public_lexical_kridantas_file : deco_krid)
  with [ _ → failwith "lexical_kridantas" ]
;
```

(* This mechanism is used by *Make_roots* at morphology generation time, and by *Morpho.print_inv_mo* and *Morpho_ext.print_inv_morpho_ext* at segmenting time. *)

Here we retrieve finer discrimination for verbs forms preceded by preverbs. This is experimental, and incurs too many conversions between strings and words, suggesting a restructuring of preverbs representation.

```
value preverbs_structure = (* Used in Morpho for display of pvs *)
  try (Gen.gobble Data.public_preverbs_file : Deco.deco Word.word)
  with [ _ → failwith "preverbs_structure" ]
;
```

Interface for module Inflected

```
open Skt_morph;
open Morphology;
open Naming;

value register_krid : Word.word → homo_krid → unit;
value access_krid : Word.word → list homo_krid;

value admits_aa : ref bool;
value morpho_gen : ref bool
;
value nouns : ref inflected_map;
value pronouns : ref inflected_map;
value vocas : ref inflected_map;
value iics : ref inflected_map;
value avyayais : ref inflected_map;
value avyayafs : ref inflected_map;
value piics : ref inflected_map;
```

```
value iivs : ref inflected_map;
value peri : ref inflected_map;
value auxi : ref inflected_map;
value auxiinv : ref inflected_map;
value auxik : ref inflected_map;
value auxiick : ref inflected_map;
value indecls : ref inflected_map;
value invs : ref inflected_map;
value absya : ref inflected_map;
value abstvaa : ref inflected_map;
value parts : ref inflected_map;
value partvocs : ref inflected_map;
value roots : ref inflected_map;
value lopas : ref inflected_map;
value lopaks : ref inflected_map;
value inftu : ref inflected_map;
value kama : ref inflected_map;
value preverbs : ref (Deco.deco Word.word);

value lexicalized_kridantas : ref deco_krid;
value unique_kridantas : ref deco_krid;
```

Inflectional categories

```
type nominal =
  [ Noun (* lexicalized stem - noun, adjective or number *)
  | Pron (* lexicalized stem - pronoun *)
  | Krid of verbal and string (* kridantas of roots *)
  ]
;
type flexion =
  [ Declined of nominal and gender and list (number × list (case × Word.word))
  | Conju of finite and list (number × list (person × Word.word))
  | Indecl of ind_kind and Word.word
  | Bare of nominal and Word.word
  | Avyayai of Word.word (* Iic of avyayiibhaava cpd *)
  | Avyayaf of Word.word (* Ifc of avyayiibhaava cpd *)
  | Cvi of Word.word
  | Preverb of Word.word and list Word.word
  | Invar of modal and Word.word (* inf abs-ya perpft *)
  | Inftu of conjugation and Word.word (* infinitive in -tu *)
  | Absotvaa of conjugation and Word.word (* abs-tvaa *)
```

```
  ]
;
value enter1 : string → flexion → unit
;
value enter_ind_ifc : string → flexion → unit
;
value enter : string → list flexion → unit
;
value enter_form : Word.word → flexion → unit
;
value enter_forms : Word.word → list flexion → unit
;
value nominal_databases : unit →
  (inflected_map × inflected_map × inflected_map × inflected_map × inflected_map)
;
value reset_nominal_databases : unit → unit
;
```

Module Inflected

Morphology : computation of inflected forms in *inflected_map decls*.

```
open Skt_morph;
open Morphology; (* inflected_map *)
open Word;
```

Holds the state vector : (*nouns*, *roots*, *preverbs*, *segmenting_mode*) where:
nouns is accumulator for the set of declined forms of substantives
pronouns is accumulator for the set of declined forms of pronouns
vocas is accumulator for the set of vocative forms of substantives
roots is accumulator for the set of conjugated forms of roots
preverbs is accumulator for the set of preverb sequences
segmenting_mode tells whether phantom phonemes are generated or not.
This code serves 2 purposes: at building morphology time, it generates all the forms to populate the morphology banks. It is also reused at execution time, by Declension and Conjugation cgis.
Admits aa- as a preverb – global set in *Verbs.compute_conjugs_stems*

```
value admits_aa = ref False
and admits_lopa = ref False
;
```

```
value morpho_gen = ref True (* morphology generation time *)
(* Turn to False for cgi execution (fake conjugation and no phantoms) *)
;
(* The inflected_map lexicons of inflected forms: nouns, iics, etc are computed by Make_nouns
and are dumped as persistent global databases nouns.rem etc. They are also used on the fly
locally by Declension and Conjugation. *)
value lexicalized_kridantas = ref (Deco.empty : Naming.deco_krid)
(* It will be set by Make_roots.roots_to_conjugs in the first pass of grinding, and used for
the unique_kridantas computation in the second. *)
;
value access_lexical_krid stem = Deco.assoc stem lexicalized_kridantas.val
;
(* We look up the lexicalized kridantas register to see if entry is a krid. *)
value is_kridanta entry = try
  let (hom, stem) = Encode.decompose_str entry in
  let krids = access_lexical_krid stem in
  let _ = List.find (fun (_, h) → h = hom) krids in True
  with [ Not_found → False ]
;
value unique_kridantas = ref Deco.empty
(* This structure holds the unique names to kridantas. It is initialized to the lexical-
ized one in Make_roots.roots_to_conjugs, which completes it with the kridantas generated
by Parts. At the end of morphological generation its final value is stored in persistent
Install.unique_kridantas_file, and transfered to Install.public_unique_kridantas_file read
from module Naming. This allows the tag of kridantas to link to the lexicon entry in case
it has been lexicalized. This does not work for kridantas with preverbs, and at the moment
concerns only participles, not nouns of agent or action. *)
;
value access_krid stem = Deco.assoc stem unique_kridantas.val
and register_krid stem vrp = (* used in Parts.gen_stem *)
  unique_kridantas.val := Deco.add1 unique_kridantas.val stem vrp
;
(* Inflected forms of nouns pronouns numbers, *)
(* also used separately for ifc only nouns *)
value nouns = ref (Deco.empty : inflected_map)
and pronouns = ref (Deco.empty : inflected_map) (* demonstrative + personal pn *)
and vocas = ref (Deco.empty : inflected_map)
;
(* Add morphological feature i to form w relative to entry e, with d = diff e *)
value add_morph w d i =
```

```
  nouns.val := Lexmap.addl nouns.val w (d w, i)
and add_morphpro w d i = (* pronouns not usable as ifc *)
  pronouns.val := Lexmap.addl pronouns.val w (d w, i)
(* Add vocative feature i to form w relative to entry e, with d = diff e *)
and add_voca w d i =
  vocas.val := Lexmap.addl vocas.val w (d w, i)
;
(* auxiliary verbs used in the inchoative cvi construction *)
value auxiliary = fun
  [ "bhuu#1" | "k.r#1" | "as#1" → True | _ → False ]
;
(* iic forms *)
value iics = ref (Deco.empty : inflected_map)
;
value add_morphi w d i =
  iics.val := Lexmap.addl iics.val w (d w, i)
;
(* avyaya iic forms *)
value avyayais = ref (Deco.empty : inflected_map)
;
(* avyaya ifc forms *)
value avyayafs = ref (Deco.empty : inflected_map)
;
value add_morphyai w d i =
  avyayais.val := Lexmap.addl avyayais.val w (d w, i)
;
value add_morphyaf w d i =
  avyayafs.val := Lexmap.addl avyayafs.val w (d w, i)
;
(* Used by Nouns.fake_compute_decls for declension of single entry *)
value nominal_databases () =
  (nouns.val, pronouns.val, vocas.val, iics.val, avyayafs.val)
and reset_nominal_databases () = do
  { nouns.val := Deco.empty
  ; pronouns.val := Deco.empty
  ; vocas.val := Deco.empty
  ; iics.val := Deco.empty
  ; avyayafs.val := Deco.empty
  }
;
```

iiv forms

```
value iivs = ref (Deco.empty : inflected_map)
;
value add_morphvi w d i =
  iivs.val := Lexmap.addl iivs.val w (d w, i)
;
(* forms of auxiliary roots "k.r" "bhuu" and "as" *)
value auxi = ref (Deco.empty : inflected_map) (* red tinantas *)
and auxiinv = ref (Deco.empty : inflected_map) (* mauve abs and inf *)
;
value add_morphauxi w d i =
  if Phonetics.phantomatic w then () else
  auxi.val := Lexmap.addl auxi.val w (d w, i)
and add_morphauxiinv w d i =
  if Phonetics.phantomatic w then () else
  auxiinv.val := Lexmap.addl auxiinv.val w (d w, i)
;
(* periphrastic perfect forms *)
value peri = ref (Deco.empty : inflected_map)
;
value add_morphperi w d i =
  peri.val := Lexmap.addl peri.val w (d w, i)
;
(* indeclinable forms - adverbs, conjonctions, particles *)
value indecls = ref (Deco.empty : inflected_map)
;
value add_morphind w d i =
  indecls.val := Lexmap.addl indecls.val w (d w, i)
;
(* invocations are registered in invs *)
value invs = ref (Deco.empty : inflected_map)
;
value add_invoc w d i =
  invs.val := Lexmap.addl invs.val w (d w, i)
;
(* indeclinable verbal forms usable without preverbs: infinitives, abs-tvaa *)
value abstvaa = ref (Deco.empty : inflected_map)
;
value add_morphabstvaa w d i =
  abstvaa.val := Lexmap.addl abstvaa.val w (d w, i)
```

```
;
(* indeclinable verbal forms usable with preverbs: infinitives, abs-ya *)
value absya = ref (Deco.empty : inflected_map)
;
value add_morphabsya w d i aapv = do
  { absya.val := Lexmap.addl absya.val w (d w, i)
  (* now we add fake absol forms with phantom phonemes *)
  ; if morpho_gen.val ∧ aapv then match w with
        [ [ 1 :: r ] → (* aa-a gives *a *)
            let fake = [ (* *a *) -3 :: r ] in
            absya.val := Lexmap.addl absya.val fake (d fake, i)
        | [ 2 :: r ] →
            let fake = [ (* *A *) -9 :: r ] in
            absya.val := Lexmap.addl absya.val fake (d fake, i)
        | [ 3 :: r ] →
            let fake = [ (* *i *) -4 :: r ] in
            absya.val := Lexmap.addl absya.val fake (d fake, i)
        | [ 4 :: r ] →
            let fake = [ (* *I *) -7 :: r ] in
            absya.val := Lexmap.addl absya.val fake (d fake, i)
        | [ 5 :: r ] →
            let fake = [ (* *u *) -5 :: r ] in
            absya.val := Lexmap.addl absya.val fake (d fake, i)
        | [ 6 :: r ] →
            let fake = [ (* *U *) -8 :: r ] in
            absya.val := Lexmap.addl absya.val fake (d fake, i)
        | [ 7 :: r ] →
            let fake = [ (* *r *) -6 :: r ] in
            absya.val := Lexmap.addl absya.val fake (d fake, i)
        | [ 23 :: r ] → (* aa-ch gives *C *)
            let fake = [ (* *C *) 123 :: r ] in
            absya.val := Lexmap.addl absya.val fake (d fake, i)
        | _ → ()
        ]
    else ()
  }
;
(* root finite conjugated forms *)
value roots = ref (Deco.empty : inflected_map)
;
```

```
value add_morphc w d i aapv = do
  { roots.val := Lexmap.addl roots.val w (d w, i)
  (* now we add fake conjugated forms with phantom phonemes *)
  ; if morpho_gen.val ∧ aapv then do (* P{6,1,95} *)
      { match w with
        [ [ 1 :: r ] → (* aa-a gives *a *)
              let fake = [ (* *a *) -3 :: r ] in
              roots.val := Lexmap.addl roots.val fake (d fake, i)
        | [ 2 :: r ] →
              let fake = [ (* *A *) -9 :: r ] in
              roots.val := Lexmap.addl roots.val fake (d fake, i)
        | [ 3 :: r ] →
              let fake = [ (* *i *) -4 :: r ] in
              roots.val := Lexmap.addl roots.val fake (d fake, i)
        | [ 4 :: r ] →
              let fake = [ (* *I *) -7 :: r ] in
              roots.val := Lexmap.addl roots.val fake (d fake, i)
        | [ 5 :: r ] →
              let fake = [ (* *u *) -5 :: r ] in
              roots.val := Lexmap.addl roots.val fake (d fake, i)
        | [ 6 :: r ] →
              let fake = [ (* *U *) -8 :: r ] in
              roots.val := Lexmap.addl roots.val fake (d fake, i)
        | [ 7 :: r ] →
              let fake = [ (* *r *) -6 :: r ] in
              roots.val := Lexmap.addl roots.val fake (d fake, i)
        | [ 23 :: r ] → (* aa-ch gives *C *)
              let fake = [ (* *C *) 123 :: r ] in
              roots.val := Lexmap.addl roots.val fake (d fake, i)
        | _ → ()
        ]
      }
    else ()
  }
;
(* root finite forms starting with e or o *)
value lopas = ref (Deco.empty : inflected_map)
and lopaks = ref (Deco.empty : inflected_map)
;
(* Concerns P{6,1,94} a,ā (preverb) — e (root) -¿ e; same for o. *)
```

```
(* Ex: upelayati prelayati upo.sati pro.sati *)
value add_morphlopa w d i = match w with
  [ [ 10 :: _ ]
  | [ 12 :: _ ] → let amui = [ −2 :: w ] (* amuitic form *) in
                    lopas.val := Lexmap.addl lopas.val amui (d amui, i)
  | _ → ()
  ]
;
(* New style of forms generators - stem argument generated as pseudo-entry *)
```

inflected forms of participles - and more generally kridantas

```
value parts = ref (Deco.empty : inflected_map)
;
value add_morphpa w stem i aapv = do
  { parts.val := Lexmap.addl parts.val w (diff w stem, i)
  (* now we add fake participial forms with phantom phonemes *)
  ; if morpho_gen.val ∧ aapv then match w with
      [ [ 1 :: r ] → (* aa-a gives *a *)
          let fake = [ (* *a *) -3 :: r ] in
          parts.val := Lexmap.addl parts.val fake (diff fake stem, i)
      | [ 2 :: r ] →
          let fake = [ (* *A *) -9 :: r ] in
          parts.val := Lexmap.addl parts.val fake (diff fake stem, i)
      | [ 3 :: r ] →
          let fake = [ (* *i *) -4 :: r ] in
          parts.val := Lexmap.addl parts.val fake (diff fake stem, i)
      | [ 4 :: r ] →
          let fake = [ (* *I *) -7 :: r ] in
          parts.val := Lexmap.addl parts.val fake (diff fake stem, i)
      | [ 5 :: r ] →
          let fake = [ (* *u *) -5 :: r ] in
          parts.val := Lexmap.addl parts.val fake (diff fake stem, i)
      | [ 6 :: r ] →
          let fake = [ (* *U *) -8 :: r ] in
          parts.val := Lexmap.addl parts.val fake (diff fake stem, i)
      | [ 7 :: r ] → (* aa-.r gives *r *)
          let fake = [ (* *r *) -6 :: r ] in
          parts.val := Lexmap.addl parts.val fake (diff fake stem, i)
      | [ 23 :: r ] → (* aa-ch gives *C *)
          let fake = [ (* *C *) 123 :: r ] in
```

```
        parts.val := Lexmap.addl parts.val fake (diff fake stem, i)
      | _ → ()
      ]
    else ()
  }
and add_morphlopak w stem i aapv = match w with
  [ [ 10 :: _ ]
  | [ 12 :: _ ] → let amui = [ −2 :: w ] (* amuitic form *) in
        lopaks.val := Lexmap.addl lopaks.val amui (diff amui stem, i)
  | _ → ()
  ]
;
(* participial vocatives *)
value partvocs = ref (Deco.empty : inflected_map)
;
value add_morphpav w stem i aapv = do
  { partvocs.val := Lexmap.addl partvocs.val w (diff w stem, i)
  (* now we add fake participial forms with phantom phonemes *)
  ; if morpho_gen.val ∧ aapv then match w with
      [ [ 1 :: r ] → (* aa-a gives *a *)
          let fake = [ (* *a *) -3 :: r ] in
          partvocs.val := Lexmap.addl partvocs.val fake (diff fake stem, i)
      | [ 2 :: r ] →
          let fake = [ (* *A *) -9 :: r ] in
          partvocs.val := Lexmap.addl partvocs.val fake (diff fake stem, i)
      | [ 3 :: r ] →
          let fake = [ (* *i *) -4 :: r ] in
          partvocs.val := Lexmap.addl partvocs.val fake (diff fake stem, i)
      | [ 4 :: r ] →
          let fake = [ (* *I *) -7 :: r ] in
          partvocs.val := Lexmap.addl partvocs.val fake (diff fake stem, i)
      | [ 5 :: r ] →
          let fake = [ (* *u *) -5 :: r ] in
          partvocs.val := Lexmap.addl partvocs.val fake (diff fake stem, i)
      | [ 6 :: r ] →
          let fake = [ (* *U *) -8 :: r ] in
          partvocs.val := Lexmap.addl partvocs.val fake (diff fake stem, i)
      | [ 7 :: r ] → (* aa-.r gives *r *)
          let fake = [ (* *r *) -6 :: r ] in
          partvocs.val := Lexmap.addl partvocs.val fake (diff fake stem, i)
```

```
        | [ 23 :: r ] → (* aa-ch gives *C *)
            let fake = [ (* *C *) 123 :: r ] in
            partvocs.val := Lexmap.addl partvocs.val fake (diff fake stem, i)
        | _ → ()
        ]
    else ()
  }
;
(* piic forms *)
value piics = ref (Deco.empty : inflected_map)
;
value add_morphpi w stem i aapv = do
  { piics.val := Lexmap.addl piics.val w (diff w stem, i)
  (* now we add fake participial iic forms with phantom phonemes *)
  ; if morpho_gen.val ∧ aapv then match w with
        [ [ 1 :: r ] → (* aa-a gives *a *)
            let fake = [ (* *a *) -3 :: r ] in
            piics.val := Lexmap.addl piics.val fake (diff fake stem, i)
        | [ 2 :: r ] →
            let fake = [ (* *A *) -9 :: r ] in
            piics.val := Lexmap.addl piics.val fake (diff fake stem, i)
        | [ 3 :: r ] →
            let fake = [ (* *i *) -4 :: r ] in
            piics.val := Lexmap.addl piics.val fake (diff fake stem, i)
        | [ 4 :: r ] →
            let fake = [ (* *I *) -7 :: r ] in
            piics.val := Lexmap.addl piics.val fake (diff fake stem, i)
        | [ 5 :: r ] →
            let fake = [ (* *u *) -5 :: r ] in
            piics.val := Lexmap.addl piics.val fake (diff fake stem, i)
        | [ 6 :: r ] →
            let fake = [ (* *U *) -8 :: r ] in
            piics.val := Lexmap.addl piics.val fake (diff fake stem, i)
        | [ 7 :: r ] → (* aa-.r gives *r *)
            let fake = [ (* *r *) -6 :: r ] in
            piics.val := Lexmap.addl piics.val fake (diff fake stem, i)
        | [ 23 :: r ] → (* aa-ch gives *C *)
            let fake = [ (* *C *) 123 :: r ] in
            piics.val := Lexmap.addl piics.val fake (diff fake stem, i)
        | _ → ()
```

```
        ]
    else ()
  }
;
(* kridantas of auxiliary roots k.r bhuu for cvi -ii compounds *)
value auxik = ref (Deco.empty : inflected_map)
;
value add_morphauxik w stem i =
  if Phonetics.phantomatic w then () else
  auxik.val := Lexmap.addl auxik.val w (diff w stem, i)
;
value auxiick = ref (Deco.empty : inflected_map)
;
value add_morphauxiick w stem i =
  if Phonetics.phantomatic w then () else
  auxiick.val := Lexmap.addl auxiick.val w (diff w stem, i)
;
(* Root infinitives in -tu with forms of kaama *)
value inftu = ref (Deco.empty : inflected_map)
and kama = ref (Deco.empty : inflected_map)
;
value add_morphinftu w d i = (* similar to add_morphind *)
                          if Phonetics.phantomatic w then () else
  inftu.val := Lexmap.addl inftu.val w (d w, i)
and add_morphkama w d i = (* similar to add_morph *)
  kama.val := Lexmap.addl kama.val w (d w, i)
;
```

Preverb sequences

```
value preverbs = ref (Deco.empty : Deco.deco word)
;
value add_morphp w i = preverbs.val := Deco.add preverbs.val w i
;
(* Inflectional categories *)
type nominal =
  [ Noun (* lexicalized stem - noun, adjective or number *)
  | Pron (* lexicalized stem - pronoun *)
  | Krid of verbal and string (* kridantas of roots *)
  ]
;
```

```
type sup = (number × list (case × word (* stem *))) (* nominal generators *)
 and tin = (number × list (person × word (* root *))) (* verbal generators *)
;
type flexion =
  [ Declined of nominal and gender and list sup
  | Conju of finite and list tin
  | Indecl of ind_kind and word (* avyaya, particle, interjection, nota *)
  | Bare of nominal and word (* Iic *)
  | Avyayai of word (* Iic of avyayiibhaava cpd *)
  | Avyayaf of word (* Ifc of avyayiibhaava cpd *)
  | Cvi of word (* -cvi suffixed stem (iiv) for inchoative compound verbs *)
  | Preverb of word and list word
  | Invar of modal and word (* infinitive abs-ya perpft *)
  | Inftu of conjugation and Word.word (* infinitive in -tu *)
  | Absotvaa of conjugation and word (* abs-tvaa *)
  ]
;
```

Now functions that populate the inflected forms treebanks from the lexemes enter1: string -¿ flexion -¿ unit

```
value enter1 entry =
  let delta = Encode.diff_str entry (* partial application for patching *)
  and aapv = admits_aa.val (* for phantom forms generation *) in fun
   [ Declined Noun g lg → List.iter enterg lg (* nouns *)
      where enterg (n, ln) = List.iter entern ln
      where entern (c, w) =
           let f = Noun_form g n c in
           if c = Voc then
               if morpho_gen.val ∧ is_kridanta entry then ((* f is in Kridv *))
               else add_voca w delta f (* non-generative Voca *)
           else do { add_morph w delta f
                      ; match entry with (* generative ifcs of infinitive bahus *)
                         [ "kaama" (* volition : who wants to do *)
                         | "manas" (* consideration : who thinks about doing *)
                    (* — "zakya" (* possibility : who is able to do *) (* not here since
kridanta, cf enter_form *) *)
                                      → add_morphkama w delta f
                         | _ → ()
                         ]
                    }
```

```
  | Declined Pron g lg → List.iter enterg lg (* pronouns *)
    where enterg (n, ln) = List.iter entern ln
    where entern (c, w) = let f = Noun_form g n c in
                          if c = Voc then add_voca w delta f
                                     else add_morphpro w delta f
  | Conju f lv → List.iter enterv lv
    where enterv (n, ln) = List.iter entern ln
    where entern (p, w) = let v = Verb_form f n p in do
      { add_morphc w delta v aapv
      (* Now we take care of P{6,1,94} when not blocked by P{6,1,89} *)
      (* ex: prejate + (Kazikaa) upelayati prelayati upo.sati pro.sati *)
      ; if morpho_gen.val then
            if entry = "i" ∨ entry = "edh" then () (* P{6,1,89} *)
            else add_morphlopa w delta v
        else ()
      ; (* Now auxiliaries for verbal cvi compounds *)
        if auxiliary entry then add_morphauxi w delta v else ()
      }
  | Indecl k w → match k with
    [ Adv | Part | Conj | Default | Prep | Tas →
                add_morphind w delta (Ind_form k)
    | Interj → add_invoc w delta (Ind_form k)
    | Avya → () (* marked as such in lexicon only if generative avyayiibhava *)
    | Abs | Infl | Nota → () (* no recording in morph tables *)
    (* Abs generated by absolutives of verbs, Infl by flexions of nouns, and our parser
ignores the specific notations of Panini suutras *)
    ]
  | Bare Noun w
  | Bare Pron w → add_morphi w delta Bare_stem
  | Avyayai w → add_morphyai w delta Avyayai_form
  | Avyayaf w → add_morphyaf w delta Avyayaf_form
  | Cvi w → add_morphvi w delta Gati
  | Invar m w → let (_, vi) = m
                and f = Ind_verb m in
                match vi with
    [ Infi → do (* 2 cases: with and without preverbs - saves one phase *)
        { add_morphabsya w delta f aapv
        ; add_morphabstvaa w delta f
        ; if auxiliary entry then add_morphauxiinv w delta f else ()
        }
```

```
          | Absoya (* abso in -ya *) → do
        { add_morphabsya w delta f aapv (* abs-ya: pv or cvii (gati) mandatory *)
        ; if auxiliary entry then add_morphauxiinv w delta f else ()
        }
    | Perpft → add_morphperi w delta f
    (* NB Allows perpft of verbs with preverbs but overgenerates since it allows perpft
followed by a non perfect form of auxiliary *)
    ]
  | Inftu m w → let f = Ind_verb (m, Infi) in
                add_morphinftu w delta f (* infinitive in -tu *)
  | Absotvaa c w → let f = Abs_root c in
                add_morphabstvaa w delta f (* abs-tvaa: no preverb *)
  | Preverb w lw → add_morphp w lw (* w is (normalised) sandhi of lw *)
  | _ → failwith "Unexpected arg to enter"
  ]
;
(* Glitch to add certain undeclinables as ifcs by Make_nouns Phase 2 *)
value enter_ind_ifc entry =
  let delta = Encode.diff_str entry in fun
  [ Indecl k w → add_morph w delta (Ind_form k)
  | _ → failwith "Unexpected arg to enter"
  ]
;
(* enter_form : word → flexion → unit *)
(* 1st argument is a stem generated by derivational morphology, it may have a homo index
computed by Parts.gen_stem. *)
(* enter_form enters in the relevant data bank one of its inflected forms. *)
(* Special treatment to have kridanta forms for auxiliaries, since their lexicalised action
nouns are not recognized as generative, and thus must be skipped to avoid overgeneration.
*)
value enter_form stem =
  let aapv = admits_aa.val (* for phantom forms generation *) in fun
  [ Declined (Krid v root) g lg → List.iter enterg lg
    where enterg (n, ln) = List.iter entern ln
    where entern (c, w) =
      let p = Part_form v g n c in (* We lose the root, and v is used only in Constraints.
Both can be recovered from stem using unique_kridantas *)
      if c = Voc then add_morphpav w stem p aapv
      else do
       { match v with
```

```
          [ (_, Action_noun) → add_morphauxik w stem p (* cvi patch *)
          | _ → do
                { add_morphpa w stem p aapv
                ; if auxiliary root then add_morphauxik w stem p else ()
                }
          ]
        ; if morpho_gen.val then
             if root = "i" ∨ root = "edh" then () (* P{6,1,89} *)
             else add_morphlopak w stem p aapv
          else ()
        }
  | Bare (Krid (_, Action_noun) root) w →
      add_morphauxiick w stem Bare_stem (* cvi patch *)
  | Bare (Krid _ root) w → let f = Bare_stem in do (* losing verbal and root *)
      { add_morphpi w stem f aapv
      ; if auxiliary root then add_morphauxiick w stem f else ()
      }
  | _ → failwith "Unexpected arg to enter_form"
  ]
;
value enter entry = List.iter (enter1 entry)
and enter_forms w = List.iter (enter_form w)
;
```

Module Sandhi

This module defines external sandhi for compound and sentence construction. It proceeds as a finite transducer with two input tapes, one for the right stream of phonemes, the other for the reversal of the left stream. It is deterministic, and thus makes choices in optional situations, so that sandhi is a deterministic function.
This algorithm is only used by service *Sandhier*, while sandhi viccheda proceeds by building tables in *Compile_sandhi* with the help of a clone of this code, then completes the tables with optional rules, making predictive sandhi a truly non-deterministic relation. The code below ought NOT be modified without inspection of its improved clone in module *Compile_sandhi*.

```
open Phonetics; (* finalize visargor *)
open Canon; (* decode *)
```

```
value code str = Encode.code_string str
;
value visargcomp1 first = fun
  [ [] → failwith "left␣arg␣of␣sandhi␣too␣short␣(1)"
  | [ penu :: _ ] → match penu with
        [ 1 → [ −1; 12; first ] (∗ o -1 means erase a ∗)
        | 2 → [ first ] (∗ visarga dropped after aa ∗)
        | _ → [ 43; first ] (∗ visarga goes to r ∗)
        ]
  ]
;
value visargcomp2 = fun (∗ first = ‘r‘, visarga goes to r ∗)
  [ [] → raise (Failure "left␣arg␣of␣sandhi␣too␣short␣(2)")
  | [ penu :: _ ] → match penu with
        [ 1 → [ −1; 12; 43 ] (∗ "a.h+r␣{\R}␣or" -1 means erase a ∗)
        | 2 → [ 43 ] (∗ "aa.h+r␣{\R}␣aar" ∗)
        | c → [ −1; long c; 43 ]
        ]
  ]
;
value visargcompr = fun
  [ [] → failwith "left␣arg␣of␣sandhi␣too␣short␣(r)"
  | [ penu :: _ ] → [ −1; long penu; 43 ]
  ]
;
value visargcompv first (∗ vowel ∗) = fun
  [ [] → failwith "left␣arg␣of␣sandhi␣too␣short␣(v)"
  | [ penu :: _ ] → match penu with
        [ 1 → if first = 1 then [ −1; 12; −1 ] (∗ erase a, o, then avagraha ∗)
              else [ 50; first ] (∗ hiatus ∗)
        | 2 → [ 50; first ] (∗ hiatus ∗)
        | c → [ 43; first ]
        ]
  ]
;
```

(∗ External sandhi core algorithm - *wl* is the reverse left word *wr* is the right word, result is a zip pair (left,right) of words. Caution. This code is used mostly by the Web interface Sandhier, where phantoms may not occur in the input. However, phantom is tested in the code in order to keep consistency with *Compile_sandhi*, which builds the sandhi rules for transducers decorations. This function is also used for glueing preverbs in Roots. ∗)

```
(* unphantom - should instead call Phonetics.un_phantom *)
value uph = fun
  [ -3 -> [ 2 ]
  | -4 -> [ 10 ]
  | -5 -> [ 12 ]
  | -6 -> [ 2; 43 ] (* aar *)
  | r -> [ r ]
  ]
;
value ext_sandhi_pair wl wr =
  match wl with
    [ [] -> failwith "left arg of sandhi empty"
    | [ last :: before ] -> match wr with
  (* Nota Bene : we assume wl to be in final sandhi form except r or s. *)
  (* Thus in the following code all cases last = 34 (* d *) could be omitted *)
  (* for the sandhi viccheda algorithm when inflected forms are known final. *)
        [ [] -> (wl, []) (* no visarga for final s or r *)
        | [ first :: after ] ->
    if vowel last then
       if vowel_or_phantom first then (* first may be *e or *o, thus uph below *)
         let glue =
(* glue is the string replacing [ last; first ] with a special convention: when it starts with -1,
it means the last letter of before is erased, which occurs only when last is visarga *)
           if savarna_ph last first then [ long last ]
           else if avarna last then asandhi first
           else if ivarna last then [ 42 :: uph first ] (* y *)
           else if uvarna last then [ 45 :: uph first ] (* v *)
           else match last with
             [ 7 | 8 -> [ 43 :: uph first ] (* .r -> r *)
             | 10 | 12 (* e o *) ->
                if first = 1 then [ last; -1 ] (* avagraha *)
                else if first = (-11) then [ 1; if last = 10 then 42 else 44; 2 ]
                     (* e+aa+a -¿ ayaa o+aa+a -¿ avaa (preverb aa on augment) *)
                else [ 1 :: [ 50 :: uph first ] ] (* a+hiatus *)
             | 11 (* ai *) -> [ 2 :: [ 50 :: uph first ] ] (* aa+hiatus *)
             | 13 (* au *) -> [ 2 :: [ 45 :: uph first ] ] (* aav *)
             | _ -> let message = "left arg of sandhi end illegal in " in
                     failwith (message ^ decode wl)
             ] in (before, glue @ after)
       else (wl,if first = 23 (* ch *) then [ 22 :: wr ] (* cch *) else wr)
```

```
          (* c optional except when short_vowel last or wl=ā or mā *)
    else (* we assume that last cannot be a phantom and thus is a consonant *)
      let glue =
        if vowel first then
          if visarg last then visargcompv first before (* may start with -1 *)
          else match last with
              [ 21 → match before with
                  [ [] → failwith "left␣arg␣too␣short"
                  | [ v :: rest ] → if short_vowel v then
                                          [ 21 :: [ 21 :: uph first ] ] (* ff *)
                                      else [ 21 :: uph first ]
                  ]
              | 36 → match before with
                  [ [] → failwith "left␣arg␣too␣short"
                  | [ v :: rest ] → if short_vowel v then
                                          [ 36 :: [ 36 :: uph first ] ] (* nn *)
                                      else [ 36 :: uph first ]
                  ]
              | c → [ voiced c :: uph first ] (* t → d, p → b *)
              ]
        else (* both consonant *) match first with
     [ 49 (* h *) →
        if visarg last then visargcomp1 first before
        else match last with
            [ 17 | 19 → [ 19; 20 ] (* k+h → ggh, g+h → ggh *)
            | 27 → [ 29; 30 ] (* ṭ+h → ḍḍh *)
            | 32 | 34 → [ 34; 35 ] (* t+h → ddh, d+h → ddh *)
            | 37 | 39 → [ 39; 40 ] (* p+h → bbh, b+h → bbh *)
            | 41 → [ 14; first ] (* m+h → ṃh *)
                (* but m+hm → mhm and m+hn → mhn preferably (Deshpande) *)
            | c → [ c; first ]
            ]
     | 46 (* ś *) → match last with
            [ 32 | 34 | 22 → [ 22; 23 ] (* t+ś → cch idem d c *)
                         (* optionally 22; 46 c's see compile_sandhi *)
            | 36 → [ 26; 23 ] (* n+ś → ñch (or 26; 46 ñś) *)
            | 41 → [ 14; first ] (* m+ś → ṃś (or ñch optional) *)
            | c → [ if visargor c then 16 else c; first ]
            ]
     | 36 | 41 (* n m *) →
```

```
          if visarg last then visargcomp1 first before
          else match last with
           [ 17 | 21 → [ 21; first ] (* k+n → ṅn 'n+n -¿ 'nn *)
           | 27 | 29 → [ 31; first ] (* ṭ+n → ṇn ḍ+n → ṇn *)
           | 32 | 34 → [ 36; first ] (* t+n → nn d+n → nn *)
           | 37 → [ 41; first ] (* p+n → mn *)
           | 41 → [ 14; first ] (* m+n → ṃn *)
           | c → [ c; first ] (* ṅ+n → ṅn etc. *)
           ]
  | 47 | 48 (* ṣ s *) →
            match last with
           [ 41 → [ 14; first ] (* m+s → ṃs *)
           | 34 → [ 32; first ] (* d+s → ts *)
           | c → [ if visargor c then 16 else c; first ]
           ]
  | 37 | 38 | 17 | 18 (* p ph k kh *) →
            match last with
           [ 41 → [ 14; first ] (* m+p → ṃp *)
           | 34 → [ 32; first ] (* d+p → tp *)
           | c → [ if visargor c then 16 else c; first ] (* s+k → ḥk but optional ṣk *)
           ]
  | 44 (* l *) →
          if visarg last then visargcomp1 first before
          else match last with
           [ 32 | 34 → [ 44; 44 ] (* t+l → ll d+l → ll *)
           | 36 | 41 → [ 44; 15; 44 ] (* n+l → ll̃ (candrabindu) *)
           | c → [ voiced c; 44 ]
           ]
  | 42 | 45 (* y v *) →
          if visarg last then visargcomp1 first before
          else match last with
           [ 41 → [ 14; first ] (* m+y → ṃy *)
           | c → [ voiced c; first ]
           ]
  | 43 (* r *) →
          if visarg last then visargcomp2 before
          else match last with
           [ 41 → [ 14; 43 ] (* m+r → ṃr *)
           | 43 → visargcompr before (* Gonda §16 *)
           | c → [ voiced c; first ]
```

```
          ]
  | 39 | 40 | 34 | 35 | 19 | 20 (* b bh d dh g gh *) →
          if visarg last then visargcomp1 first before
          else match last with
            [ 41 → [ 14; first ] (* m+b → ṃb == mb *)
            | c → [ voiced c; first ]
            ]
  | 29 | 30 (* ḍ ḍh *) →
          if visarg last then visargcomp1 first before
          else match last with
            [ 41 → [ 14; first ] (* m+ḍ → ṃḍ == ṇḍ *)
            | 32 | 34 → [ 29; first ] (* t+ḍ → ḍḍ d+ḍ → ḍḍ *)
            | 36 → [ 31; first ] (* n+ḍ → ṇḍ *)
            | c → [ voiced c; first ]
            ]
  | 24 | 25 (* j jh *) →
          if visarg last then visargcomp1 first before
          else match last with
            [ 41 → [ 14; first ] (* m+j → ṃj == ñj *)
            | 32 | 34 → [ 24; first ] (* t+j → jj d+j → jj *)
            | 36 → [ 26; first ] (* n+j → ñj *)
            | c → [ voiced c; first ]
            ]
  | 32 | 33 (* t th *) → match last with
            [ 41 → [ 14; first ] (* m+t → ṃt == nt *)
            | 36 → [ 14; 48; first ] (* n+t → ṃst *)
            | 34 → [ 32; first ] (* d+t → tt *)
            | c → [ if visargor c then 48 else c; first ] (* s+t → st *)
            ]
  | 27 | 28 (* ṭ ṭh *) → match last with
            [ 41 → [ 14; first ] (* m+ṭ → ṃṭ == ṇṭ *)
            | 32 | 34 → [ 27; first ] (* t+ṭ → ṭṭ d+ṭ → ṭṭ *)
            | 36 → [ 14; 47; first ] (* n+ṭ → ṃṣṭ *)
            | c → [ if visargor c then 47 else c; first ]
            ]
  | 22 | 23 (* c ch *) → match last with
            [ 41 → [ 14; first ] (* m+c → ṃc == ñc *)
            | 32 | 34 → [ 22; first ] (* t+c → cc d+c → cc *)
            | 36 → [ 14; 46; first ] (* n+c → ṃśc *)
            | c → [ if visargor c then 46 else c; first ]
```

```
                ]
  (* — 31 (* ṇ *) missing c+.n = f.n TODO *)
      | c → failwith ("illegal start of right arg of sandhi in " ^ decode wr)
      ] (* match first *) in (* let glue *)
         let (w1, w2) = match glue with
             [ [] → failwith "empty glue"
             | [ −1 :: rest ] → match before with
                    [ [] → failwith "left arg too short"
                    | [ _ (* a *) :: init ] → (init, rest)
                    ]
             | _ → (before, glue)
             ] in (w1, w2 @ after)
         ] (* match wr *)
     ] (* match wl *)
;
value ext_sandhi0 wl wr = (* No normalization *)
  let (w1, w2) = ext_sandhi_pair wl wr in
  List2.unstack w1 w2 (* w1 is pasted as left context of w2 *)
;
(* Only used in stand-alone module Sandhier; argument is rev of word *)
value final_sandhi = fun
  [ [] → failwith "Empty input Sandhi"
  | [ last :: rest ] when visargor last
        → List.rev [ 16 :: rest ] (* final visarga *)
  | rw → List.rev (finalize rw)
  ]
;
```

External sandhi - Reference version - used in *Roots.follow*
esandhi : string → string → word

```
value esandhi left right =
  let wl = List.rev (code left)
  and wr = code right in
  Encode.normalize (ext_sandhi0 wl wr) (* normalization *)
;
(* Unused directly; copied in Compile_sandhi.match_sandhi *)
(* e_sandhi : string → string → string *)
value e_sandhi left right = decode (esandhi left right)
;
(* Used in Roots.follow and Make_preverbs.preverbs_etym *)
```

```
value pv_sandhi left right =
  if left ="pra" ∧ right ="ni" then "pra.ni" (* retroflexion *)
  else e_sandhi left right
and pv_sandhi0 wl wr =
  let rwl = Word.mirror wl in
  if rwl = code "pra" ∧ wr = code "ni" then code "pra.ni" (* retroflexion *)
  else Encode.normalize (ext_sandhi0 wl wr) (* normalization *)
;
(* tests *)
assert (e_sandhi "vane" "iva" = "vana_iva");
assert (e_sandhi "na" "chinatti" = "nacchinatti");
assert (e_sandhi "tat" "zariiram" = "tacchariiram");
assert (e_sandhi "tat" "lebhe" = "tallebhe");
assert (e_sandhi "tat" "zrutvaa" = "tacchrutvaa");
assert (e_sandhi "tat" "jayati" = "tajjayati");
assert (e_sandhi "tat" "mitram" = "tanmitram");
assert (e_sandhi "azvas" "asti" = "azvo'sti");
assert (e_sandhi "azvas" "iva" = "azva_iva");
assert (e_sandhi "punar" "iva" = "punariva");
assert (e_sandhi "punar" "suuti" = "puna.hsuuti");
assert (e_sandhi "punar" "janman" = "punarjanman");
assert (e_sandhi "api" "avagacchasi" = "apyavagacchasi");
assert (e_sandhi "nanu" "upavizaama.h" = "nanuupavizaama.h");
assert (e_sandhi "ubhau" "aagacchata.h" = "ubhaavaagacchata.h");
assert (e_sandhi "katham" "smarati" = "katha.msmarati");
assert (e_sandhi "sam" "hraad" = "sa.mhraad");
assert (e_sandhi "dvi.t" "hasati" = "dvi.d.dhasati");
assert (e_sandhi "ud" "h.r" = "uddh.r");
assert (e_sandhi "tat" "hema" = "taddhema");
assert (e_sandhi "taan" "tu" = "taa.mstu");
assert (e_sandhi "nara.h" "rak.sati" = "narorak.sati");
assert (e_sandhi "punar" "rak.sati" = "punaarak.sati");
assert (e_sandhi "gaayan" "aagacchati" = "gaayannaagacchati");
assert (e_sandhi "vaak" "me" = "vaafme");
assert (e_sandhi "vaag" "hasati" = "vaagghasati");
assert (e_sandhi "bahis" "k.r" = "bahi.hk.r"); (* aussi "bahi.sk.r" *)
assert (e_sandhi ".sa.t" "naam" = ".sa.nnaam"); (* and not ".sa.n.naam" *)
assert (e_sandhi "tat" "namas" = "tannamas"); (* but "tadnamas" also correct *)
assert (e_sandhi "kim" "hmalayati" = "ki.mhmalayati"); (* but "kimhmalayati" also
correct *)
```

assert (*e_sandhi* "kim" "hnute" = "ki.mhnute"); (* but "kinhnute" also correct (metathesis) *)
assert (*e_sandhi* "tat" "mitram" = "tanmitram");
assert (*e_sandhi* "devaan" "z.r.noti" = "devaa~nch.r.noti");

Remark. *e_sandhi* is used for preverbs, and the existence of *e and *o guarantees that (*external_sandhi x* (*external_sandhi pre y*)) is the same as (*external_sandhi* (*external_sandhi x pre*) *y*): NB. form "aa—ihi" with *e-phantom generated by Inflected.

```
assert (e_sandhi "iha" "aa|ihi" = "ihehi"); (* e-phantom elim *)
assert (e_sandhi "iha" "aa" = "ihaa");
assert (e_sandhi "ihaa" "ihi" = "ihehi");
(* Idem for *o : fake sandhi "aa" "upa" = "aa|upa") generated by Inflected. *)
assert (e_sandhi "zoka" "aa|rta" = "zokaarta");
```

Context-sensitive irregularities

```
value external_sandhi left right =
  if left = "sas" ∨ left = "sa.h" then
      match code right with
          [ [] → "sa.h"
          | [ first :: after ] →
               e_sandhi (if vowel first then "sa.h" else "sa") right
          ]
  else e_sandhi left right
;
(* Sandhier version, takes a revword and a word, and returns a word *)
value ext_sandhi rvlword rword =
  let left = match rvlword with
       [ [ 48 :: [ 1; 48 ] ] | [ 16 :: [ 1; 48 ] ] → match rword with
                  [ [] → [ 16 :: [ 1; 48 ] ]
                  | [ first :: after ] →
                        if vowel first then [ 16 :: [ 1; 48 ] ] else [ 1; 48 ]
                  ]
       | l → l
       ] in ext_sandhi0 left rword (* does not finalize r or s into .h *)
;
value after_dual_sandhi left right =
  match List.rev (code left)
  with [ [] → failwith "left␣arg␣of␣sandhi␣empty"
       | [ last :: _ ] →
          if last = 4 (* ii *) ∨ last = 6 (* uu *) ∨ last = 10 (* e *)
              then (left ^ "_" ^ right) (* hiatus *)
```

```
        else e_sandhi left right
      ]
;
(* tests *)
assert (external_sandhi "sas" "gaja.h" = "sagaja.h");
assert (external_sandhi "sas" "aacaarya.h" = "sa_aacaarya.h");
assert (external_sandhi "sas" "azva.h" = "so'zva.h");
assert (external_sandhi "sas" "" = "sa.h");

assert (after_dual_sandhi "tephale" "icchaama.h" = "tephale_icchaama.h");
```

Also external sandhi does not occur after interjections and is optional after initial vocatives - TODO

Module Sandhier

Sandhi Engine cgi
It gives the most common sandhi solution, but not the optional forms
This stand-alone module is not used by the rest of the system

```
open Sandhi; (* final_sandhi ext_sandhi *)
open Int_sandhi; (* int_sandhi *)
open Html;
open Web; (* ps pl abort etc. *)
open Cgi;

value title = h1_title (if narrow_screen then "Sandhi"
                        else "The Sandhi Engine")
and meta_title = title "Sanskrit Sandhi Engine"
;
value display_rom_red s = html_red (Transduction.skt_to_html s)
and display_dev_red s = html_devared (Encode.skt_to_deva s)
;
value sandhi_engine () = do
  { pl http_header
  ; page_begin meta_title
  ; pl (body_begin (background Chamois))
  ; pl title
  ; let query = Sys.getenv "QUERY_STRING" in
    let env = create_env query in
   try
    let url_encoded_left = get "l" env ""
```

```
and url_encoded_right = get "r" env ""
and url_encoded_kind = get "k" env "external"
and translit = get "t" env Paths.default_transliteration
and lex = get "lex" env Paths.default_lexicon in
let left_str = decode_url url_encoded_left
and right_str = decode_url url_encoded_right
and lang = language_of lex
and encode = Encode.switch_code translit in
let left_word = encode left_str
and right_word = encode right_str in
let rleft_word = Word.mirror left_word
and final = (right_word = []) in
let result_word = match url_encoded_kind with
    [ "external" →
        if final then final_sandhi rleft_word
        else ext_sandhi rleft_word right_word
    | "internal" →
        if final then raise (Control.Fatal "Empty␣right␣component")
        else int_sandhi rleft_word right_word
    | _ → raise (Control.Fatal "Unexpected␣kind")
    ] in
let kind = if final then "final" else url_encoded_kind in
let left = Canon.decode left_word (* = left_str *)
and right = Canon.decode right_word (* = right_str *)
and result = Canon.decode result_word in do
{ ps (span_begin C1)
; ps ("The␣" ^ kind ^ "␣sandhi␣of␣")
; ps (display_rom_red left)
; if final then () else do
      { ps "␣and␣"
      ; ps (display_rom_red right)
      }
; ps "␣is␣"
; ps (display_rom_red result)
; ps span_end (* C1 *)
; ps center_begin
; ps (span_skt_begin Deva20c)
; ps (display_dev_red left)
; ps "␣|␣"
; if final then () else ps (display_dev_red right)
```

```
      ; ps " = "
      ; ps (display_dev_red result)
      ; ps span_end (* Deva20c *)
      ; ps center_end
      ; ps (span_begin C1)
      ; ps "NB. Other sandhi solutions may be allowed"
      ; ps span_end (* C1 *)
      ; page_end lang True
      }
    with [ Stream.Error _ → raise Exit
         | Not_found → failwith "parameter missing ?"
         ]
  }
;
value safe_engine () =
  let abor = abort default_language in
  try sandhi_engine () with
  [ Sys_error s → abor Control.sys_err_mess s (* file pb *)
  | Stream.Error s → abor Control.stream_err_mess s (* file pb *)
  | Encode.In_error s → abor "Wrong_input " s
  | Invalid_argument s → abor Control.fatal_err_mess s (* sub *)
  | Failure s → abor Control.fatal_err_mess s (* anomaly *)
  | Not_found → abor Control.fatal_err_mess "assoc" (* assoc *)
  | End_of_file → abor Control.fatal_err_mess "EOF" (* EOF *)
  | Control.Fatal s → abor "Wrong parameters " s
  | Exit → abor "Wrong character in input - "
                              "check input convention" (* Sanskrit *)
  | _ → abor Control.fatal_err_mess "Unexpected anomaly"
  ]
;
safe_engine ()
;
```

Module Pada

Pada defines the allowed padas (Para, Atma or Ubha) for a given combination of root, gana, and upasarga
It is used at conjugation computation time by Verbs, in order to generate root forms for attested lexicalizations of root and gana (over all possible upasarga usages) and at segmen-

tation time, to filter out by Dispatcher the non attested combinations of gana, pada and upasarga

```
type voices = (* permitted padas in present system *)
  (* NB. These are distinctions within the active voice, as opposed to passive ("karma.ni␣prayoga").
Atma is called "middle" by Western grammarians. *)
  [ Para (* parasmaipadin usage only - generated as Dictionary.Active *)
  | Atma (* aatmanepadin usage only - generated as Dictionary.Middle *)
  | Ubha (* ubhayapada admits both schemes - default *)
  ]
;
exception Unattested (* when a root/pada is attested only for some pvs *)
;
value voices_of = fun
  (* Simplification: invariant when prefixing by preverbs *)
  [ "ak.s" | "afg" | "aj" | "a.t" | "at" | "ad#1" | "an#2" | "am"| "argh"
  | "ard" | "av" | "az#2" | "as#1" | "as#2" | "aap" | "ifg" | "in" | "ind"
  | "inv" | "il" | "i.s#2" | "iifkh" | "iir.s" | "uk.s" | "ujjh" | "u~nch"
  | "und" | "umbh" | "u.s" | ".rc#1" | ".rdh" | ".r.s" | "ej" | "kas" | "kiirt"
  | "kiil" | "ku.t" | "ku.n.th" | "kunth" | "kup" | "kul" | "ku.s" | "kuuj"
  | "k.rt#1" | "k.rz" | "krand" | "kru~nc#1" | "krudh#1" | "kruz" | "klam"
  | "klid" | "kliz" | "kvath" | "k.sar" | "k.sal" | "k.saa" | "k.si" | "k.sii"
  | "k.su" | "k.sudh#1" | "k.subh" | "k.svi.d" | "kha~nj#1" | "khaad" | "khid"
  | "khel" | "khyaa" | "gaj" | "gad" | "garj" | "gard" | "gal" | "gaa#1"
  | "gaa#2" | "gu~nj" | "gu.n.th" | "gup" | "gumph" | "g.rdh"
  | "g.rr#2" | "granth" | "grah" | "glai" | "ghas" | "ghu.s" | "gh.r" | "gh.r.s"
  | "ghraa" | "cakaas" | "ca.t" | "cand" | "cam" | "car" | "cal" | "cit#1"
  | "cumb" | "c.rt" | "chur" | "ch.rd" | "jak.s" | "jap" | "jabh#2" | "jam"
  | "jalp" | "jas" | "jaag.r" | "jinv" | "jiiv" | "jvar" | "jval" | "tak"
  | "tak.s" | "ta~nc" | "tam" | "tarj" | "tup" | "tu.s" | "t.rp#1" | "t.r.s#1"
  | "t.rh" | "t.rr" | "tyaj#1" | "tras" | "tru.t" | "tvak.s" | "tsar" | "da.mz"
  | "dagh" | "dabh" | "dam#1" | "dal" | "das" | "dah#1" | "daa#2" | "daa#3"
  | "diiv#1" | "du" | "du.s" | "d.rp" | "d.rbh" | "d.rz#1" | "d.rh" | "d.rr"
  | "dhyaa" | "draa#1" | "dru#1" | "druh#1" | "dham" | "dhaa#2" | "dhru"
  | "dhvan" | "dhv.r" | "na.t" | "nad" | "nand" | "nam" | "nard" | "naz#1"
  | "nind" | "nu#1" | "n.rt" | "pa.t" | "pat#1" | "path" | "paa#1" | "paa#2"
  | "pii" | "piz#1" | "pi.s" | "pu.t" | "p.r#1" | "p.r#2" | "p.r.s" | "p.rr"
  | "praa#1" | "phal" | "bal" | "b.rh#1" | "b.rh#2" | "bha~nj" | "bha.n"
  | "bha.s" | "bhas" | "bhaa#1" | "bhii#1" | "bhuj#1" | "bhuu#1" | "bhuu.s"
  | "bhram" | "majj" | "ma.n.d" | "mad#1" | "manth" | "mah" | "maa#3" | "mi.s"
  | "mih" | "miil" | "mu.s#1" | "muh" | "muurch" | "m.r.d" | "m.rz" | "mnaa"
```

```
  | "mre.d" | "mlaa" | "mluc" | "mlecch" | "yabh" | "yam" | "yas" | "yaa#1"
  | "yu#2" | "ra.mh" | "rak.s" | "ra.n" | "rad" | "radh" | "raa#1" | "raadh"
  | "ri.s" | "ru" | "ruj#1" | "rudh#1" | "ru.s#1" | "ruh#1" | "lag" | "lafg"
  | "lap" | "lal" | "las" | "laa" | "laa~nch" | "likh" | "liz" | "lu.n.th"
  | "lubh" | "lul" | "vak.s" | "vac" | "vaj" | "va~nc" | "van" | "vam" | "var.n"
  | "valg" | "vaz" | "vas#1" | "vaa#2" | "vas#4" | "vaa~nch" | "vid#1" | "vidh#1"
  | "vi.s#1" | "vii#1" | "v.rj" | "v.r.s" | "v.rh" | "ven" | "vyac"| "vyadh"
  | "vraj" | "vrazc" | "za.ms" | "zak" | "zam#1" | "zam#2" | "zal" | "zaz"
  | "zas" | "zaas" | "zi.s" | "ziil" | "zuc#1" | "zudh" | "zumbh" | "zu.s"
  | "zuu" | "z.rr" | "zcut#1" | "zraa" | "zli.s" | "zvas#1" | ".s.thiiv"
  | "sad#1" | "sap#1" | "saa#1" | "sidh#1" | "sidh#2" | "siiv"
  | "sur" | "s.r" | "s.rj#1" | "s.rp" | "skand" | "skhal" | "stan" | "stubh"
  | "sthag" | "snaa" | "snih#1" | "snu" | "snuh#1" | "sp.r" | "sphal" | "sphu.t"
  | "sphur" | "sm.r" | "svan" | "svap" | "svar#1" | "svar#2" | "ha.th"
  | "haa#1" | "hi#2" | "hi.ms" | "h.r.s" | "hras" | "hrii#1" | "hval"
  | "maarg" (* root rather than nominal verb *)
(*— "viz1" Atma needed for eg nivizate P{1,3,17} *)
(*— "k.s.nu" Atma needed for sa.mk.s.ute P{1,3,65} *)
(*— "krii.d" Atma needed for aakrii.date P{1,3,21} *)
(*— "g.rr1" Atma needed for avag.rr sa.mg.rr P{1,3,51-52} *)
(*— "ji" Atma needed for eg vijayate paraajayate P{1,3,19} *)
(*— "jyaa1" Atma needed for jiiyate *)
(*— "kan" Atma needed for kaayamaana *)
(*— "gam" Atma needed for sa.mgacchate *)
(*— "van" Atma needed for vanute *)
(*— "mah" Atma needed for pft. maamahe *)
(*— "cit#1" Atma needed for pft. cikite *)
(*— "zram" Atma needed for vizramate *)
(*— "kaafk.s" — "han1" occur also in Atma in BhG: kaafk.se hani.sye *)
(*— "has" Atma needed for hasate *)
(*— "uc" Atma needed for uuce *)
(*— "zu.s" Atma for zu.syate WR epic *)
(*— "a~nj" also Atma afkte — "naath" "praz" "sp.rz#1" idem *)
(*— Doubt: "bhuu1" could also be Atma bhavate *)
(*— "zru" could be Atma in Vedic eg z.r.nu.sva *)
      → Para (* active only *)
  | "az#1" | "aas#2" | "indh" | "iik.s" | "ii.d" | "iir" | "iiz#1"
  | "ii.s" | "iih" | "edh" | "katth" | "kam" | "kamp" | "kaaz" | "kaas#1"
  | "kuu" | "k.rp" | "k.lp" (* but Henry: cak.lpur "ils␣s'arrangèrent" *)
  | "knuu" | "klav" | "k.sad" | "galbh" | "gur" | "glah"
```

```
  | "gha.t" | "jabh#1" | "ju.s#1" | "j.rmbh" | ".damb" | ".dii" | "tandr"
  | "tij" | "trap" | "trai" | "tvar" | "dak.s" | "day" | "diik.s" | "diip"
  | "d.r#1" | "dhii#1" | "dhuk.s" | "pa.n" | "pad#1" | "pi~nj"
  | "pyaa" | "prath" | "pru" | "plu" | "ba.mh" | "baadh" | "bha.n.d" | "bhand"
  | "bhaa.s" | "bhraaj" | "ma.mh" | "man" | "mand#1" | "yat#1"
  | "rabh" | "ruc#1" | "lajj" | "lamb" | "lii" | "loc" | "vand"
  | "vas#2" | "vaaz" | "vip" | "v.rdh#1" | "ve.s.t" | "vrii.d" | "zafk" | "zad"
  | "zi~nj" | "zii#1" | "zrambh" | "zlaagh" | "zvit#1" | "sac" | "sev"
  | "styaa" | "spand" | "spardh" | "spaz#1" | "sphaa" | "sra.ms"
  | "sva~nj" | "haa#2" | "hu.n.d" | "h.r#2" | "hnu" | "hraad" | "hlaad"
(*— "m.r" Ubha needed for non present tenses - see P{1,3,61} for exact rule *)
(*— "smi" Ubha needed for smitavat *)
(*— "bhuj2" Ubha needed for bhunakti to govern *)
(*— "gaah" Ubha needed for gaahet epics *)
(*— "k.sam" Ubha needed for k.samati epics *)
(*— "yudh1" Ubha needed for yudhya BhG *)
   (* DRP restriction: "dyut1" *)
       → Atma (* "deponent" verbs: middle only *)
  | _ → Ubha (* default *)
  (* Attested Ubha (over all ga.nas) :  "a~nc" |  "arth" |  "arh" |  "i" |  "i.s#1" |
 "uc" | "uurj#1" | "uuh" | ".r"  | ".rj" | "ka.n.d" | "kal" | "ka.s" | "ku.t.t" |
 "ku.n.d" | "k.r#1" | "k.r#2"  | "k.r.s" | "kram" | "krii#1" | "krii.d" | "k.san" |
 "k.sap#1" | "k.sam"  | "k.sal" | "k.sip" | "k.sud" | "k.s.nu" | "khan" | "gam" |
 "garh"  | "gaah" | "guh" | "g.rr#1" | "gras" | "gha.t.t"  | "cat" | "carc" |
 "ci" | "cint" | "cud" | "ce.s.t" | "cyu" | "chad#1"  | "chand" | "chid#1" |
 "jan" | "juu" | "j~naa#1" | "jyaa#1" | "jyut" | "ta.d"  | "tan#1" | "tan#2" |
 "tap" | "tud#1" | "tul" | "t.rd" | "traa" | "daa#1"  | "daaz#1" | "diz#1" | "dih" |
 "duh#1" | "dev#1" | "draa#2" | "dvi.s#1"  | "dhaa#1" | "dhaav#1" | "dhaav#2" |
 "dhuu#1" | "dh.r" | "dhva.ms" | "nah"  | "naath" | "nij" | "nii#1" | "nud" |
 "pac" | "paz" | "pa.th" | "pii.d"  | "pu.s#1" | "puu#1" | "puuj" | "puuy" |
 "p.rth" | "prii" | "pru.s#1"  | "budh#1" | "bruu" | "bhak.s" | "bhaj" | "bharts" |
 "bhaas#1" | "bhid#1"  | "bhuj#2" | "bh.r" | "bh.rjj" | "mantr" | "maa#4" | "mi" |
 "mith" | "mil"  | "mii" | "muc#1" | "mud#1" | "m.r" | "m.rj" | "m.rdh" | "m.r.s" |
 "yaj#1"  | "yam" | "yaac" | "yu#1" | "yuj#1" | "yudh#1" | "rac" | "ra~nj" | "ram"
| "rah" | "raaj#1" | "ri" | "ric" | "rud#1" | "rudh#2" | "lafgh" | "lak.s"
| "labh" | "la.s" | "lip" | "lih#1" | "lup" | "luu#1" | "vad" | "van" | "vap#1"
| "vap#2" | "val" | "vah#1" | "vaa#3" | "vic" | "vij" "viz#1" | "viij"  | "v.r#2" |
 "v.rt#1" | "vyath" | "vyaa" | "vrii" | "zap" | "zaa" | "zu.s" | "zubh#1" | "zyaa" |
 "zram" | "zri" | "zru" | "sru" | "san#1" | "sa~nj"  | "sah#1" | "sic" | "su#2" |
 "suud" | "stambh" | "stu" | "st.rr" | "sthaa#1"  | "sp.rz#1" | "sp.rh" | "smi" |
```

```
 "syand" | "svad" | "had" | "hikk" | "hu"  | "huu" | "h.r#1"  *)
  (* + corr. "pa.th" — "sthaa1" — "praz" — "k.rr" — "p.rc" — "bandh" *)
  (* NB. "ah" "rip" "vadh" have no pr, "mand2" is fictitious *)
  (* "iiz1", "lii" and "knuu" allowed Para in future *)
  ]
;
```

(* List of roots that admit different padas for distinct ganas: as2 1U 4P (* 4P Vedic - may overgenerate ? *) i 1A 2P 4A 5P .r 1U 3P 5P kuc 1U 6P k.r.s 1P 6U ghuur.n 1A 6P jan 4A 1U j.rr 1U 4P jyaa1 4A 9P .damb 1A 10P (vi-) tap 1U 4A daa1 2P 1U 3U draa2 2P 4U dh.r.s 1U 5P nij 2A 3U pu.s1 4U 9P budh1 1P 4A bhra.mz 1A 4P man 1U 4U 8A maa1 3A 2P mid 1A 4P 1OP mii 9P 4A m.r 4A other tenses P m.rj 1U 2P 6U m.rd1 9P 1U ri 4A 9U ric 4A 7P rud1 2P 1U 6U van 1P 8U vid2 2A 6U 7A v.r1 1P 5U zaa 3U 4P su2 1P 2P 5U suu1 1P 6P 2A stambh 1U 5P 9P svid2 1A 4P *)
(* More precise selection for present system *)
(* NB This will drive generation of verbal forms by Verbs. It may generate forms not listed in the lexicon root entry, but needed for use with some preverbs, indicated in *voices_of_pv* below. Incorrect associations will be captured at Reader time by Dispatcher. *)

```
value voices_of_gana g root = match g with
 [ 1 → match root with
        [ "cur" | "budh#1" | "van" | "v.r#1" | "su#2"
        | "suu#1"
            → Para
        | "gave.s" | "gha.t.t" | "ghuur.n" | ".damb" | "bhra.mz" | "mid"
        | "mok.s" | "lok" | "svid#2"
            → Atma
        | "i" | "i.s#1" | ".r" (* ".r" Atma for pv sam P{1,3,29} also "tap" *)
        | "j.rr" | "tap" | "daa#1" | "dh.r.s" | "as#2" | "kuc" | "k.r.s"
        | "m.rj" | "m.rd#1" | "rud#1" | "stambh"
            → Ubha
        | "kliiba" → Atma (* denominative verb *)
        | _ → voices_of root (* man U (epic P) *)
        ]
 | 2 → match root with
       [ "daa#1" | "dyaa" | "draa#2" | "maa#1" | "m.rj" | "rud#1" | "su#2"
            → Para
        | "nij" | "vid#2" | "suu#1" → Atma
        | _ → voices_of root
        ]
 | 3 → match root with
        [ ".r" → Para
```

```
          | "maa#1" → Atma
          | "daa#1" | "nij" → Ubha
          | _ → voices_of root
          ]
  | 4 → match root with
          [ "as#2" | "j.rr" | "bhra.mz" | "mid" | "zaa"
          | "svid#2" → Para
          | "i" | "jan" | "jyaa#1" | "tap" | "draa#2" | "budh#1" | "mii" | "ri"
          | "ric" | "m.r" | "vrii" → Atma
          | "pu.s#1" (* — "raadh" Bergaigne vedic *) → Ubha
          | _ → voices_of root
          ]
  | 5 → match root with
          [ "i" | ".r" | "dh.r.s" | "raadh" | "stambh" → Para
          | "v.r#1" | "su#2" → Ubha
          | _ → voices_of root
          ]
  | 6 → match root with
          [ "kuc" | "ghuur.n" | "suu#1" → Para
          | "k.r.s" | "m.rj" | "rud#1" | "vid#2" → Ubha
          | _ → voices_of root
          ]
  | 7 → match root with
          [ "vid#2" → Atma
          | _ → voices_of root
          ]
  | 8 → match root with
          [ "man" → Atma
          | _ → voices_of root
          ]
  | 9 → match root with
          [ "jyaa#1" | "pu.s#1" | "mii" | "m.rd#1" | "ri" | "vrii" | "stambh"
              → Para
          | _ → voices_of root
          ]
  | 10 → match root with
          [ "gha.t.t" | ".damb" | "mid" | "mok.s" | "lak.s" | "lok"
              → Para
          | _ → voices_of root (* other denominatives will take Ubha as default *)
          ]
```

```
 | _ → voices_of root (* in particular, for non-present forms, without gana *)
 ]
;
```

Refining with potential preverb

```
value voices_of_pv upasarga gana = fun
    (* Beware: gana only used for "tap" "i" ".r" but gana=0 for non-present forms *)
(* Paninian requirements *)
[ "zru" | "gam" | "svar" | "vid#1" (* — "praz" *) →
                if upasarga = "sam" then Atma else Para (* P{1,3,29} *)
(* "praz" used in Atma with aa- but also without pv in epics (MW) *)
| ".r" | "car" → if upasarga = "sam" then Ubha else Para (* P{1,3,54} *)
| "viz#1" → if upasarga = "ni" then Atma else Para (* P{1,3,17} *)
| "k.s.nu" → if upasarga = "sam" then Atma else Para (* P{1,3,65} *)
| "huu" → match upasarga with
              [ "ni" | "sam" | "upa" | "vi" → Atma (* P{1,3,30} *)
              | "aa" → Ubha (* P{1,3,31} *)
              | _ → Para
              ]
| "yam" → match upasarga with
              [ "aa" | "upa" → Ubha
              | _ → Para (* P{1,3,28} and P{1,3,56} *)
              ]
| "vah#1" → if upasarga = "pra" then Para else Ubha (* P{1,3,81} *)
(* Complex vad (* P{1,3,47-50,73} *) depends on meaning - now all Ubha — "vad" -
¿ match upasarga with "" | "sam" → Ubha (× \Pan{1,3,47,48} × ) | "anu" →
 Ubha (× \Pan{1,3,49} × ) | "pra" | "vipra" → Ubha (× \Pan{1,3,49} × ) |
 "apa" → Ubha (× \Pan{1,3,73} × ) | _ → Para *)
| "g.rr#1" → match upasarga with
              [ "ava" → Atma (* P{1,3,51} *)
              | "sam" → Ubha (* P{1,3,52} *)
              | _ → Para
              ]
| "ji" → match upasarga with
            [ "vi" | "paraa" → Atma (* P{1,3,19} *)
            | _ → Ubha (* was Para but "satyam␣eva␣jayate" *)
            ]
| "krii.d" → match upasarga with
            [ "aa" | "anu" | "pari" → Atma (* P{1,3,21} *)
            | "sam" → Ubha (* P{1,3,21} vaartikaa *)
```

```
          | _ → Para (* "" — "vi" *)
          ]
| "m.rz" → if upasarga = "pari" then Para else Ubha (* P{1,3,82} *)
| "tap" when gana = 1 → match upasarga with
                         [ "ut" | "vi" → Ubha
                         | _ → Para (* P{1,3,27} *)
                         ]
| "i" → match gana with
          [ 2 | 0 → match upasarga with
                      [ "adhi" | "anu" | "abhi" → Ubha
                      | _ → Para
                      ]
          | _ (* 1 — 4 *) → match upasarga with
                      [ "" → Atma
                      | "antar" | "ut" → Para (* gana 1 antarayati udayati *)
                      | _ → raise Unattested
                      ]
          ]
| "zii#1" → if upasarga = "sam" then Ubha else Atma
| "zram" → if upasarga = "vi" then Ubha (* epic vizramate *) else Para
| "k.r#1" → match upasarga with
              [ "anu" | "paraa" → Para (* P{1,3,79} *)
              | _ → Ubha
              ]
| "krii#1" → match upasarga with
              [ "vi" → Ubha (* vikrii.naati/vikrii.niite *)
              | "pari" | "ava" → Atma
              | _ → Para (* P{1,3,18} *)
              ]
(* Next four equivalent to marking "unused" in lexicon *)
| "ta~nc" | "saa#1" | "zam#2" | "zal" (* — "khyaa" ? *) →
   match upasarga with
   [ "" → raise Unattested (* thus braa.hmasya "Ô␣Brahmane,␣tue" unrecognized *)
   | _ → Para
   ]
| "loc" | "zrambh" | "hnu" → match upasarga with
   [ "" → raise Unattested
   | _ → Atma
   ]
| ".damb" → match upasarga with
```

```
    [ "vi" → Ubha
    | _ → raise Unattested
    ]
(* Usage, MW *)
| "gha.t.t" → if gana = 1 then
                  if upasarga = "" then raise Unattested
                  else Atma (* only "vi" — "sam", NOT "" *)
              else (* gana = 10 *) Para
| "i.s#1" when gana = 1 → match upasarga with
          [ "" → raise Unattested
          | _ → Ubha
          ]
| "mantr" → match upasarga with
    [ "" → Ubha
    | "aa" | "ni" → Atma
    | "anu" | "sam" → Para
    | _ → raise Unattested
    ]
| "arth" → match upasarga with
    [ "" → Ubha
    | "aa" | "ni" → Atma
    | "anu" | "sam" → Para
    | _ → raise Unattested
    ]
| root → voices_of_gana gana root
]
;
```

Interface for module Nouns

```
open Skt_morph;
open Morphology; (* inflected_map *)

type declension_class =
  [ Gender of gender (* declined substantive, adjective, number, pronoun *)
  | Ind of ind_kind (* indeclinable form *)
  ]
and nmorph = (string × declension_class)
;
exception Report of string
```

```
;
value compute_decls : Word.word → list nmorph → unit;
value compute_extra_iic : list string → unit;
value compute_extra : list string → unit;
value enter_extra_ifcs : unit → unit;
value enter_extra_iifcs : unit → unit;
value fake_compute_decls :
      nmorph → string → ( inflected_map (* nouns *)
                        × inflected_map (* pronouns *)
                        × inflected_map (* vocas *)
                        × inflected_map (* iics *)
                        × inflected_map (* adverbs ifcs *)
                        );
(* Used in Interface for User-aid *)
value extract_current_caches : string → ( inflected_map × inflected_map )
;
```

Module Nouns

Computes the declensions of substantives, adjectives, pronouns, numerals and records the nominal inflected forms in databases by *Inflected.enter*. It is called from *Make_nouns* nominal generation process.

```
open List; (* exists, iter *)
open Word; (* mirror *)
open Skt_morph (* morphology datatypes *);
open Phonetics; (* finalize, finalize_r *)
open Inflected; (* Declined, Bare, Cvi, enter, enter1, morpho_gen, reset_nominal_databases, nomina
*)
```

*** Error handling ***

```
exception Report of string
;
value report revstem gen =
  let stem = Canon.rdecode revstem
  and gender_str = match gen with
      [ Mas → "M" | Neu → "N" | Fem → "F" | Deictic _ → "*" ] in
  let message = stem ^ " missing gender " ^ gender_str in
  raise (Report message)
;
```

```
value warn revstem str =
  let stem = Canon.decode (mirror revstem) in
  let message = stem ^ " is declined as " ^ str in
  raise (Report message)
;
value print_report s =
  output_string stderr (s ^ "\n")
;
```

Word encodings of strings

```
value code = Encode.code_string (* normalized *)
and revcode = Encode.rev_code_string (* reversed (mirror o code) *)
and revstem = Encode.rev_stem (* stripped of homo counter *)
and normal_stem = Encode.normal_stem
;
(* Declension generators *)
type declension_class =
  [ Gender of gender (* declined substantive, adjective, number, pronoun *)
  | Ind of ind_kind (* indeclinable form *)
  ]
and nmorph = (string × declension_class)
;
(* Affix a suffix string to a stem word using internal sandhi *)
(* fix : Word.word → string → Word.word *)
value fix rstem suff =
  Int_sandhi.int_sandhi rstem (code suff)
;
(* raw affixing for build_han Whitney§195a *)
value fixno rstem suff = List2.unstack rstem (code suff)
;
value wrap rstem c = mirror [ c :: rstem ]
;
(* monosyllabic stems, for feminine in ii or uu *)
(* NB - condition not preserved by prefixing and compounding. See Whitney§352 for differing
opinions of grammarians *)
value monosyl = Phonetics.all_consonants (* Z NOT Phonetics monosyllabic *)
;
(* An attempt at treating a few compounds of monosyllabic in -ii *)
(* This question is not clear at all, cf. mail by Malhar Kulkarni *)
(* eg loc sg fem abhii = abhiyi (Zukla) or abhyaam (Malhar) ? *)
```

```
(* Malhar actually says: 3 forms abhyi according to commentators *)
(* if consonant clutter before ii or uu, then not nadii P{1.4.4} *)
(* This is dubious, see -vii -nii below *)
(* See Kale §76 §77 *)
value compound_monosyl_ii = fun
  [ [ 40 :: l ] (* -bhii *) → match l with
       [ [ 1 ] | [ 1; 37; 1 ] → True (* abhii apabhii *)
       | _ → False
       ]
  | [ 35 :: l ] (* -dhii *) → match l with
       [ [ 1; 37; 44; 1 ] | [ 2; 33; 32; 3 ] | [ 5; 17 ] | [ 43; 5; 34 ]
       | [ 5; 48 ] → True (* alpa- itthaa- ku- dur- su- *)
       | _ → False
       ]
  | [ 43 :: [ 37 :: l ] ] (* -prii *) → match l with
       [ [ 2 ] (* aaprii *) → True
       | _ → False
       ]
  | [ 43 :: [ 46 :: _ ] ] (* -zrii *) → True (* ma njuzrii *)
(*— 31 :: l (* -.nii for -nii *) -¿ match l with [ 1; 41; 2; 43; 19 ] (× graama − × ) →
 True (× wrong − \Pan{6,4,82} × ) | _ → False *)
(*— 36 :: l (* -nii *) -¿ match l with [ 2; 36; 10; 48 ] (× senaa− ×) → True (× wrong Deshpande gr p14
) | _ → False *)
(*— 45 :: l (* -vii *) -¿ match l with (* wrong: padaviim *) [ 1; 34; 1; 37 ] →
 True (× pada − × ) | _ → False *)
  | _ → False (* to be completed for other roots *)
  ]
;
(* Similarly for -uu roots *)
value compound_monosyl_uu = fun
  [ [ 40 :: _ ] (* -bhuu *) → True (* abhiibhuu (may be too wide) *)
  | [ 48 :: _ ] (* -suu *) → True (* prasuu (may be too wide) *)
  | [ 43 :: [ 40 :: _ ] ] (* -bhruu *) → True (* subhruu (may be too wide) *)
  | _ → False (* eg m. khalapuu to be completed for other stems *)
  ]
;
```

Stems with possible pronominal declension

```
value pronominal_usage = fun
  [ "prathama" | "dvitaya" | "t.rtiiya" | "apara"
```

```
  | "alpa" | "ardha" | "kevala" | "baahya" → True (* Whitney§526 *)
  | _ → False
  ]
;
```

(* The following restrict the generative capacity of certain entries, in order to reduce over-generation. Such information should ultimately be lexicalized *)

Masculine a-entries may be all used as iiv (inchoative cvi suffix)
NB pronouns "eka" and "sva" produces cvi form in *build_pron_a*
idem for masculines in -i and -in
Now for neuter stems (ad-hoc - should be more general)

```
value a_n_iiv = fun
  [ "aaspada" | "kara.na" | "go.spada" | "t.r.na" | "nimitta" | "paatra"
  | "pi~njara" | "pratibimba" | "pratyak.sa" | "pramaa.na" | "prahara.na"
  | "yuddha" | "vahana" | "vize.sa.na" | "vi.sa" | "vyajana" | "zayana"
  | "zo.na" | "sukha"
  | (* NavyaNyaaya *) "adhikara.na" | "kaara.na" | "saadhana"
    (* missing compound: "si.mhavyaaghraami.sa" *)
      → True
  | _ → False
  ]
and man_iiv = fun (* sn *)
  [ "karman" | "bhasman"
      → True
  | _ → False
  ]
and as_iiv = fun (* sn *)
  [ "unmanas" | "uras" | "cetas" | "manas" | "rajas" | "rahas"
      → True
  | _ → False
  ]
and aa_iiv = fun
  [ "kathaa" → True
  | _ → False
  ]
(* NB aa_iic obsolete, now use separate entry femcf marked fstem *)
;
(*********************************)
(************ Paradigms ***********)
(*********************************)
```

For use in mono-entries paradigms

```
value register case form = (case, code form)
;
value build_mas_a stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry (
   [ Declined Noun Mas
   [ (Singular,if entry = "ubha" (* dual only *)
                 ∨ entry = "g.rha" (* plural only *)
                 ∨ entry = "daara" then [] else
         [ decline Voc "a"
         ; decline Nom "as"
         ; decline Acc "am"
         ; decline Ins "ena"
         ; decline Dat "aaya"
         ; decline Abl "aat"
         ; decline Gen "asya"
         ; decline Loc "e"
         ])
   ; (Dual, if entry = "g.rha"
              ∨ entry = "daara" then [] else
         [ decline Voc "au"
         ; decline Nom "au"
         ; decline Acc "au"
         ; decline Ins "aabhyaam"
         ; decline Dat "aabhyaam"
         ; decline Abl "aabhyaam"
         ; decline Gen "ayos"
         ; decline Loc "ayos"
         ])
   ; (Plural, if entry = "ubha" then [] else
        let l =
         [ decline Voc "aas"
         ; decline Nom "aas"
         ; decline Acc "aan"
         ; decline Ins "ais"
         ; decline Dat "ebhyas"
         ; decline Abl "ebhyas"
         ; decline Gen "aanaam"
         ; decline Loc "esu"
```

```
        ] in
      if pronominal_usage entry then [ decline Nom "e" :: l ] else l)
  ]
  ; Bare Noun (wrap stem 1)
  ; Avyayaf (fix stem "am"); Avyayaf (fix stem "aat") (* avyayiibhaava *)
  ; Indecl Tas (fix stem "atas") (* tasil productive *)
  ; Cvi (wrap stem 4) (* cvi productive *)
  ])
;
value build_mas_i stem trunc entry = (* declension of "ghi" class *)
 let declines case suff = (case, fix stem suff)
 and declineg case suff = (case, fix [ 10 :: trunc ] suff)
 and declinel case suff = (case, fix [ 4 :: trunc ] suff)
 and declinau case = (case, wrap trunc 13) in
 enter entry (
  [ Declined Noun Mas
  [ (Singular,
        [ declineg Voc ""
        ; declines Nom "s"
        ; declines Acc "m"
        ; declines Ins "naa"
        ; declineg Dat "e"
        ; declineg Abl "s"
        ; declineg Gen "s"
        ; declinau Loc
        ])
  ; (Dual,
        [ declinel Voc ""
        ; declinel Nom ""
        ; declinel Acc ""
        ; declines Ins "bhyaam"
        ; declines Dat "bhyaam"
        ; declines Abl "bhyaam"
        ; declines Gen "os"
        ; declines Loc "os"
        ])
  ; (Plural,
        [ declineg Voc "as"
        ; declineg Nom "as"
        ; declinel Acc "n"
```

```
          ; declines Ins "bhis"
          ; declines Dat "bhyas"
          ; declines Abl "bhyas"
          ; declinel Gen "naam"
          ; declines Loc "su"
          ])
    ]
    ; Bare Noun (mirror stem)
    ; Avyayaf (mirror stem)
    ; Avyayaf (mirror stem)
    ; Indecl Tas (fix stem "tas")
    ; Cvi (wrap trunc 4) (* "aadhi1" "pratinidhi" *)
    ])
;
value build_sakhi stem entry sakhi = (* Whitney§343a *)
  let decline case suff = (case, fix stem suff) in
  enter entry (
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "e"
          ; decline Nom "aa"
          ; decline Acc "aayam"
          ; decline Ins "yaa"
          ; decline Dat "ye"
          ; decline Abl "yus"
          ; decline Gen "yus"
          ; decline Loc "yau"
          ])
    ; (Dual,
          [ decline Voc "aayau"
          ; decline Nom "aayaa" (* ved. Whitney§343b *)
          ; decline Nom "aayau"
          ; decline Acc "aayau"
          ; decline Ins "ibhyaam"
          ; decline Dat "ibhyaam"
          ; decline Abl "ibhyaam"
          ; decline Gen "yos"
          ; decline Loc "yos"
          ])
    ; (Plural,
```

```
        [ decline Voc "aayas"
        ; decline Nom "aayas"
        ; decline Acc "iin"
        ; decline Ins "ibhis"
        ; decline Dat "ibhyas"
        ; decline Abl "ibhyas"
        ; decline Gen "iinaam"
        ; decline Loc "isu"
        ])
  ]
  ; Avyayaf (wrap stem 3)
(* ; Cvi (wrap stem 4) *)
  ] @ (if sakhi then [ Bare Noun (wrap stem 1) ] (* sakha *) else []))
;
value build_mas_u stem trunc entry = (* similar to build_mas_i *)
 let declines case suff = (case, fix stem suff)
 and declineg case suff = (case, fix [ 12 :: trunc ] suff)
 and declinel case suff = (case, fix [ 6 :: trunc ] suff)
 and declinau case = (case, wrap trunc 13) in
 enter entry
  [ Declined Noun Mas
  [ (Singular,
        [ declineg Voc ""
        ; declines Nom "s"
        ; declines Acc "m"
        ; declines Ins "naa"
        ; declineg Dat "e"
        ; declineg Abl "s"
        ; declineg Gen "s"
        ; declinau Loc
        ])
  ; (Dual,
        [ declinel Voc ""
        ; declinel Nom ""
        ; declinel Acc ""
        ; declines Ins "bhyaam"
        ; declines Dat "bhyaam"
        ; declines Abl "bhyaam"
        ; declines Gen "os"
        ; declines Loc "os"
```

```
          ])
   ; (Plural,
          [ declineg Voc "as"
          ; declineg Nom "as"
          ; declinel Acc "n"
          ; declines Ins "bhis"
          ; declines Dat "bhyas"
          ; declines Abl "bhyas"
          ; declinel Gen "naam"
          ; declines Loc "su"
          ])
   ]
   ; Bare Noun (mirror stem)
   ; Cvi (wrap trunc 6) (* .rju maru m.rdu laghu *)
   ; Avyayaf (mirror stem)
   ; Indecl Tas (fix stem "tas")
   ]
;
value build_mas_ri_v stem entry = (* vriddhi in strong cases *)
  let decline case suff = (case, fix stem suff)
  and bare = wrap stem 7 in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
          [ decline Voc "ar"
          ; decline Nom "aa"
          ; decline Acc "aaram"
          ; decline Ins "raa"
          ; decline Dat "re"
          ; decline Abl "ur"
          ; decline Gen "ur"
          ; decline Loc "ari"
          ])
   ; (Dual,
          [ decline Voc "aarau"
          ; decline Nom "aarau"
          ; decline Acc "aarau"
          ; decline Ins ".rbhyaam"
          ; decline Dat ".rbhyaam"
          ; decline Abl ".rbhyaam"
```

```
          ; decline Gen "ros"
          ; decline Loc "ros"
          ])
    ; (Plural,
          [ decline Voc "aaras"
          ; decline Nom "aaras"
          ; decline Acc ".rrn"
          ; decline Ins ".rbhis"
          ; decline Dat ".rbhyas"
          ; decline Abl ".rbhyas"
          ; decline Gen ".rr.naam"
          ; decline Loc ".r.su"
          ])
    ]
    ; Bare Noun bare
    ; Avyayaf bare
    ]
;
(* kro.s.t.r irregular with stem krostu Muller§236 P{7,1,95-97} *)
value build_krostu stem entry =
  let decline case suff = (case, fix stem suff)
  and bare = wrap stem 5 in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "o"
          ; decline Nom "aa"
          ; decline Acc "aaram"
          ; decline Ins "unaa"
          ; decline Ins "raa"
          ; decline Dat "ave"
          ; decline Dat "re"
          ; decline Abl "or"
          ; decline Abl "ur"
          ; decline Gen "or"
          ; decline Gen "ur"
          ; decline Loc "au"
          ; decline Loc "ari"
          ])
    ; (Dual,
```

```
        [ decline Voc "aarau"
        ; decline Nom "aarau"
        ; decline Acc "aarau"
        ; decline Ins "ubhyaam"
        ; decline Dat "ubhyaam"
        ; decline Abl "ubhyaam"
        ; decline Gen "vos"
        ; decline Gen "ros"
        ; decline Loc "vos"
        ; decline Loc "ros"
        ])
    ; (Plural,
        [ decline Voc "aaras"
        ; decline Nom "aaras"
        ; decline Acc "uun"
        ; decline Ins "ubhis"
        ; decline Dat "ubhyas"
        ; decline Abl "ubhyas"
        ; decline Gen "uunaam"
        ; decline Loc "u.su"
        ])
    ]
    ; Bare Noun bare
    ; Avyayaf bare
    ]
;
value build_mas_ri_g stem entry = (* parenté avec gu.na *)
  let decline case suff = (case, fix stem suff)
  and bare = wrap stem 7 in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
        [ decline Voc "ar"
        ; decline Nom "aa"
        ; decline Acc "aram"
        ; decline Ins "raa"
        ; decline Dat "re"
        ; decline Abl "ur"
        ; decline Gen "ur"
        ; decline Loc "ari"
```

```
          ])
    ; (Dual,
          [ decline Voc "arau"
          ; decline Nom "arau"
          ; decline Acc "arau"
          ; decline Ins ".rbhyaam"
          ; decline Dat ".rbhyaam"
          ; decline Abl ".rbhyaam"
          ; decline Gen "ros"
          ; decline Loc "ros"
          ])
    ; (Plural,
          [ decline Voc "aras"
          ; decline Nom "aras"
          ; decline Acc ".rrn"
          ; decline Acc "aras" (* epics Whitney§373c *)
          ; decline Ins ".rbhis"
          ; decline Dat ".rbhyas"
          ; decline Abl ".rbhyas"
          ; decline Gen ".rr.naam"
          ; decline Loc ".r.su"
          ])
    ]
    ; Bare Noun bare
    ; Bare Noun (wrap stem 2) (* for dvandva eg ved hotaapotarau P{6,3,47} *)
    ; Avyayaf bare
    ; Indecl Tas (fix stem ".rtas") (* pit.rtas *)
    ]
;
value build_nri stem entry = (* currently disabled by skip in Dico *)
  let decline case suff = (case, fix stem suff)
  and bare = wrap stem 7 in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Nom "aa" (* other cases from nara like naram *)
          ; decline Loc "ari" (* MacDonell §101b *)
          ])
    ; (Dual,
          [ decline Voc "arau"
```

```
        ; decline Nom "arau"
        ; decline Acc "arau"
        ; decline Ins ".rbhyaam"
        ; decline Dat ".rbhyaam"
        ; decline Abl ".rbhyaam"
        ; decline Gen "ros"
        ; decline Loc "ros"
        ])
   ; (Plural,
        [ decline Voc "aras"
        ; decline Nom "aras"
        ; decline Acc ".rrn"
        ; decline Ins ".rbhis"
        ; decline Dat ".rbhyas"
        ; decline Abl ".rbhyas"
        ; decline Gen ".rr.naam"
        ; decline Gen ".r.naam" (* Veda, but .r metrically long *)
        ; decline Loc ".r.su"
        ])
   ]
   ; Bare Noun bare
   ]
;
value build_mas_red stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
        [ decline Voc "t"
        ; decline Nom "t"
        ; decline Acc "tam"
        ; decline Ins "taa"
        ; decline Dat "te"
        ; decline Gen "tas"
        ; decline Loc "ti"
        ])
   ; (Dual,
        [ decline Voc "tau"
        ; decline Nom "tau"
        ; decline Acc "tau"
```

```
          ; decline Ins "dbhyaam"
          ; decline Dat "dbhyaam"
          ; decline Abl "dbhyaam"
          ; decline Gen "tos"
          ; decline Loc "tos"
          ])
    ; (Plural,
          [ decline Voc "tas"
          ; decline Nom "tas"
          ; decline Acc "tas"
          ; decline Ins "dbhis"
          ; decline Dat "dbhyas"
          ; decline Abl "dbhyas"
          ; decline Gen "taam"
          ; decline Loc "tsu"
          ])
    ]
    ; Indecl Tas (fix stem "tas")
    ]
;
value build_mas_at stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "n"
          ; decline Nom "n"
          ; decline Acc "ntam"
          ; decline Ins "taa"
          ; decline Dat "te"
          ; decline Abl "tas"
          ; decline Gen "tas"
          ; decline Loc "ti"
          ])
    ; (Dual,
          [ decline Voc "ntau"
          ; decline Nom "ntau"
          ; decline Acc "ntau"
          ; decline Ins "dbhyaam"
          ; decline Dat "dbhyaam"
```

```
          ; decline Abl "dbhyaam"
          ; decline Gen "tos"
          ; decline Loc "tos"
          ])
    ; (Plural,
          [ decline Voc "ntas"
          ; decline Nom "ntas"
          ; decline Acc "tas"
          ; decline Ins "dbhis"
          ; decline Dat "dbhyas"
          ; decline Abl "dbhyas"
          ; decline Gen "taam"
          ; decline Loc "tsu"
          ])
    ]
    ; Bare Noun (wrap stem 32) (* at - e.g. b.rhadazva *)
    ; Avyayaf (fix stem "ntam") (* tam ? *)
    ]
;
value build_mas_mat stem entry = (* poss adj mas in -mat or -vat *)
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "an"
          ; decline Nom "aan"
          ; decline Acc "antam"
          ; decline Ins "ataa"
          ; decline Dat "ate"
          ; decline Abl "atas"
          ; decline Gen "atas"
          ; decline Loc "ati"
          ])
    ; (Dual,
          [ decline Voc "antau"
          ; decline Nom "antau"
          ; decline Acc "antau"
          ; decline Ins "adbhyaam"
          ; decline Dat "adbhyaam"
          ; decline Abl "adbhyaam"
```

```
        ; decline Gen "atos"
        ; decline Loc "atos"
        ])
    ; (Plural,
        [ decline Voc "antas"
        ; decline Nom "antas"
        ; decline Acc "atas"
        ; decline Ins "adbhis"
        ; decline Dat "adbhyas"
        ; decline Abl "adbhyas"
        ; decline Gen "ataam"
        ; decline Loc "atsu"
        ])
    ]
    ; Bare Noun (mirror [ 32 :: [ 1 :: stem ] ]) (* mat - e.g. zriimat *)
    ; Avyayaf (fix stem "antam") (* atam ? *)
    ]
;
value build_mas_mahat stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
        [ decline Voc "aan"
        ; decline Nom "aan"
        ; decline Acc "aantam"
        ; decline Ins "ataa"
        ; decline Dat "ate"
        ; decline Abl "atas"
        ; decline Gen "atas"
        ; decline Loc "ati"
        ])
    ; (Dual,
        [ decline Voc "aantau"
        ; decline Nom "aantau"
        ; decline Acc "aantau"
        ; decline Ins "adbhyaam"
        ; decline Dat "adbhyaam"
        ; decline Abl "adbhyaam"
        ; decline Gen "atos"
```

```
          ; decline Loc "atos"
          ])
    ; (Plural,
          [ decline Voc "aantas"
          ; decline Nom "aantas"
          ; decline Acc "atas"
          ; decline Ins "adbhis"
          ; decline Dat "adbhyas"
          ; decline Abl "adbhyas"
          ; decline Gen "ataam"
          ; decline Loc "atsu"
          ])
    ]
    ; Bare Noun (wrap stem 2) (* mahaa- *)
    ; Cvi (wrap stem 4)
    ; Avyayaf (fix stem "aantam") (* atam ? *)
    ]
;
(* stems having a consonant before man or van have vocalic endings an *)
value avocalic = fun
  [ [ last :: _ ] → ¬ (Phonetics.vowel last)
  | [] → failwith "Nouns.avocalic:␣empty␣stem"
  ]
;
(* NB impossible to factorise with build_van because "mne" and not "nne" *)
value build_man g stem entry =
  let vedic_drop = match entry with (* Whitney§425e *)
      [ "mahiman" | "prathiman" | "variman" | "daaman" | "preman" | "bhuuman"
          → True
      | _ → False
      ]
  and avoc = avocalic stem in
  let decline case suff = (case, fix stem suff) in
  enter entry (
    [ Declined Noun g
    [ (Singular,
          [ decline Voc "man"
          ; decline Nom (if g = Neu then "ma" else "maa")
          ; decline Acc (if g = Neu then "ma" else "maanam")
          ; decline Ins (if avoc then "manaa" else "mnaa")
```

```
        ; decline Dat (if avoc then "mane" else "mne")
        ; decline Abl (if avoc then "manas" else "mnas")
        ; decline Gen (if avoc then "manas" else "mnas")
        ; decline Loc "mani"
        ] @ (if g = Neu then [ decline Voc "ma" ] else [])
          @ (if vedic_drop then [ decline Ins "naa" ] else [])
          @ (if avoc then [] else [ decline Loc "mni" ]))
   ; (Dual, (if g = Neu then
        [ decline Voc "manii"
        ; decline Voc "mnii"
        ; decline Nom "manii"
        ; decline Nom "mnii"
        ; decline Acc "manii"
        ; decline Acc "mnii"
        ]
             else
        [ decline Voc "maanau"
        ; decline Nom "maanau"
        ; decline Acc "maanau"
        ]) @
        [ decline Ins "mabhyaam"
        ; decline Dat "mabhyaam"
        ; decline Abl "mabhyaam"
        ; decline Gen (if avoc then "manos" else "mnos")
        ; decline Loc (if avoc then "manos" else "mnos")
        ])
   ; (Plural, if g = Neu then
        [ decline Voc "maani"
        ; decline Nom "maani"
        ; decline Acc "maani"
        ] else
        [ decline Voc "maanas"
        ; decline Nom "maanas"
        ; decline Acc (if avoc then "manas" else "mnas")
        ])
   ; (Plural,
        [ decline Ins "mabhis"
        ; decline Dat "mabhyas"
        ; decline Abl "mabhyas"
        ; decline Gen (if avoc then "manaam" else "mnaam")
```

```
          ; decline Loc "masu"
          ])
    ]
    ; Avyayaf (fix stem "mam")
    ; Indecl Tas (fix stem "matas")
    ] @ (if entry = "dharman" then [ ] (* redundant with dharma *)
          else [ Bare Noun (mirror [ 1 :: [ 41 :: stem ]]) ])
      @ (if g = Neu ∧ man_iiv entry then [ Cvi (mirror [ 4 :: [ 41 :: stem ]]) ]
          else [ ])
      @ if g = Neu then [ Avyayaf (fix stem "ma") ] else [ ]) (* P{5,4,109} *)
;
value build_man_god stem entry = (* Aryaman Whitney §426a; Kale §118 *)
  let decline case suff = (case, fix stem suff) in
  enter entry (
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "man"
          ; decline Nom "maa"
          ; decline Acc "manam"
          ; decline Ins "mnaa" (* aryam.naa and not *arya.n.naa *)
          ; decline Dat "mne" (* above forbids merging with build_an_god *)
          ; decline Abl "mnas"
          ; decline Gen "mnas"
          ; decline Loc "mani"
          ; decline Loc "mni"
          ])
    ; (Dual,
          [ decline Voc "manau"
          ; decline Nom "manau"
          ; decline Acc "manau"
          ; decline Ins "mabhyaam"
          ; decline Dat "mabhyaam"
          ; decline Abl "mabhyaam"
          ; decline Gen "mnos"
          ; decline Loc "mnos"
          ])
    ; (Plural,
          [ decline Voc "manas"
          ; decline Nom "manas"
          ; decline Acc "mnas"
```

```
          ; decline Ins "mabhis"
          ; decline Dat "mabhyas"
          ; decline Abl "mabhyas"
          ; decline Gen "mnaam"
          ; decline Loc "masu"
          ])
    ]
    ; Bare Noun (mirror [ 1 :: [ 41 :: stem ]])
    ])
;
value build_van g stem entry =
  let avoc = avocalic stem in
  let decline case suff = (case, fix stem suff) in
  enter entry (
    [ Declined Noun g
    [ (Singular,
          [ decline Voc "van"
          ; decline Nom (if entry = "piivan" then "vaan" (* Gonda *)
                         else if g = Neu then "va" else "vaa")
          ; decline Acc (if g = Neu then "va" else "vaanam")
          ; decline Ins (if avoc then "vanaa" else "vnaa")
          ; decline Dat (if avoc then "vane" else "vne")
          ; decline Abl (if avoc then "vanas" else "vnas")
          ; decline Gen (if avoc then "vanas" else "vnas")
          ; decline Loc "vani"
          ] @ (if g = Neu then [ decline Voc "va" ] else [])
            @ (if avoc then [] else [ decline Loc "vni" ]))
    ; (Dual, (if g = Neu then
          [ decline Voc "vanii"
          ; decline Voc "vnii" (* if avoc ? *)
          ; decline Nom "vanii"
          ; decline Nom "vnii" (* if avoc ? *)
          ; decline Acc "vanii"
          ; decline Acc "vnii" (* if avoc ? *)
          ]
              else
          [ decline Voc "vaanau"
          ; decline Nom "vaanau"
          ; decline Acc "vaanau"
          ]) @
```

```
          [ decline Ins "vabhyaam"
          ; decline Dat "vabhyaam"
          ; decline Abl "vabhyaam"
          ; decline Gen (if avoc then "vanos" else "vnos")
          ; decline Loc (if avoc then "vanos" else "vnos")
          ])
    ; (Plural, if g = Neu then
          [ decline Voc "vaani"
          ; decline Nom "vaani"
          ; decline Acc "vaani"
          ] else
          [ decline Voc "vaanas"
          ; decline Nom "vaanas"
          ; decline Acc (if avoc then "vanas" else "vnas")
          ])
    ; (Plural,
          [ decline Ins "vabhis"
          ; decline Dat "vabhyas"
          ; decline Abl "vabhyas"
          ; decline Gen (if avoc then "vanaam" else "vnaam")
          ; decline Loc "vasu"
          ])
    ]
    ; Bare Noun (mirror [ 1 :: [ 45 :: stem ]])
    ; Avyayaf (fix stem "vam")
    ; Indecl Tas (fix stem "vatas")
    ]
    @ if g = Neu then [ Avyayaf (fix stem "va") ] else [ ]) (* P{5,4,109} *)
;
value build_an g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry (
    [ Declined Noun g
    [ (Singular,
          [ decline Voc "an"
          ; decline Nom (if g = Neu then "a" else "aa")
          ; decline Acc (if g = Neu then "a" else "aanam")
          ; decline Ins "naa"
          ; decline Dat "ne"
          ; decline Abl "nas"
```

```
        ; decline Gen "nas"
        ; decline Loc "ani"
        ; decline Loc "ni"
        ] @ (if g = Neu then
        [ decline Voc "a" ] else []))
   ; (Dual, (if g = Neu then
        [ decline Voc "anii"
        ; decline Voc "nii"
        ; decline Nom "anii"
        ; decline Nom "nii"
        ; decline Acc "anii"
        ; decline Acc "nii"
        ]
             else
        [ decline Voc "aanau"
        ; decline Nom "aanau"
        ; decline Acc "aanau"
        ]) @
        [ decline Ins "abhyaam"
        ; decline Dat "abhyaam"
        ; decline Abl "abhyaam"
        ; decline Gen "nos"
        ; decline Loc "nos"
        ])
   ; (Plural, if g = Neu then
        [ decline Voc "aani"
        ; decline Nom "aani"
        ; decline Acc "aani"
        ] else
        [ decline Voc "aanas"
        ; decline Nom "aanas"
        ; decline Acc "nas"
        ])
   ; (Plural,
        [ decline Ins "abhis"
        ; decline Dat "abhyas"
        ; decline Abl "abhyas"
        ; decline Gen "naam"
        ; decline Loc "asu"
        ])
```

```
    ]
    ; Bare Noun (wrap stem 1)
    ; Avyayaf (fix stem "am")
    ; Indecl Tas (fix stem "atas")
    ] @ if g = Neu then [ Avyayaf (fix stem "a") ] else []) (* P{5,4,109} *)
;
value build_an_god stem entry = (* Whitney §426a *)
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "an"
          ; decline Nom "aa"
          ; decline Acc "anam"
          ; decline Ins "naa"
          ; decline Dat "ne"
          ; decline Abl "nas"
          ; decline Gen "nas"
          ; decline Loc "ani"
          ; decline Loc "ni"
          ])
    ; (Dual,
          [ decline Voc "anau"
          ; decline Nom "anau"
          ; decline Acc "anau"
          ; decline Ins "abhyaam"
          ; decline Dat "abhyaam"
          ; decline Abl "abhyaam"
          ; decline Gen "nos"
          ; decline Loc "nos"
          ])
    ; (Plural,
          [ decline Voc "anas"
          ; decline Nom "anas"
          ; decline Acc "nas"
          ; decline Ins "abhis"
          ; decline Dat "abhyas"
          ; decline Abl "abhyas"
          ; decline Gen "naam"
          ; decline Loc "asu"
```

```
          ])
    ]
    ; Bare Noun (wrap stem 1)
    ]
;
value build_sp_an stem entry =
(* Whitney§432 these stems substitute the following for Voc Nom Acc : "yakan" →"yak.rt"
"zakan" →"zak.rt" "udan" →"udaka" "yuu.san" →"yuu.sa" "do.san" →"dos" "asan"
→"as.rj" "aasan" →"aasya" Kale§129 Renou§241d *)
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ decline Ins "naa"
          ; decline Dat "ne"
          ; decline Abl "nas"
          ; decline Gen "nas"
          ; decline Loc "ani"
          ])
    ; (Dual,
          [ decline Ins "abhyaam"
          ; decline Dat "abhyaam"
          ; decline Abl "abhyaam"
          ; decline Gen "nos"
          ; decline Loc "nos"
          ])
    ; (Plural,
          [ decline Acc "aani" (* Kale§129 zakan but not yakan, Renou: trouble *)
          ; decline Ins "abhis"
          ; decline Dat "abhyas"
          ; decline Abl "abhyas"
          ; decline Gen "naam"
          ; decline Loc "asu"
          ])
    ]
    ; Bare Noun (wrap stem 1)
(* ; Avyayaf ? *)
    ]
;
value build_han stem entry = (* stem = ...-han Whitney§402 *)
```

```
(* g=Mas only, since g=Neu is dubious specially -ha *)
let decline case suff = (case, fix stem suff)
and declino case suff = (case, fixno stem suff) in (* no retroflexion of n *)
enter entry
 [ Declined Noun Mas
 [ (Singular,
     [ decline Voc "han"
     ; decline Nom "haa" (* if g=Neu then "ha" else "haa" *)
     ; decline Acc "hanam" (* if g=Neu then "ha" else "hanam" *)
     ; declino Ins "ghnaa" (* v.rtraghnaa, not *v.rtragh.naa Whitney§195a *)
     ; declino Dat "ghne"
     ; declino Abl "ghnas"
     ; declino Gen "ghnas"
     ; declino Loc "ghni"
     ; decline Loc "hani"
     ]) (* @ (if g=Neu then decline Voc "ha" else )) *)
  ; (Dual, (* if g=Neu then decline Voc "hanii" ; declino Voc "ghnii" ; decline Nom "hanii"
; declino Nom "ghnii" ; decline Acc "hanii" ; declino Acc "ghnii" else *)
     [ decline Voc "hanau"
     ; decline Nom "hanau"
     ; decline Acc "hanau"
     ; decline Ins "habhyaam"
     ; decline Dat "habhyaam"
     ; decline Abl "habhyaam"
     ; declino Gen "ghnos"
     ; declino Loc "ghnos"
     ])
  ; (Plural, (* if g=Neu then decline Voc "haani" ; decline Nom "haani" ; decline Acc "haani"
else *)
     [ decline Voc "hanas"
     ; decline Nom "hanas"
     ; declino Acc "ghnas"
     ; decline Ins "habhis"
     ; decline Dat "habhyas"
     ; decline Abl "habhyas"
     ; declino Gen "ghnaam"
     ; decline Loc "hasu"
     ])
  ]
  ; Avyayaf (fix stem "hanam")
```

```
    ]
;
value build_mas_zvan stem entry = (* P{6,4,133} *)
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
         [ decline Voc "van"
         ; decline Nom "vaa"
         ; decline Acc "vaanam"
         ; decline Ins "unaa"
         ; decline Dat "une"
         ; decline Abl "unas"
         ; decline Gen "unas"
         ; decline Loc "uni"
         ])
   ; (Dual,
         [ decline Voc "vaanau"
         ; decline Nom "vaanau"
         ; decline Acc "vaanau"
         ; decline Ins "vabhyaam"
         ; decline Dat "vabhyaam"
         ; decline Abl "vabhyaam"
         ; decline Gen "unos"
         ; decline Loc "unos"
         ])
   ; (Plural,
         [ decline Voc "vaanas"
         ; decline Nom "vaanas"
         ; decline Acc "unas"
         ; decline Ins "vabhis"
         ; decline Dat "vabhyas"
         ; decline Abl "vabhyas"
         ; decline Gen "unaam"
         ; decline Loc "vasu"
         ])
   ]
   (* Bare Noun (code "zunas") abl/gen pour zuna.hzepa non génératif *)
   (* Bare Noun (code "zvaa") zvaapada avec nom. non génératif *)
   ; Bare Noun (mirror [ 1 :: [ 45 :: stem ] ]) (* eg zva-v.rtti *)
```

```
    ; Avyayaf (fix stem "vaanam") (* "vam" ? *)
    ]
;
value build_athin stem entry = (* pathin, supathin, mathin *)
  let decline case suff = (case, fix stem suff)
  and bare = fix stem "thi" in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "nthaas"
          ; decline Nom "nthaas"
          ; decline Acc "nthaanam"
          ; decline Ins "thaa"
          ; decline Dat "the"
          ; decline Abl "thas"
          ; decline Gen "thas"
          ; decline Loc "thi"
          ])
    ; (Dual,
          [ decline Voc "nthaanau"
          ; decline Nom "nthaanau"
          ; decline Acc "nthaanau"
          ; decline Ins "thibhyaam"
          ; decline Dat "thibhyaam"
          ; decline Abl "thibhyaam"
          ; decline Gen "thos"
          ; decline Loc "thos"
          ])
    ; (Plural,
          [ decline Voc "nthaanas"
          ; decline Nom "nthaanas"
          ; decline Acc "thas"
          ; decline Ins "thibhis"
          ; decline Dat "thibhyas"
          ; decline Abl "thibhyas"
          ; decline Gen "thaam"
          ; decline Loc "thisu"
          ])
    ]
    ; Bare Noun bare
```

```
    ; Avyayaf bare
    ]
;
value build_ribhuksin stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "aas"
          ; decline Nom "aas"
          ; decline Acc "aanam"
          ; decline Acc "anam" (* P{6,4,9} *)
          ; decline Ins "aa"
          ; decline Dat "e"
          ; decline Abl "as"
          ; decline Gen "as"
          ; decline Loc "i"
          ])
    ; (Dual,
          [ decline Voc "aanau"
          ; decline Nom "aanau"
          ; decline Acc "aanau"
          ; decline Ins "ibhyaam"
          ; decline Dat "ibhyaam"
          ; decline Abl "ibhyaam"
          ; decline Gen "os"
          ; decline Loc "os"
          ])
    ; (Plural,
          [ decline Voc "aanas"
          ; decline Nom "aanas"
          ; decline Acc "as"
          ; decline Ins "ibhis"
          ; decline Dat "ibhyas"
          ; decline Abl "ibhyas"
          ; decline Gen "aam"
          ; decline Loc "asu"
          ])
    ]
(* ; Avyayaf ? *)
```

```
  ]
;
value build_mas_yuvan entry = (* P{6,4,133} *)
  let stem = [ 42 ] (* y *) in
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
        [ decline Voc "uvan"
        ; decline Nom "uvaa"
        ; decline Acc "uvaanam"
        ; decline Ins "uunaa"
        ; decline Dat "uune"
        ; decline Abl "uunas"
        ; decline Gen "uunas"
        ; decline Loc "uuni"
        ])
   ; (Dual,
        [ decline Voc "uvaanau"
        ; decline Nom "uvaanau"
        ; decline Acc "uvaanau"
        ; decline Ins "uvabhyaam"
        ; decline Dat "uvabhyaam"
        ; decline Abl "uvabhyaam"
        ; decline Gen "uunos"
        ; decline Loc "uunos"
        ])
   ; (Plural,
        [ decline Voc "uvaanas"
        ; decline Nom "uvaanas"
        ; decline Acc "uunas"
        ; decline Ins "uvabhis"
        ; decline Dat "uvabhyas"
        ; decline Abl "uvabhyas"
        ; decline Gen "uunaam"
        ; decline Loc "uvasu"
        ])
   ]
   ; Bare Noun (code "yuva")
   ; Avyayaf (code "yuvam") (* ? *)
```

```
  ]
;
value build_mas_maghavan entry = (* P{6,4,133} *)
  let stem = revcode "magh" in
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
         [ decline Voc "avan"
         ; decline Nom "avaa"
         ; decline Acc "avaanam"
         ; decline Ins "onaa"
         ; decline Dat "one"
         ; decline Abl "onas"
         ; decline Gen "onas"
         ; decline Loc "oni"
         ])
   ; (Dual,
         [ decline Voc "avaanau"
         ; decline Nom "avaanau"
         ; decline Acc "avaanau"
         ; decline Ins "avabhyaam"
         ; decline Dat "avabhyaam"
         ; decline Abl "avabhyaam"
         ; decline Gen "onos"
         ; decline Loc "onos"
         ])
   ; (Plural,
         [ decline Voc "avaanas"
         ; decline Nom "avaanas"
         ; decline Acc "onas"
         ; decline Ins "avabhis"
         ; decline Dat "avabhyas"
         ; decline Abl "avabhyas"
         ; decline Gen "onaam"
         ; decline Loc "avasu"
         ])
   ]
   ; Avyayaf (fix stem "avam") (* ? *)
];
```

```
value build_mas_in stem entry =
  let decline case suff = (case, fix stem suff)
  and bare = wrap stem 3 in
  enter entry (
   [ Declined Noun Mas
   [ (Singular,
         [ decline Voc "in"
         ; decline Nom "ii"
         ; decline Acc "inam"
         ; decline Ins "inaa"
         ; decline Dat "ine"
         ; decline Abl "inas"
         ; decline Gen "inas"
         ; decline Loc "ini"
         ])
   ; (Dual,
         [ decline Voc "inau"
         ; decline Nom "inau"
         ; decline Acc "inau"
         ; decline Ins "ibhyaam"
         ; decline Dat "ibhyaam"
         ; decline Abl "ibhyaam"
         ; decline Gen "inos"
         ; decline Loc "inos"
         ])
   ; (Plural,
         [ decline Voc "inas"
         ; decline Nom "inas"
         ; decline Acc "inas"
         ; decline Ins "ibhis"
         ; decline Dat "ibhyas"
         ; decline Abl "ibhyas"
         ; decline Gen "inaam"
         ; decline Loc "i.su"
         ])
   ]
   ; Bare Noun bare
   ; Avyayaf bare
   ; Cvi (wrap stem 4) (* "saak.sin" "sthaayin" overgenerates with vi.saayin *)
   ])
```

```
;
value build_as gen stem entry =
  let decline case suff = (case, fix stem suff)
  and rstem = [ 48 :: [ 1 :: stem ] ] in
  enter entry (
   [ Declined Noun gen
   [ (Singular, let l =
         [ decline Voc "as"
         ; decline Nom (match gen with
             [ Mas → match entry with (* gram Muller p 72, Whitney §416 *)
                       [ "anehas" | "uzanas" | "da.mzas" (* Puruda.mzas *) → "aa"
                       | _ → "aas" (* Kane§108 candramas vedhas su/dur/unmanas *)
                       ]
             | Fem → "aas"
             | Neu → "as" (* manas payas vyas? avas1 zreyas saras vacas *)
             | _ → raise (Control.Anomaly "Nouns")
             ])
         ; decline Acc (match gen with
             [ Mas | Fem → "asam"
             | Neu → "as"
             | _ → raise (Control.Anomaly "Nouns")
             ])
         ; decline Ins "asaa"
         ; decline Dat "ase"
         ; decline Abl "asas"
         ; decline Gen "asas"
         ; decline Loc "asi"
         ] in if entry = "uzanas" ∧ gen = Mas then (* gram Muller p 72 *)
                 [ decline Voc "a"; decline Voc "an" ] @ l
              else l)
   ; (Dual,
         let direct = match gen with
             [ Mas | Fem → "asau"
             | Neu → "asii"
             | _ → raise (Control.Anomaly "Nouns")
             ] in
         [ decline Voc direct
         ; decline Nom direct
         ; decline Acc direct
         ; decline Ins "obhyaam"
```

```
          ; decline Dat "obhyaam"
          ; decline Abl "obhyaam"
          ; decline Gen "asos"
          ; decline Loc "asos"
          ])
    ; (Plural,
        let direct = match gen with
            [ Mas | Fem → "asas"
            | Neu → "aa.msi"
(* eg chandaa.msi: chandas-as Pan7,1,201,1,42 chandas-zi Pan7,1,72 chandans-i Pan6,4,10
chandaans-i Pan8,3,24 chandaa.msi *)
            | _ → raise (Control.Anomaly "Nouns")
            ] in
          [ decline Voc direct
          ; decline Nom direct
          ; decline Acc direct
          ; decline Ins "obhis"
          ; decline Dat "obhyas"
          ; decline Abl "obhyas"
          ; decline Gen "asaam"
          ; decline Loc "a.hsu" (* decline Loc "assu" *) (* Kane§108 opt "astu" *)
          ])
    ]
    ; Bare Noun (mirror rstem) (* as *)
    ; Indecl Tas (fix rstem "tas") (* eg manastas *)
    ]
      @ (match entry with
          ["uras" | "manas" → [ Bare Noun (wrap stem 1) ] (* ura- mana- *)
          | _ → []
          ])
      @ (match entry with
          [ "anas" | "manas" | "cetas" | "jaras" → [ Avyayaf (fix stem "asam") ]
          | _ → []
          ])
      @ (match entry with
          [ "nabhas" → [ Avyayaf (fix stem "as"); Avyayaf (fix stem "yam") ]
          | _ → []
          ])
      @ (if gen = Neu ∧ as_iiv entry then [ Cvi (wrap stem 4) ] else []))
;
```

```
value build_maas () =
  let decline case form = (case, code form) in
  enter "maas"
   [ Declined Noun Mas
   [ (Singular,
          [ decline Nom "maas" (* no Acc Voc ? *)
          ; decline Ins "maasaa"
          ; decline Dat "maase"
          ; decline Abl "maasas"
          ; decline Gen "maasas"
          ; decline Loc "maasi"
          ])
   ; (Dual,
          [ decline Ins "maadbhyaam" (* ou "maabhyaam" ?? *)
          ; decline Ins "maabhyaam" (* Siddhaanta kaumudii - Jha *)
          ; decline Dat "maadbhyaam"
          ; decline Abl "maadbhyaam"
          ; decline Gen "maasos"
          ; decline Loc "maasos"
          ])
   ; (Plural,
          [ decline Ins "maadbhis"
          ; decline Dat "maadbhyas"
          ; decline Abl "maadbhyas"
          ; decline Gen "maasaam"
          ; decline Loc "maa.hsu" (* maassu *)
          ])
   ]
   ]
;
value build_nas entry =
  let decline case form = (case, code form) in
  enter entry
   [ Declined Noun Fem
   [ (Singular,
          [ decline Ins "nasaa"
          ; decline Dat "nase"
          ; decline Abl "nasas"
          ; decline Gen "nasas"
          ; decline Loc "nasi"
```

```
        ])
  ; (Dual,
        [ decline Nom "naasaa" (* RV narines Whitney§397 *)
        ; decline Gen "nasos"
        ; decline Loc "nasos"
        ])
  ]
  ]
;
value build_dos gen entry = (* Kale§108a *)
 let decline case form = (case, code form) in
 enter entry
  [ Declined Noun gen
  [ (Singular,
        [ decline Voc "dos"
        ; decline Nom "dos"
        ; decline Acc "dos"
        ; decline Ins "do.saa"
        ; decline Dat "do.se"
        ; decline Abl "do.sas"
        ; decline Gen "do.sas"
        ; decline Loc "do.si"
        ])
  ; (Dual, let form = match gen with
                          [ Mas | Fem → "do.sau"
                          | Neu → "do.sii"
                          | _ → raise (Control.Anomaly "Nouns")
                          ] in
        [ decline Voc form
        ; decline Nom form
        ; decline Acc form
        ; decline Ins "dorbhyaam"
        ; decline Dat "dorbhyaam"
        ; decline Abl "dorbhyaam"
        ; decline Gen "dor.sos"
        ; decline Loc "dor.sos"
        ])
  ; (Plural, let form = match gen with
            [ Mas | Fem → "do.sas"
            | Neu → "do.m.si"
```

```
            | _ → raise (Control.Anomaly "Nouns")
            ] in
          [ decline Voc form
          ; decline Nom form
          ; decline Acc form
          ; decline Ins "dorbhis"
          ; decline Dat "dorbhyas"
          ; decline Abl "dorbhyas"
          ; decline Gen "do.saam"
          ; decline Loc "do.h.su"
          ])
    ]
    ]
;
value build_is gen stem entry =
  let decline case suff = (case, fix stem suff)
  and rstem = [ 48 :: [ 3 :: stem ] ] in
  let bare = mirror rstem in
  enter entry
    [ Declined Noun gen
    [ (Singular,
          [ decline Voc "is"
          ; decline Nom "is"
          ; decline Acc (match gen with
            [ Mas | Fem → "i.sam"
            | Neu → "is"
            | _ → raise (Control.Anomaly "Nouns")
            ])
          ; decline Ins "i.saa"
          ; decline Dat "i.se"
          ; decline Abl "i.sas"
          ; decline Gen "i.sas"
          ; decline Loc "i.si"
          ])
    ; (Dual,
          let direct = match gen with
              [ Mas | Fem → "i.sau"
              | Neu → "i.sii"
              | _ → raise (Control.Anomaly "Nouns")
              ] in
```

```
          [ decline Voc direct
          ; decline Nom direct
          ; decline Acc direct
          ; decline Ins "irbhyaam"
          ; decline Dat "irbhyaam"
          ; decline Abl "irbhyaam"
          ; decline Gen "i.sos"
          ; decline Loc "i.sos"
          ])
    ; (Plural,
          let direct = match gen with
              [ Mas | Fem → "i.sas"
              | Neu → "ii.msi"
              | _ → raise (Control.Anomaly "Nouns")
              ] in
          [ decline Voc direct
          ; decline Nom direct
          ; decline Acc direct
          ; decline Ins "irbhis"
          ; decline Dat "irbhyas"
          ; decline Abl "irbhyas"
          ; decline Gen "i.saam"
          ; decline Loc "i.h.su" (* decline Loc "i.s.su" *)
          ])
    ]
    ; Bare Noun bare (* is *)
    ; Indecl Tas (fix rstem "tas")
    ; Avyayaf bare
    ]
;
value build_us gen stem entry =
  let decline case suff = (case, fix stem suff)
  and rstem = [ 48 :: [ 5 :: stem ] ] in
  let bare = mirror rstem in
  enter entry
    [ Declined Noun gen
    [ (Singular,
          [ decline Voc "us"
          ; decline Nom "us"
          ; decline Acc (match gen with
```

```
        [ Mas | Fem → "u.sam"
        | Neu → "us"
        | _ → raise (Control.Anomaly "Nouns")
        ])
      ; decline Ins "u.saa"
      ; decline Dat "u.se"
      ; decline Abl "u.sas"
      ; decline Gen "u.sas"
      ; decline Loc "u.si"
      ])
  ; (Dual,
      let direct = match gen with
          [ Mas | Fem → "u.sau"
          | Neu → "u.sii"
          | _ → raise (Control.Anomaly "Nouns")
          ] in
      [ decline Voc direct
      ; decline Nom direct
      ; decline Acc direct
      ; decline Ins "urbhyaam"
      ; decline Dat "urbhyaam"
      ; decline Abl "urbhyaam"
      ; decline Gen "u.sos"
      ; decline Loc "u.sos"
      ])
  ; (Plural,
    let direct = match gen with
        [ Mas | Fem → "u.sas"
        | Neu → "uu.msi"
        | _ → raise (Control.Anomaly "Nouns")
        ] in
      [ decline Voc direct
      ; decline Nom direct
      ; decline Acc direct
      ; decline Ins "urbhis"
      ; decline Dat "urbhyas"
      ; decline Abl "urbhyas"
      ; decline Gen "u.saam"
      ; decline Loc "u.h.su" (* decline Loc "u.s.su" *)
      ])
```

```
    ]
    ; Bare Noun bare (* us *)
    ; Cvi (wrap stem 6) (* arus cak.sus *)
    ; Indecl Tas (fix rstem "tas")
    ; Avyayaf bare
    ]
;
value build_mas_yas stem entry =
  let bare = fix stem "as"
  and decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
        [ decline Voc "an"
        ; decline Nom "aan"
        ; decline Acc "aa.msam"
        ; decline Ins "asaa"
        ; decline Dat "ase"
        ; decline Abl "asas"
        ; decline Gen "asas"
        ; decline Loc "asi"
        ])
   ; (Dual,
        [ decline Voc "aa.msau"
        ; decline Nom "aa.msau"
        ; decline Acc "aa.msau"
        ; decline Ins "obhyaam"
        ; decline Dat "obhyaam"
        ; decline Abl "obhyaam"
        ; decline Gen "asos"
        ; decline Loc "asos"
        ])
   ; (Plural,
        [ decline Voc "aa.msas"
        ; decline Nom "aa.msas"
        ; decline Acc "asas"
        ; decline Ins "obhis"
        ; decline Dat "obhyas"
        ; decline Abl "obhyas"
        ; decline Gen "asaam"
```

```
          ; decline Loc "a.hsu" (* decline Loc "assu" *)
          ])
    ]
    ; Bare Noun bare
    ; Avyayaf bare
    ]
;
value build_mas_vas stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "van"
          ; decline Nom "vaan"
          ; decline Acc "vaa.msam"
          ; decline Ins "u.saa"
          ; decline Dat "u.se"
          ; decline Abl "u.sas"
          ; decline Gen "u.sas"
          ; decline Loc "u.si"
          ])
    ; (Dual,
          [ decline Voc "vaa.msau"
          ; decline Nom "vaa.msau"
          ; decline Acc "vaa.msau"
          ; decline Ins "vadbhyaam"
          ; decline Dat "vadbhyaam"
          ; decline Abl "vadbhyaam"
          ; decline Gen "u.sos"
          ; decline Loc "u.sos"
          ])
    ; (Plural,
          [ decline Voc "vaa.msas"
          ; decline Nom "vaa.msas"
          ; decline Acc "u.sas"
          ; decline Ins "vadbhis"
          ; decline Dat "vadbhyas"
          ; decline Abl "vadbhyas"
          ; decline Gen "u.saam"
          ; decline Loc "vatsu"
```

```
        ])
    ]
    ; Bare Noun (fix stem "vat") (* eg vidvat- *)
    ; Avyayaf (fix stem "vas")
    ]
;
(* i is dropped before u.s - Macdonnel §89a *)
value build_mas_ivas stem entry =
  let decline case suff = (case, fix stem suff)
  and declinev case suff = (case, fix stem ("i" ^ suff)) in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
        [ declinev Voc "van"
        ; declinev Nom "vaan"
        ; declinev Acc "vaa.msam"
        ; decline Ins "u.saa"
        ; decline Dat "u.se"
        ; decline Abl "u.sas"
        ; decline Gen "u.sas"
        ; decline Loc "u.si"
        ])
   ; (Dual,
        [ declinev Voc "vaa.msau"
        ; declinev Nom "vaa.msau"
        ; declinev Acc "vaa.msau"
        ; declinev Ins "vadbhyaam"
        ; declinev Dat "vadbhyaam"
        ; declinev Abl "vadbhyaam"
        ; decline Gen "u.sos"
        ; decline Loc "u.sos"
        ])
   ; (Plural,
        [ declinev Voc "vaa.msas"
        ; declinev Nom "vaa.msas"
        ; decline Acc "u.sas"
        ; declinev Ins "vadbhis"
        ; declinev Dat "vadbhyas"
        ; declinev Abl "vadbhyas"
        ; decline Gen "u.saam"
```

```
        ; declinev Loc "vatsu"
        ])
   ]
   ; Avyayaf (fix stem "vas")
   ]
;
value build_mas_aac stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
        [ decline Voc "f"
        ; decline Nom "f"
        ; decline Acc "~ncam"
        ; decline Ins "caa"
        ; decline Dat "ce"
        ; decline Abl "cas"
        ; decline Gen "cas"
        ; decline Loc "ci"
        ])
   ; (Dual,
        [ decline Voc "~ncau"
        ; decline Nom "~ncau"
        ; decline Acc "~ncau"
        ; decline Ins "gbhyaam"
        ; decline Dat "gbhyaam"
        ; decline Abl "gbhyaam"
        ; decline Gen "cos"
        ; decline Loc "cos"
        ])
   ; (Plural,
        [ decline Voc "~ncas"
        ; decline Nom "~ncas"
        ; decline Acc "cas"
        ; decline Ins "gbhis"
        ; decline Dat "gbhyas"
        ; decline Abl "gbhyas"
        ; decline Gen "caam"
        ; decline Loc "k.su"
        ])
```

```
    ]
    ; Bare Noun (fix stem "f") (* nasale gutturale *)
    ; Avyayaf (fix stem "~nc") (* ? *)
    ]
;
value build_mas_yac stem entry =
  let decline case suff = (case, fix stem suff)
  and prevoc = if stem = revcode "tir" then "azc"
                                        else "iic" in
      (* exception tiryac -¿ weakest stem tiriic in prevocalic flexions *)
  enter entry
    [ Declined Noun Mas
    [ (Singular,
         [ decline Voc "yaf"
         ; decline Nom "yaf"
         ; decline Acc "ya~ncam"
         ; decline Ins (prevoc ^ "aa")
         ; decline Dat (prevoc ^ "e")
         ; decline Abl (prevoc ^ "as")
         ; decline Gen (prevoc ^ "as")
         ; decline Loc (prevoc ^ "i")
         ])
    ; (Dual,
         [ decline Voc "ya~ncau"
         ; decline Nom "ya~ncau"
         ; decline Acc "ya~ncau"
         ; decline Ins "yagbhyaam"
         ; decline Dat "yagbhyaam"
         ; decline Abl "yagbhyaam"
         ; decline Gen (prevoc ^ "os")
         ; decline Loc (prevoc ^ "os")
         ])
    ; (Plural,
         [ decline Voc "ya~ncas"
         ; decline Nom "ya~ncas"
         ; decline Acc (prevoc ^ "as")
         ; decline Ins "yagbhis"
         ; decline Dat "yagbhyas"
         ; decline Abl "yagbhyas"
         ; decline Gen (prevoc ^ "aam")
```

```
          ; decline Loc "yak.su"
          ])
    ]
    ; Bare Noun (fix stem "yak")
    ; Avyayaf (fix stem "yaf") (* ? *)
    ]
;
value build_mas_vac stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "vaf"
          ; decline Nom "vaf"
          ; decline Acc "va~ncam"
          ; decline Ins "uucaa"
          ; decline Dat "uuce"
          ; decline Abl "uucas"
          ; decline Gen "uucas"
          ; decline Loc "uuci"
          ])
    ; (Dual,
          [ decline Voc "va~ncau"
          ; decline Nom "va~ncau"
          ; decline Acc "va~ncau"
          ; decline Ins "vagbhyaam"
          ; decline Dat "vagbhyaam"
          ; decline Abl "vagbhyaam"
          ; decline Gen "uucos"
          ; decline Loc "uucos"
          ])
    ; (Plural,
          [ decline Voc "va~ncas"
          ; decline Nom "va~ncas"
          ; decline Acc "uucas"
          ; decline Ins "vagbhis"
          ; decline Dat "vagbhyas"
          ; decline Abl "vagbhyas"
          ; decline Gen "uucaam"
          ; decline Loc "vak.su"
```

```
        ])
    ]
  ; Bare Noun (fix stem "vak")
  ; Avyayaf (fix stem "vaf") (* ? *)
  ]
;
value build_mas_ac stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
  [ Declined Noun Mas
  [ (Singular,
        [ decline Voc "af"
        ; decline Nom "af"
        ; decline Acc "a~ncam"
        ; decline Ins "iicaa"
        ; decline Dat "iice"
        ; decline Abl "iicas"
        ; decline Gen "iicas"
        ; decline Loc "iici"
        ])
  ; (Dual,
        [ decline Voc "a~ncau"
        ; decline Nom "a~ncau"
        ; decline Acc "a~ncau"
        ; decline Ins "agbhyaam"
        ; decline Dat "agbhyaam"
        ; decline Abl "agbhyaam"
        ; decline Gen "iicos"
        ; decline Loc "iicos"
        ])
  ; (Plural,
        [ decline Voc "a~ncas"
        ; decline Nom "a~ncas"
        ; decline Acc "iicas"
        ; decline Ins "agbhis"
        ; decline Dat "agbhyas"
        ; decline Abl "agbhyas"
        ; decline Gen "iicaam"
        ; decline Loc "ak.su"
        ])
```

```
    ]
    ; Bare Noun (fix stem "ak")
    ; Avyayaf (fix stem "af") (* ? *)
    ]
;
value build_pums pum pums entry = (* for pu.ms et napu.ms *)
(* hi.ms pu.ms no retroflexion of s - Whitney§183a Kale §113 *)
  let decline case suff = (case, List2.unstack pum (code suff))
  and declines case suff = (case, List2.unstack pums (code suff)) in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
         [ decline Voc "an"
         ; decline Nom "aan"
         ; decline Acc "aa.msam"
         ; declines Ins "aa"
         ; declines Dat "e"
         ; declines Abl "as"
         ; declines Gen "as"
         ; declines Loc "i"
         ])
   ; (Dual,
         [ decline Voc "aa.msau"
         ; decline Nom "aa.msau"
         ; decline Acc "aa.msau"
         ; decline Ins "bhyaam"
         ; decline Dat "bhyaam"
         ; decline Abl "bhyaam"
         ; declines Gen "os"
         ; declines Loc "os"
         ])
   ; (Plural,
         [ decline Voc "aa.msas"
         ; decline Nom "aa.msas"
         ; declines Acc "as"
         ; decline Ins "bhis"
         ; decline Dat "bhyas"
         ; decline Abl "bhyas"
         ; declines Gen "aam"
         ; declines Loc "u"
```

```
          ])
    ]
    ; Bare Noun (mirror pum) (* for pul  lifga *)
    ; Bare Noun (mirror pums) (* for pu.mzcala *)
    (* ; Avyayaf ? *)
    ]
;
value build_mas_vah stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "van"
          ; decline Nom "vaa.t"
          ; decline Acc "vaaham"
          ; decline Ins "ohaa" (* becomes auhaa by sandhi with a- *)
          ; decline Dat "ohe" (* Whitney 403 gives uuhaa etc *)
          ; decline Abl "ohas" (* but has special sandhi rule §137c *)
          ; decline Gen "ohas"
          ; decline Loc "ohi"
          ])
    ; (Dual,
          [ decline Voc "vaahau"
          ; decline Nom "vaahau"
          ; decline Acc "vaahau"
          ; decline Ins "vaa.dbhyaam"
          ; decline Dat "vaa.dbhyaam"
          ; decline Abl "vaa.dbhyaam"
          ; decline Gen "ohos"
          ; decline Loc "ohos"
          ])
    ; (Plural,
          [ decline Voc "vaahas"
          ; decline Nom "vaahas"
          ; decline Acc "ohas"
          ; decline Ins "vaa.dbhis"
          ; decline Dat "vaa.dbhyas"
          ; decline Abl "vaa.dbhyas"
          ; decline Gen "ohaam"
          ; decline Loc "vaa.tsu"
```

```
        ])
    ]
    ; Avyayaf (fix stem "vah")
    ]
;
value build_anadvah stem entry = (* ana.dvah *)
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Singular,
          [ decline Voc "van"
          ; decline Nom "vaan"
          ; decline Acc "vaaham"
          ; decline Ins "uhaa"
          ; decline Dat "uhe"
          ; decline Abl "uhas"
          ; decline Gen "uhas"
          ; decline Loc "uhi"
          ])
    ; (Dual,
          [ decline Voc "vaahau"
          ; decline Nom "vaahau"
          ; decline Acc "vaahau"
          ; decline Ins "udbhyaam"
          ; decline Dat "udbhyaam"
          ; decline Abl "udbhyaam"
          ; decline Gen "uhos"
          ; decline Loc "uhos"
          ])
    ; (Plural,
          [ decline Voc "vaahas"
          ; decline Nom "vaahas"
          ; decline Acc "uhas"
          ; decline Ins "udbhis"
          ; decline Dat "udbhyas"
          ; decline Abl "udbhyas"
          ; decline Gen "uhaam"
          ; decline Loc "utsu"
          ])
    ]
```

```
    ]
;
value build_neu_a stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry (
   [ Declined Noun Neu
   [ (Singular,if entry = "ubha" (* dual only *) then [] else
         [ decline Voc "a"
        (* decline Voc "am" - rare - disconnected for avoiding overgeneration *)
         ; decline Nom "am"
         ; decline Acc "am"
         ; decline Ins "ena"
         ; decline Dat "aaya"
         ; decline Abl "aat"
         ; decline Gen "asya"
         ; decline Loc "e"
         ])
   ; (Dual,
         [ decline Voc "e"
         ; decline Nom "e"
         ; decline Acc "e"
         ; decline Ins "aabhyaam"
         ; decline Dat "aabhyaam"
         ; decline Abl "aabhyaam"
         ; decline Gen "ayos"
         ; decline Loc "ayos"
         ])
   ; (Plural, if entry = "ubha" (* dual only *) then [] else let l =
         [ decline Voc "aani"
         ; decline Nom "aani"
         ; decline Acc "aani"
         ; decline Ins "ais"
         ; decline Dat "ebhyas"
         ; decline Abl "ebhyas"
         ; decline Gen "aanaam"
         ; decline Loc "esu"
         ] in if entry = "durita" then [ decline Nom "aa" :: l ] (* vedic *)
              else l)
   ]
   ; Bare Noun (wrap stem 1)
```

```
   ; Avyayaf (fix stem "am"); Avyayaf (fix stem "aat")
   ; Indecl Tas (fix stem "atas")
   ] @ (if a_n_iiv entry then [ Cvi (wrap stem 4) ] else []))
;
value build_neu_i trunc entry = (* stems in -i and -ii *)
  let stems = [ 3 :: trunc ]
  and steml = [ 4 :: trunc ] in
  let rstems = mirror stems
  and declines case suff = (case, fix stems suff)
  and declinel case suff = (case, fix steml suff) in
  enter entry
   [ Declined Noun Neu
   [ (Singular,
         [ declines Voc ""
         ; declines Nom ""
         ; declines Acc ""
         ; declines Ins "naa"
         ; declines Dat "ne"
         ; declines Abl "nas"
         ; declines Gen "nas"
         ; declines Loc "ni"
         ])
   ; (Dual,
         [ declines Voc "nii"
         ; declines Nom "nii"
         ; declines Acc "nii"
         ; declines Ins "bhyaam"
         ; declines Dat "bhyaam"
         ; declines Abl "bhyaam"
         ; declines Gen "nos"
         ; declines Loc "nos"
         ])
   ; (Plural,
         [ declinel Voc "ni"
         ; declinel Nom "ni"
         ; declinel Acc "ni"
         ; declines Ins "bhis"
         ; declines Dat "bhyas"
         ; declines Abl "bhyas"
         ; declinel Gen "naam"
```

```
        ; declines Loc "su"
        ])
    ]
    ; Bare Noun rstems
    ; Avyayaf rstems
    ]
;
value build_neu_u trunc entry = (* stems in -u and -uu *)
  let stems = [ 5 :: trunc ]
  and steml = [ 6 :: trunc ] in
  let declines case suff = (case, fix stems suff)
  and declinel case suff = (case, fix steml suff) in
  enter entry
   [ Declined Noun Neu
   [ (Singular,
        [ declines Voc ""
        ; declines Nom ""
        ; declines Acc ""
        ; declines Ins "naa"
        ; declines Dat "ne"
        ; declines Abl "nas"
        ; declines Gen "nas"
        ; declines Loc "ni"
        ])
   ; (Dual,
        [ declines Voc "nii"
        ; declines Nom "nii"
        ; declines Acc "nii"
        ; declines Ins "bhyaam"
        ; declines Dat "bhyaam"
        ; declines Abl "bhyaam"
        ; declines Gen "nos"
        ; declines Loc "nos"
        ])
   ; (Plural,
        [ declinel Voc "ni"
        ; declinel Nom "ni"
        ; declinel Acc "ni"
        ; declines Ins "bhis"
        ; declines Dat "bhyas"
```

```
          ; declines Abl "bhyas"
          ; declinel Gen "naam"
          ; declines Loc "su"
          ])
    ]
    ; Bare Noun (mirror stems)
    ; Avyayaf (mirror stems)
    ; Indecl Tas (fix stems "tas") (* eg vastutas *)
    ]
;
value build_neu_ri trunc entry =
  let stems = [ 7 :: trunc ]
  and steml = [ 8 :: trunc ] in
  let declines case suff = (case, fix stems suff)
  and declinel case suff = (case, fix steml suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ declines Voc ""
          ; declines Nom ""
          ; declines Acc ""
          ; declines Ins "naa"
          ; declines Dat "ne"
          ; declines Abl "nas"
          ; declines Gen "nas"
          ; declines Loc "ni"
          ])
    ; (Dual,
          [ declines Voc "nii"
          ; declines Nom "nii"
          ; declines Acc "nii"
          ; declines Ins "bhyaam"
          ; declines Dat "bhyaam"
          ; declines Abl "bhyaam"
          ; declines Gen "nos"
          ; declines Loc "nos"
          ])
    ; (Plural,
          [ declinel Voc "ni"
          ; declinel Nom "ni"
```

```
            ; declinel Acc "ni"
            ; declines Ins "bhis"
            ; declines Dat "bhyas"
            ; declines Abl "bhyas"
            ; declinel Gen "naam"
            ; declines Loc "su"
            ])
    ]
    ; Bare Noun (mirror stems)
    ; Avyayaf (mirror stems)
    ]
;
value build_neu_yas stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
            [ decline Voc "as"
            ; decline Nom "as"
            ; decline Acc "as"
            ; decline Ins "asaa"
            ; decline Dat "ase"
            ; decline Abl "asas"
            ; decline Gen "asas"
            ; decline Loc "asi"
            ])
    ; (Dual,
            [ decline Voc "asii"
            ; decline Nom "asii"
            ; decline Acc "asii"
            ; decline Ins "obhyaam"
            ; decline Dat "obhyaam"
            ; decline Abl "obhyaam"
            ; decline Gen "asos"
            ; decline Loc "asos"
            ])
    ; (Plural,
            [ decline Voc "aa.msi"
            ; decline Nom "aa.msi"
            ; decline Acc "aa.msi"
```

```
        ; decline Ins "obhis"
        ; decline Dat "obhyas"
        ; decline Abl "obhyas"
        ; decline Gen "asaam"
        ; decline Loc "a.hsu" (* decline Loc "assu" *)
        ])
   ]
   ; Bare Noun (fix stem "as")
   ; Avyayaf (fix stem "as")
   ]
;
value build_neu_vas stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Neu
   [ (Singular,
        [ decline Voc "vat"
        ; decline Nom "vat"
        ; decline Acc "vat"
        ; decline Ins "u.saa"
        ; decline Dat "u.se"
        ; decline Abl "u.sas"
        ; decline Gen "u.sas"
        ; decline Loc "u.si"
        ])
   ; (Dual,
        [ decline Voc "u.sii"
        ; decline Nom "u.sii"
        ; decline Acc "u.sii"
        ; decline Ins "vadbhyaam"
        ; decline Dat "vadbhyaam"
        ; decline Abl "vadbhyaam"
        ; decline Gen "u.sos"
        ; decline Loc "u.sos"
        ])
   ; (Plural,
        [ decline Voc "vaa.msi"
        ; decline Nom "vaa.msi"
        ; decline Acc "vaa.msi"
        ; decline Ins "vadbhis"
```

```
          ; decline Dat "vadbhyas"
          ; decline Abl "vadbhyas"
          ; decline Gen "u.saam"
          ; decline Loc "vatsu"
          ])
    ]
    ; Bare Noun (fix stem "vat") (* eg vidvat- *)
    ; Avyayaf (fix stem "vas") (* vat Acc ? *)
    ]
;
(* i is dropped before u.s - Macdonnel §89a *)
value build_neu_ivas stem entry =
  let decline case suff = (case, fix stem suff)
  and declinev case suff = (case, fix stem ("i" ^ suff)) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ declinev Voc "vat"
          ; declinev Nom "vat"
          ; declinev Acc "vat"
          ; decline Ins "u.saa"
          ; decline Dat "u.se"
          ; decline Abl "u.sas"
          ; decline Gen "u.sas"
          ; decline Loc "u.si"
          ])
    ; (Dual,
          [ decline Voc "u.sii"
          ; decline Nom "u.sii"
          ; decline Acc "u.sii"
          ; declinev Ins "vadbhyaam"
          ; declinev Dat "vadbhyaam"
          ; declinev Abl "vadbhyaam"
          ; decline Gen "u.sos"
          ; decline Loc "u.sos"
          ])
    ; (Plural,
          [ declinev Voc "vaa.msi"
          ; declinev Nom "vaa.msi"
          ; declinev Acc "vaa.msi"
```

```
          ; declinev Ins "vadbhis"
          ; declinev Dat "vadbhyas"
          ; declinev Abl "vadbhyas"
          ; decline Gen "u.saam"
          ; declinev Loc "vatsu"
          ])
    ]
    ; Bare Noun (fix stem "ivat")
    ; Avyayaf (fix stem "ivas") (* why not ivat Acc ? *)
    ]
;
value build_neu_red stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ decline Voc "t"
          ; decline Nom "t"
          ; decline Acc "tam"
          ; decline Ins "taa"
          ; decline Dat "te"
          ; decline Abl "tas"
          ; decline Gen "tas"
          ; decline Loc "ti"
          ])
    ; (Dual,
          [ decline Voc "tii"
          ; decline Nom "tii"
          ; decline Acc "tii"
          ; decline Ins "dbhyaam"
          ; decline Dat "dbhyaam"
          ; decline Abl "dbhyaam"
          ; decline Gen "tos"
          ; decline Loc "tos"
          ])
    ; (Plural,
          [ decline Voc "ti"
          ; decline Voc "nti"
          ; decline Nom "ti"
          ; decline Nom "nti"
```

```
          ; decline Acc "ti"
          ; decline Acc "nti"
          ; decline Ins "dbhis"
          ; decline Dat "dbhyas"
          ; decline Abl "dbhyas"
          ; decline Gen "taam"
          ; decline Loc "tsu"
          ])
    ]
    ; Avyayaf (fix stem "tam")
    ]
;
value build_neu_at stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ decline Voc "t"
          ; decline Nom "t"
          ; decline Acc "t"
          ; decline Ins "taa"
          ; decline Dat "te"
          ; decline Abl "tas"
          ; decline Gen "tas"
          ; decline Loc "ti"
          ])
    ; (Dual,
          [ decline Voc "tii"
          ; decline Voc "ntii"
          ; decline Nom "tii"
          ; decline Nom "ntii"
          ; decline Acc "tii"
          ; decline Acc "ntii"
          ; decline Ins "dbhyaam"
          ; decline Dat "dbhyaam"
          ; decline Abl "dbhyaam"
          ; decline Gen "tos"
          ; decline Loc "tos"
          ])
    ; (Plural,
```

```
        [ decline Voc "nti"
        ; decline Nom "nti"
        ; decline Acc "nti"
        ; decline Ins "dbhis"
        ; decline Dat "dbhyas"
        ; decline Abl "dbhyas"
        ; decline Gen "taam"
        ; decline Loc "tsu"
        ])
  ]
  ; Avyayaf (fix stem "tam") (* why not Acc ? *)
  ]
;
value build_neu_mahat stem entry =
 let decline case suff = (case, fix stem suff) in
 enter entry
  [ Declined Noun Neu
  [ (Singular,
        [ decline Voc "at"
        ; decline Nom "at"
        ; decline Acc "at"
        ; decline Ins "ataa"
        ; decline Dat "ate"
        ; decline Abl "atas"
        ; decline Gen "atas"
        ; decline Loc "ati"
        ])
  ; (Dual,
        [ decline Voc "atii"
        ; decline Nom "atii"
        ; decline Acc "atii"
        ; decline Ins "adbhyaam"
        ; decline Dat "adbhyaam"
        ; decline Abl "adbhyaam"
        ; decline Gen "atos"
        ; decline Loc "atos"
        ])
  ; (Plural,
        [ decline Voc "aanti"
        ; decline Nom "aanti"
```

```
          ; decline Acc "aanti"
          ; decline Ins "adbhis"
          ; decline Dat "adbhyas"
          ; decline Abl "adbhyas"
          ; decline Gen "ataam"
          ; decline Loc "atsu"
          ])
    ]
    ; Avyayaf (fix stem "atam")
    ]
;
(* pronominal use of aatman in sg for refl use of 3 genders and 3 numbers *)
value build_aatman entry =
  let stem = revcode "aatm" in
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun (Deictic Self)
    [ (Singular,
          [ decline Voc "an"
          ; decline Nom "aa"
          ; decline Acc "aanam"
          ; decline Ins "anaa"
          ; decline Dat "ane"
          ; decline Abl "anas"
          ; decline Gen "anas"
          ; decline Loc "ani"
          ])
    ]
    ; Bare Pron (code "aatma")
    ; Avyayaf (code "aatmam") (* aatmaanam Acc ? *)
    ; Cvi (code "aatmii")
    ]
;
value build_neu_yuvan entry =
  let stem = [ 42 ] (* y *) in
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ decline Voc "uva"
```

```
          ; decline Voc "uvan"
          ; decline Nom "uva"
          ; decline Acc "uva"
          ; decline Ins "uunaa"
          ; decline Dat "uune"
          ; decline Abl "uunas"
          ; decline Gen "uunas"
          ; decline Loc "uuni"
          ])
    ; (Dual,
          [ decline Voc "uvanii"
          ; decline Nom "uvanii"
          ; decline Acc "uvanii"
          ; decline Ins "uvabhyaam"
          ; decline Dat "uvabhyaam"
          ; decline Abl "uvabhyaam"
          ; decline Gen "uunos"
          ; decline Loc "uunos"
          ])
    ; (Plural,
          [ decline Voc "uvaani"
          ; decline Nom "uvaani"
          ; decline Acc "uvaani"
          ; decline Ins "uvabhis"
          ; decline Dat "uvabhyas"
          ; decline Abl "uvabhyas"
          ; decline Gen "uunaam"
          ; decline Loc "uvasu"
          ])
    ]
    ; Avyayaf (fix stem "uvam") (* uva Acc ? *)
    ]
;
value build_neu_brahman entry =
  let stem = revcode "brahm" in
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ decline Voc "a"
```

```
      ; decline Nom "a"
      ; decline Acc "a"
      ; decline Ins "a.naa"
      ; decline Dat "a.ne"
      ; decline Abl "a.nas"
      ; decline Gen "a.nas"
      ; decline Loc "a.ni"
      ])
  ; (Dual,
      [ decline Voc "a.nii"
      ; decline Nom "a.nii"
      ; decline Acc "a.nii"
      ; decline Ins "abhyaam"
      ; decline Dat "abhyaam"
      ; decline Abl "abhyaam"
      ; decline Gen "a.nos"
      ; decline Loc "a.nos"
      ])
  ; (Plural,
      [ decline Voc "aa.ni"
      ; decline Nom "aa.ni"
      ; decline Acc "aa.ni"
      ; decline Ins "abhis"
      ; decline Dat "abhyas"
      ; decline Abl "abhyas"
      ; decline Gen "a.naam"
      ; decline Loc "asu"
      ])
  ]
  ; Bare Noun (code "brahma")
  ; Avyayaf (code "brahma") (* Acc *)
  ]
;
value build_aksan stem entry =
 (* stem = ak.san, asthan, dadhan, sakthan Whitney §431 Kale§126 *)
 let decline case suff = (case, fix stem suff) in
 enter entry
  [ Declined Noun Neu
  [ (Singular,
      [ decline Voc "e"
```

```
        ; decline Nom "i"
        ; decline Acc "i"
        ; decline Ins "naa"
        ; decline Dat "ne"
        ; decline Abl "nas"
        ; decline Gen "nas"
        ; decline Loc "ni"
        ; decline Loc "ani" (* P{7,1,75} *)
        ])
  ; (Dual, let l =
        [ decline Voc "inii"
        ; decline Nom "inii"
        ; decline Acc "inii"
        ; decline Ins "ibhyaam"
        ; decline Dat "ibhyaam"
        ; decline Abl "ibhyaam"
        ; decline Gen "nos"
        ; decline Loc "nos"
        ] in if entry ="ak.san" then
        [ decline Voc "ii"
        ; decline Nom "ii"
        ; decline Acc "ii"
        ] @ l (* Vedic: Sun and moon *)
     else l)
  ; (Plural,
        [ decline Voc "iini"
        ; decline Nom "iini"
        ; decline Acc "iini"
        ; decline Acc "aani" (* MW véd. sakthaani RV10,86,16 AV6,9,1 *)
        ; decline Ins "ibhis"
        ; decline Dat "ibhyas"
        ; decline Abl "ibhyas"
        ; decline Gen "naam"
        ; decline Loc "isu"
        ])
  ]
  ; Bare Noun (fix stem "i") (* also indirectly generated by var subentry *)
  ; Avyayaf (fix stem "i") (* Acc *)
  ]
;
```

```
value build_ahan stem entry = (* stem = "ah" *)
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Neu
   [ (Singular,
         [ decline Voc "ar"
         ; decline Nom "ar"
         ; decline Acc "ar"
         ; decline Ins "naa"
         ; decline Dat "ne"
         ; decline Abl "nas"
         ; decline Gen "nas"
         ; decline Loc "ni"
         ; decline Loc "ani"
         ])
   ; (Dual,
         [ decline Voc "nii"
         ; decline Voc "anii"
         ; decline Nom "nii"
         ; decline Nom "anii"
         ; decline Acc "nii"
         ; decline Acc "anii"
         ; decline Ins "obhyaam"
         ; decline Dat "obhyaam"
         ; decline Abl "obhyaam"
         ; decline Gen "nos"
         ; decline Loc "nos"
         ])
   ; (Plural,
         [ decline Voc "aani"
         ; decline Nom "aani"
         ; decline Acc "aani"
         ; decline Ins "obhis"
         ; decline Dat "obhyas"
         ; decline Abl "obhyas"
         ; decline Gen "naam"
         ; decline Loc "a.hsu" (* decline Loc "assu" *)
         ])
   ]
   ; Bare Noun (fix stem "ar")
```

```
    ; Bare Noun (fix stem "as") (* before r Pan8;2;68 *)
    (* Avyayaf (fix stem "am") NO pratyaham Acc of pratyaha *)
    ; Avyayaf (fix stem "ar") (* Acc pratyaha.h *)
    ]
;
value build_uudhan stem entry = (* stem = "uudh" *) (* Whitney §430d *)
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ decline Voc "ar"
          ; decline Nom "ar"
          ; decline Acc "ar"
(* ; decline Voc "as" redundant *)
(* ; decline Nom "as" redundant *)
(* ; decline Acc "as" redundant *)
          ; decline Ins "naa"
          ; decline Dat "ne"
          ; decline Abl "nas"
          ; decline Gen "nas"
          ; decline Loc "an"
          ; decline Loc "ani"
          ])
    ; (Dual,
          [ decline Voc "nii"
          ; decline Voc "anii"
          ; decline Nom "nii"
          ; decline Nom "anii"
          ; decline Acc "nii"
          ; decline Acc "anii"
          ; decline Ins "abhyaam"
          ; decline Dat "abhyaam"
          ; decline Abl "abhyaam"
          ; decline Gen "nos"
          ; decline Loc "nos"
          ])
    ; (Plural,
          [ decline Voc "aani"
          ; decline Nom "aani"
          ; decline Acc "aani"
```

```
          ; decline Ins "abhis"
          ; decline Dat "abhyas"
          ; decline Abl "abhyas"
          ; decline Gen "naam"
          ; decline Loc "a.hsu" (* decline Loc "assu" *)
          ])
    ]
    ; Bare Noun (code "uudhar")
    ; Avyayaf (code "uudham")
    ; Avyayaf (code "uudha")
    ]
;
value build_neu_in stem entry =
  let decline case suff = (case, fix stem suff)
  and bare = wrap stem 3 in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ decline Voc "in"
          ; decline Voc "i"
          ; decline Nom "i"
          ; decline Acc "i"
          ; decline Ins "inaa"
          ; decline Dat "ine"
          ; decline Abl "inas"
          ; decline Gen "inas"
          ; decline Loc "ini"
          ])
    ; (Dual,
          [ decline Voc "inii"
          ; decline Nom "inii"
          ; decline Acc "inii"
          ; decline Ins "ibhyaam"
          ; decline Dat "ibhyaam"
          ; decline Abl "ibhyaam"
          ; decline Gen "inos"
          ; decline Loc "inos"
          ])
    ; (Plural,
          [ decline Voc "iini"
```

```
          ; decline Nom "iini"
          ; decline Acc "iini"
          ; decline Ins "ibhis"
          ; decline Dat "ibhyas"
          ; decline Abl "ibhyas"
          ; decline Gen "inaam"
          ; decline Loc "i.su"
          ])
    ]
    ; Bare Noun bare (* same as Acc *)
    ; Avyayaf bare
    ]
;
value build_neu_aac stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ decline Voc "k"
          ; decline Nom "k"
          ; decline Acc "~ncam"
          ; decline Ins "caa"
          ; decline Dat "ce"
          ; decline Abl "cas"
          ; decline Gen "cas"
          ; decline Loc "ci"
          ])
    ; (Dual,
          [ decline Voc "cii"
          ; decline Nom "cii"
          ; decline Acc "cii"
          ; decline Ins "gbhyaam"
          ; decline Dat "gbhyaam"
          ; decline Abl "gbhyaam"
          ; decline Gen "cos"
          ; decline Loc "cos"
          ])
    ; (Plural,
          [ decline Voc "~nci"
          ; decline Nom "~nci"
```

```
        ; decline Acc "~nci"
        ; decline Ins "gbhis"
        ; decline Dat "gbhyas"
        ; decline Abl "gbhyas"
        ; decline Gen "caam"
        ; decline Loc "k.su"
        ])
    ]
  ; Bare Noun (fix stem "k") (* eg praaguttara *)
  ]
;
value build_neu_yac stem entry =
  let prevoc = if stem = revcode "tir" then "azc"
                                         else "iic" in
              (* exception tiryac -¿ tiriic in prevocalic flexions *)
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Neu
   [ (Singular,
        [ decline Voc "yak"
        ; decline Nom "yak"
        ; decline Acc "yak"
        ; decline Ins (prevoc ^ "aa")
        ; decline Dat (prevoc ^ "e")
        ; decline Abl (prevoc ^ "as")
        ; decline Gen (prevoc ^ "as")
        ; decline Loc (prevoc ^ "i")
        ])
   ; (Dual,
        [ decline Voc (prevoc ^ "ii")
        ; decline Nom (prevoc ^ "ii")
        ; decline Acc (prevoc ^ "ii")
        ; decline Ins "yagbhyaam"
        ; decline Dat "yagbhyaam"
        ; decline Abl "yagbhyaam"
        ; decline Gen (prevoc ^ "os")
        ; decline Loc (prevoc ^ "os")
        ])
   ; (Plural,
        [ decline Voc "ya~nci"
```

```
          ; decline Nom "ya~nci"
          ; decline Acc "ya~nci"
          ; decline Ins "yagbhis"
          ; decline Dat "yagbhyas"
          ; decline Abl "yagbhyas"
          ; decline Gen (prevoc ^ "aam")
          ; decline Loc "yak.su"
          ])
    ] ]
;
value build_neu_vac stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
          [ decline Voc "vak"
          ; decline Nom "vak"
          ; decline Acc "vak"
          ; decline Ins "uucaa"
          ; decline Dat "uuce"
          ; decline Abl "uucas"
          ; decline Gen "uucas"
          ; decline Loc "uuci"
          ])
    ; (Dual,
          [ decline Voc "uucii"
          ; decline Nom "uucii"
          ; decline Acc "uucii"
          ; decline Ins "vagbhyaam"
          ; decline Dat "vagbhyaam"
          ; decline Abl "vagbhyaam"
          ; decline Gen "uucos"
          ; decline Loc "uucos"
          ])
    ; (Plural,
          [ decline Voc "va~nci"
          ; decline Nom "va~nci"
          ; decline Acc "va~nci"
          ; decline Ins "vagbhis"
          ; decline Dat "vagbhyas"
```

```
        ; decline Abl "vagbhyas"
        ; decline Gen "uucaam"
        ; decline Loc "vak.su"
        ])
  ]
  ; Avyayaf (code "vacam") (* check *)
  ]
;
value build_neu_ac stem entry =
 let decline case suff = (case, fix stem suff) in
 enter entry
  [ Declined Noun Neu
  [ (Singular,
        [ decline Voc "ak"
        ; decline Nom "ak"
        ; decline Acc "ak"
        ; decline Ins "iicaa"
        ; decline Dat "iice"
        ; decline Abl "iicas"
        ; decline Gen "iicas"
        ; decline Loc "iici"
        ])
  ; (Dual,
        [ decline Voc "iicii"
        ; decline Nom "iicii"
        ; decline Acc "iicii"
        ; decline Ins "agbhyaam"
        ; decline Dat "agbhyaam"
        ; decline Abl "agbhyaam"
        ; decline Gen "iicos"
        ; decline Loc "iicos"
        ])
  ; (Plural,
        [ decline Voc "a~nci"
        ; decline Nom "a~nci"
        ; decline Acc "a~nci"
        ; decline Ins "agbhis"
        ; decline Dat "agbhyas"
        ; decline Abl "agbhyas"
        ; decline Gen "iicaam"
```

```
        ; decline Loc "ak.su"
        ])
    ]
    ; Avyayaf (code "acam")
    ]
;
value build_neu_aas stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Neu
    [ (Singular,
        [ decline Ins "aa"
        ; decline Ins "ayaa"
        ; decline Abl "as"
        ])
    ] ]
;
value build_fem_aa stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry (
    [ Declined Noun Fem
    [ (Singular, if entry = "ubha" then [] else let l =
        [ if entry = "allaa" ∨ entry = "akkaa" (* Pan7,3,107 *)
          then decline Voc "a"
          else decline Voc "e"
        ; decline Nom "aa"
        ; decline Acc "aam"
        ; decline Ins "ayaa"
        ; decline Dat "aayai"
        ; decline Abl "aayaas"
        ; decline Gen "aayaas"
        ; decline Loc "aayaam"
        ] in if entry = "ambaa" then
        [ decline Voc "a" :: l ] (* Pan7,3,107 but also ambe vedic *)
            else if entry = "guha" then (* guhaa fde guha *)
        [ decline Loc "aa" :: l ] (* Vedic *)
            else l)
    ; (Dual,
        [ decline Voc "e"
        ; decline Nom "e"
```

```
          ; decline Acc "e"
          ; decline Ins "aabhyaam"
          ; decline Dat "aabhyaam"
          ; decline Abl "aabhyaam"
          ; decline Gen "ayos"
          ; decline Loc "ayos"
          ])
    ; (Plural, if entry = "ubha" then [] else
          [ decline Voc "aas"
          ; decline Nom "aas"
          ; decline Acc "aas"
          ; decline Ins "aabhis"
          ; decline Dat "aabhyas"
          ; decline Abl "aabhyas"
          ; decline Gen "aanaam"
          ; decline Loc "aasu"
          ])
    ]
    ; Avyayaf (fix stem "am") (* acc of neuter stem with hrasva of vowel *)
    ] @ (if aa_iiv entry then [ Cvi (wrap stem 4) ] else []))
;
(* vedic g = Fem, rare (jaa) Whitney 351 *)
value build_mono_aa g stem entry =
  let decline case suff = (case,fix stem suff) in
  enter entry
    [ Declined Noun g
    [ (Singular,
          [ decline Voc "aas"
          ; decline Nom "aas"
          ; decline Acc "aam"
          ; decline Ins "aa"
          ; decline Dat "e"
          ; decline Abl "as"
          ; decline Gen "as"
          ; decline Loc "i"
          ])
    ; (Dual,
          [ decline Voc "au"
          ; decline Nom "au"
          ; decline Acc "au"
```

```
        ; decline Ins "aabhyaam"
        ; decline Dat "aabhyaam"
        ; decline Abl "aabhyaam"
        ; decline Gen "os"
        ; decline Loc "os"
        ])
  ; (Plural,
        [ decline Voc "aas"
        ; decline Nom "aas"
        ; decline Acc "aas" (* Whitney *)
        ; decline Acc "as" (* Paninian form, according to Deshpande *)
        ; decline Ins "aabhis"
        ; decline Dat "aabhyas"
        ; decline Abl "aabhyas"
        ; decline Gen "aam"
        ; decline Gen "anaam"
        ; decline Loc "aasu"
        ])
  ]
  ; Avyayaf (fix stem "am") (* acc of neuter stem with hrasva of vowel *)
  ]
;
(* gandharva Haahaa Tirupati and pkt raa.naa *)
value build_mas_aa_no_root stem entry =
 let decline case suff = (case, fix stem suff) in
 enter entry
  [ Declined Noun Mas
  [ (Singular,
        [ decline Voc "aas"
        ; decline Nom "aas"
        ; decline Acc "aam"
        ; decline Ins "aa"
        ; decline Dat "ai"
        ; decline Abl "aas"
        ; decline Gen "aas"
        ; decline Loc "e"
        ])
  ; (Dual,
        [ decline Voc "au"
        ; decline Nom "au"
```

```
          ; decline Acc "au"
          ; decline Ins "aabhyaam"
          ; decline Dat "aabhyaam"
          ; decline Abl "aabhyaam"
          ; decline Gen "aus"
          ; decline Loc "aus"
          ])
   ; (Plural,
          [ decline Voc "aas"
          ; decline Nom "aas"
          ; decline Acc "aan"
          ; decline Ins "aabhis"
          ; decline Dat "aabhyas"
          ; decline Abl "aabhyas"
          ; decline Gen "aam"
          ; decline Loc "aasu"
          ])
   ] ]
;
(* Special for gandharva Huuhuu Tirupati *)
(* Also a few exceptions *)
value build_huuhuu entry =
  let stem = revcode "huuh" in
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun Mas
   [ (Singular,
          [ decline Voc "uus"
          ; decline Nom "uus"
          ; decline Acc "uum"
          ; decline Ins "vaa"
          ; decline Dat "ve"
          ; decline Abl "vas"
          ; decline Gen "vas"
          ; decline Loc "vi"
          ])
   ; (Dual,
          [ decline Voc "vau"
          ; decline Nom "vau"
          ; decline Acc "vau"
```

```
        ; decline Ins "uubhyaam"
        ; decline Dat "uubhyaam"
        ; decline Abl "uubhyaam"
        ; decline Gen "vau"
        ; decline Loc "vau"
        ])
   ; (Plural,
        [ decline Voc "vas"
        ; decline Nom "vas"
        ; decline Acc "uun"
        ; decline Ins "uubhis"
        ; decline Dat "uubhyas"
        ; decline Abl "uubhyas"
        ; decline Gen "vaam"
        ; decline Loc "uu.su"
        ])
   ] ]
;
value build_fem_i stem trunc entry =
  let declines case suff = (case, fix stem suff)
  and declineg case suff = (case, fix [ 10 :: trunc ] suff)
  and declinel case suff = (case, fix [ 4 :: trunc ] suff)
  and declinau case = (case, wrap trunc 13) in
  enter entry (
   [ Declined Noun Fem
   [ (Singular,
        [ declineg Voc ""
        ; declines Nom "s"
        ; declines Acc "m"
        ; declines Ins "aa"
        ; declines Dat "ai"
        ; declineg Dat "e"
        ; declines Abl "aas"
        ; declineg Abl "s"
        ; declines Gen "aas"
        ; declineg Gen "s"
        ; declines Loc "aam"
        ; declinau Loc
        ])
   ; (Dual,
```

```
            [ declinel Voc ""
            ; declinel Nom ""
            ; declinel Acc ""
            ; declines Ins "bhyaam"
            ; declines Dat "bhyaam"
            ; declines Abl "bhyaam"
            ; declines Gen "os"
            ; declines Loc "os"
            ])
    ; (Plural,
            [ declineg Voc "as"
            ; declineg Nom "as"
            ; declinel Acc "s"
            ; declines Ins "bhis"
            ; declines Dat "bhyas"
            ; declines Abl "bhyas"
            ; declinel Gen "naam"
            ; declines Loc "su"
            ])
    ]
    ; Bare Noun (mirror stem)
    ; Avyayaf (mirror stem) (* actually acc of neuter stem *)
    ; Indecl Tas (fix stem "tas")
    ] @ (if entry = "vi.mzati"
            then [ Bare Noun (mirror trunc) (* vi.mzat *) ]
            else []))
;
(* NB concerning Avyayaf of stems ending in long vowels. According to Pan2,4,18 avyayi-
ibhaava compounds are of neuter gender, incurring hrasva of ifc stem *)
value build_fem_ii trunc entry =
  let stems = [ 3 :: trunc ]
  and steml = [ 4 :: trunc ] in
  let declines case suff = (case, fix stems suff)
  and declinel case suff = (case, fix steml suff) in
  enter entry (
   [ Declined Noun Fem
   [ (Singular,
            [ declines Voc ""
            ; declinel Nom ""
            ; declinel Acc "m"
```

```
          ; declines Ins "aa"
          ; declines Dat "ai"
          ; declines Abl "aas"
          ; declines Gen "aas"
          ; declines Loc "aam"
          ])
    ; (Dual, if entry = "ubhayii" then [] else
          [ declines Voc "au"
          ; declines Nom "au"
          ; declines Acc "au"
          ; declinel Ins "bhyaam"
          ; declinel Dat "bhyaam"
          ; declinel Abl "bhyaam"
          ; declines Gen "os"
          ; declines Loc "os"
          ])
    ; (Plural,
          [ declines Voc "as"
          ; declines Nom "as"
          ; declinel Acc "s"
          ; declinel Ins "bhis"
          ; declinel Dat "bhyas"
          ; declinel Abl "bhyas"
          ; declinel Gen "naam"
          ; declinel Loc "su"
          ])
    ]
    ; Bare Noun (mirror steml)
    ; Bare Noun (mirror stems) (* Pan6,3,61 *)
    ; Avyayaf (mirror stems)
    ] @ match entry with
        [ "nadii" | "paur.namasii" | "aagrahaaya.nii"
          → [ Avyayaf (fix trunc "am") ]
        | _ → []
        ])
;
(* g = Fem, rarely Mas *)
value build_mono_ii g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
```

```
[ Declined Noun g
[ (Singular,
    [ decline Voc "iis"
    ; decline Nom "iis"
    ; decline Acc "iyam"
    ; decline Ins "iyaa"
    ; decline Dat "iye"
    ; decline Dat "iyai"
    ; decline Abl "iyas"
    ; decline Abl "iyaas"
    ; decline Gen "iyas"
    ; decline Gen "iyaas"
    ; decline Loc "iyi"
    ; decline Loc "iyaam" (* niyaam Kale§77 p46 *)
    ])
; (Dual,
    [ decline Voc "iyau"
    ; decline Nom "iyau"
    ; decline Acc "iyau"
    ; decline Ins "iibhyaam"
    ; decline Dat "iibhyaam"
    ; decline Abl "iibhyaam"
    ; decline Gen "iyos"
    ; decline Loc "iyos"
    ])
; (Plural,
    [ decline Voc "iyas"
    ; decline Nom "iyas"
    ; decline Acc "iyas"
    ; decline Ins "iibhis"
    ; decline Dat "iibhyas"
    ; decline Abl "iibhyas"
    ; decline Gen "iyaam"
    ; decline Gen "iinaam"
    ; decline Loc "ii.su"
    ])
]
; Bare Noun (wrap stem 4) (* productive ? shortened ? *)
; Avyayaf (wrap stem 3)
]
```

```
;
(* g = Mas scheme 42 Bucknell p26 p90 *)
value build_bicons_ii g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun g
   [ (Singular,
         [ decline Voc "iis"
         ; decline Nom "iis"
         ; decline Acc "iyam"
         ; decline Ins "iyaa"
         ; decline Dat "iye"
         ; decline Abl "iyas"
         ; decline Gen "iyas"
         ; decline Loc "iyi"
         ])
   ; (Dual,
         [ decline Voc "iyau"
         ; decline Nom "iyau"
         ; decline Acc "iyau"
         ; decline Ins "iibhyaam"
         ; decline Dat "iibhyaam"
         ; decline Abl "iibhyaam"
         ; decline Gen "iyos"
         ; decline Loc "iyos"
         ])
   ; (Plural,
         [ decline Voc "iyas"
         ; decline Nom "iyas"
         ; decline Acc "iyas"
         ; decline Ins "iibhis"
         ; decline Dat "iibhyas"
         ; decline Abl "iibhyas"
         ; decline Gen "iyaam"
         ; decline Loc "ii.su"
         ])
   ]
   ; Bare Noun (wrap stem 4)
   ; Avyayaf (wrap stem 3)
   ]
```

```
;
value poly_ii_decls decline =
  [ (Singular,
      [ decline Voc "i"
      ; decline Voc "iis" (* Bucknell senaanii.h Table 7 Deshpande p146 *)
      ; decline Nom "iis"
      ; decline Acc "yam"
      ; decline Ins "yaa"
      ; decline Dat "ye"
      ; decline Abl "yas"
      ; decline Gen "yas"
      ; decline Loc "yi"
      ; decline Loc "yaam" (* Bucknell senaanyaam Table 7 Deshpande p146 *)
      ])
  ; (Dual,
      [ decline Voc "yaa"
      ; decline Nom "yaa"
      ; decline Acc "yaa"
      ; decline Ins "iibhyaam"
      ; decline Dat "iibhyaam"
      ; decline Abl "iibhyaam"
      ; decline Gen "yos"
      ; decline Loc "yos"
      ])
  ; (Plural,
      [ decline Voc "yas"
      ; decline Nom "yas"
      ; decline Acc "yas"
      ; decline Ins "iibhis"
      ; decline Dat "iibhyas"
      ; decline Abl "iibhyas"
      ; decline Gen "iinaam"
      ; decline Loc "ii.su"
      ])
  ]
;
(* vedic forms g = Fem, rarely Mas (rathii) *)
value build_poly_ii g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
```

```
    [ Declined Noun g (poly_ii_decls decline)
    ; Bare Noun (wrap stem 4)
(* ; Bare Noun (wrap stem 3) eg kumaarimataa Pan6,3,42 *)
    ; Avyayaf (wrap stem 3)
    ]
;
value build_strii stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Fem
    [ (Singular,
          [ decline Voc "i"
          ; decline Nom "ii"
          ; decline Acc "iyam"
          ; decline Acc "iim"
          ; decline Ins "iyaa"
          ; decline Dat "iyai"
          ; decline Abl "iyaas"
          ; decline Gen "iyaas"
          ; decline Loc "iyaam"
          ])
    ; (Dual,
          [ decline Voc "iyau"
          ; decline Nom "iyau"
          ; decline Acc "iyau"
          ; decline Ins "iibhyaam"
          ; decline Dat "iibhyaam"
          ; decline Abl "iibhyaam"
          ; decline Gen "iyos"
          ; decline Loc "iyos"
          ])
    ; (Plural,
          [ decline Voc "iyas"
          ; decline Nom "iyas"
          ; decline Acc "iyas"
          ; decline Acc "iis"
          ; decline Ins "iibhis"
          ; decline Dat "iibhyas"
          ; decline Abl "iibhyas"
          ; decline Gen "iinaam"
```

```
          ; decline Loc "ii.su"
          ])
    ]
    ; Bare Noun (wrap stem 4)
    ; Avyayaf (wrap stem 3)
    ]
;
value build_fem_u stem trunc entry =
  let declines case suff = (case, fix stem suff)
  and declineg case suff = (case, fix [ 12 :: trunc ] suff)
  and declinel case suff = (case, fix [ 6 :: trunc ] suff)
  and declinau case = (case, wrap trunc 13) in
  enter entry (
    [ Declined Noun Fem
    [ (Singular,
          [ declineg Voc ""
          ; declines Nom "s"
          ; declines Acc "m"
          ; declines Ins "aa"
          ; declines Dat "ai"
          ; declineg Dat "e"
          ; declines Abl "aas"
          ; declineg Abl "s"
          ; declines Gen "aas"
          ; declineg Gen "s"
          ; declines Loc "aam"
          ; declinau Loc
          ])
    ; (Dual,
          [ declinel Voc ""
          ; declinel Nom ""
          ; declinel Acc ""
          ; declines Ins "bhyaam"
          ; declines Dat "bhyaam"
          ; declines Abl "bhyaam"
          ; declines Gen "os"
          ; declines Loc "os"
          ])
    ; (Plural,
          [ declineg Voc "as"
```

```
          ; declineg Nom "as"
          ; declinel Acc "s"
          ; declines Ins "bhis"
          ; declines Dat "bhyas"
          ; declines Abl "bhyas"
          ; declinel Gen "naam"
          ; declines Loc "su"
          ])
    ]
    ; Avyayaf (mirror stem)
    ] @ (if entry ="ku#2" ∨ entry ="go" then [] (* avoids overgeneration *)
          else [ Bare Noun (mirror stem) ]))
;
value build_fem_uu stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Fem
    [ (Singular,
          [ decline Voc "u"
          ; decline Nom "uus"
          ; decline Acc "uum"
          ; decline Ins "vaa"
          ; decline Dat "vai"
          ; decline Abl "vaas"
          ; decline Gen "vaas"
          ; decline Loc "vaam"
          ])
    ; (Dual,
          [ decline Voc "vau"
          ; decline Nom "vau"
          ; decline Acc "vau"
          ; decline Ins "uubhyaam"
          ; decline Dat "uubhyaam"
          ; decline Abl "uubhyaam"
          ; decline Gen "vos"
          ; decline Loc "vos"
          ])
    ; (Plural,
          [ decline Voc "vas"
          ; decline Nom "vas"
```

```
          ; decline Acc "uus"
          ; decline Ins "uubhis"
          ; decline Dat "uubhyas"
          ; decline Abl "uubhyas"
          ; decline Gen "uunaam"
          ; decline Loc "uu.su"
          ])
    ]
  ; Bare Noun (wrap stem 6)
  ; Bare Noun (wrap stem 5) (* Pan6,3,61 *)
  ; Avyayaf (wrap stem 5)
  ]
;
(* g = Fem, rarely Mas *)
value build_mono_uu g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun g
   [ (Singular,
          [ decline Voc "uus"
          ; decline Voc "u" (* alternative Renou §234 MW gram §126h Vopadeva *)
          ; decline Nom "uus"
          ; decline Acc "uvam"
          ; decline Ins "uvaa"
          ; decline Dat "uve"
          ; decline Dat "uvai"
          ; decline Abl "uvas"
          ; decline Abl "uvaas"
          ; decline Gen "uvas"
          ; decline Gen "uvaas"
          ; decline Loc "uvi"
          ; decline Loc "uvaam"
          ])
   ; (Dual,
          [ decline Voc "uvau"
          ; decline Nom "uvau"
          ; decline Acc "uvau"
          ; decline Ins "uubhyaam"
          ; decline Dat "uubhyaam"
          ; decline Abl "uubhyaam"
```

```
        ; decline Gen "uvos"
        ; decline Loc "uvos"
        ])
    ; (Plural,
        [ decline Voc "uvas"
        ; decline Nom "uvas"
        ; decline Acc "uvas"
        ; decline Ins "uubhis"
        ; decline Dat "uubhyas"
        ; decline Abl "uubhyas"
        ; decline Gen "uvaam"
        ; decline Gen "uunaam"
        ; decline Loc "uu.su"
        ])
    ]
    ; Bare Noun (wrap stem 6)
    ; Avyayaf (wrap stem 5)
    ]
;
value poly_uu_decls decline =
    [ (Singular,
        [ decline Voc "u"
        ; decline Nom "uus"
        ; decline Acc "vam"
        ; decline Ins "vaa"
        ; decline Dat "ve"
        ; decline Abl "vas"
        ; decline Gen "vas"
        ; decline Loc "vi"
        ])
    ; (Dual,
        [ decline Voc "vaa"
        ; decline Nom "vaa"
        ; decline Acc "vaa"
        ; decline Ins "uubhyaam"
        ; decline Dat "uubhyaam"
        ; decline Abl "uubhyaam"
        ; decline Gen "vos"
        ; decline Loc "vos"
        ])
```

```
    ; (Plural,
        [ decline Voc "vas"
        ; decline Nom "vas"
        ; decline Acc "vas"
        ; decline Ins "uubhis"
        ; decline Dat "uubhyas"
        ; decline Abl "uubhyas"
        ; decline Gen "uunaam"
        ; decline Loc "uu.su"
        ])
    ]
;
(* vedic forms g = Fem, very rarely Mas (praazuu) *)
value build_poly_uu g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun g (poly_uu_decls decline)
    ; Bare Noun (wrap stem 6)
    ; Avyayaf (wrap stem 5)
    ]
;
value build_fem_ri_v stem entry = (* vriddhi in strong cases *)
  let decline case suff = (case, fix stem suff)
  and bare = wrap stem 7 in
  enter entry
    [ Declined Noun Fem
    [ (Singular,
        [ decline Voc "ar"
        ; decline Nom "aa"
        ; decline Acc "aaram"
        ; decline Ins "raa"
        ; decline Dat "re"
        ; decline Abl "ur"
        ; decline Gen "ur"
        ; decline Loc "ari"
        ])
    ; (Dual,
        [ decline Voc "aarau"
        ; decline Nom "aarau"
        ; decline Acc "aarau"
```

```
        ; decline Ins ".rbhyaam"
        ; decline Dat ".rbhyaam"
        ; decline Abl ".rbhyaam"
        ; decline Gen "ros"
        ; decline Loc "ros"
        ])
   ; (Plural,
        [ decline Voc "aaras"
        ; decline Nom "aaras"
        ; decline Acc ".rrs"
        ; decline Ins ".rbhis"
        ; decline Dat ".rbhyas"
        ; decline Abl ".rbhyas"
        ; decline Gen ".rr.naam"
        ; decline Loc ".r.su"
        ])
   ]
   ; Bare Noun bare
   ; Avyayaf bare
   ]
;
value build_fem_ri_g stem entry = (* lien de parenté avec gu.na *)
  let decline case suff = (case, fix stem suff)
  and bare = wrap stem 7 in
  enter entry
   [ Declined Noun Fem
   [ (Singular,
        [ decline Voc "ar"
        ; decline Nom "aa"
        ; decline Acc "aram"
        ; decline Ins "raa"
        ; decline Dat "re"
        ; decline Abl "ur"
        ; decline Gen "ur"
        ; decline Loc "ari"
        ])
   ; (Dual,
        [ decline Voc "arau"
        ; decline Nom "arau"
        ; decline Acc "arau"
```

```
          ; decline Ins ".rbhyaam"
          ; decline Dat ".rbhyaam"
          ; decline Abl ".rbhyaam"
          ; decline Gen "ros"
          ; decline Loc "ros"
          ])
    ; (Plural,
          [ decline Voc "aras"
          ; decline Nom "aras"
          ; decline Acc ".rrs"
          ; decline Acc "aras" (* epics Whitney 373c *)
          ; decline Ins ".rbhis"
          ; decline Dat ".rbhyas"
          ; decline Abl ".rbhyas"
          ; decline Gen ".rr.naam"
          ; decline Loc ".r.su"
          ])
    ]
    ; Bare Noun bare
    ; Avyayaf bare
    ; Indecl Tas (fix stem ".rtas") (* maat.rtas *)
    ]
;
value build_fem_ir stem entry = (* gir *)
  let decline case suff = (case, fix stem suff)
  and short = fix stem "ir"
  and long = fix stem "iir" in
  enter entry
    [ Declined Noun Fem
    [ (Singular,
          [ decline Voc "iir"
          ; decline Nom "iir"
          ; decline Acc "iram"
          ; decline Ins "iraa"
          ; decline Dat "ire"
          ; decline Abl "iras"
          ; decline Gen "iras"
          ; decline Loc "iri"
          ])
    ; (Dual,
```

```
          [ decline Voc "irau"
          ; decline Nom "irau"
          ; decline Acc "irau"
          ; decline Ins "iirbhyaam"
          ; decline Dat "iirbhyaam"
          ; decline Abl "iirbhyaam"
          ; decline Gen "iros"
          ; decline Loc "iros"
          ])
    ; (Plural,
          [ decline Voc "iras"
          ; decline Nom "iras"
          ; decline Acc "iras"
          ; decline Ins "iirbhis"
          ; decline Dat "iirbhyas"
          ; decline Abl "iirbhyas"
          ; decline Gen "iraam"
          ; decline Loc "iir.su"
          ])
    ]
    ; Bare Noun short (* gir- *)
    ; Bare Noun long (* giir- *)
    ; Avyayaf short
    ]
;
(* Similar to preceding paradigm - for aazis fem et niraazis adj *)
value build_aazis g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun g
    [ (Singular,
          [ decline Voc "iis"
          ; decline Nom "iis"
          ; decline Acc "i.sam"
          ; decline Ins "i.saa"
          ; decline Dat "i.se"
          ; decline Abl "i.sas"
          ; decline Gen "i.sas"
          ; decline Loc "i.si"
          ])
```

```
    ; (Dual,
          [ decline Voc "i.sau"
          ; decline Nom "i.sau"
          ; decline Acc "i.sau"
          ; decline Ins "iirbhyaam"
          ; decline Dat "iirbhyaam"
          ; decline Abl "iirbhyaam"
          ; decline Gen "i.sos"
          ; decline Loc "i.sos"
          ])
    ; (Plural,
          [ decline Voc "i.sas"
          ; decline Nom "i.sas"
          ; decline Acc "i.sas"
          ; decline Ins "iirbhis"
          ; decline Dat "iirbhyas"
          ; decline Abl "iirbhyas"
          ; decline Gen "i.saam"
          ; decline Loc "ii.h.su"
          ; decline Loc "ii.s.su" (* necessary *)
          ])
    ]
    ; Bare Noun (fix stem "iir") (* aazis1- *)
    ; Bare Noun (fix stem "ii") (* aazis2- *)
    ; Avyayaf (fix stem "is")
    ]
;
value build_fem_ur stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Fem
    [ (Singular,
          [ decline Voc "uur"
          ; decline Nom "uur"
          ; decline Acc "uram"
          ; decline Ins "uraa"
          ; decline Dat "ure"
          ; decline Abl "uras"
          ; decline Gen "uras"
          ; decline Loc "uri"
```

```
          ])
    ; (Dual,
          [ decline Voc "urau"
          ; decline Nom "urau"
          ; decline Acc "urau"
          ; decline Ins "uurbhyaam"
          ; decline Dat "uurbhyaam"
          ; decline Abl "uurbhyaam"
          ; decline Gen "uros"
          ; decline Loc "uros"
          ])
    ; (Plural,
          [ decline Voc "uras"
          ; decline Nom "uras"
          ; decline Acc "uras"
          ; decline Ins "uurbhis"
          ; decline Dat "uurbhyas"
          ; decline Abl "uurbhyas"
          ; decline Gen "uraam"
          ; decline Loc "uur.su"
          ])
    ]
    ; Bare Noun (fix stem "uur") (* dhuur- *)
    ; Avyayaf (fix stem "ur")
    ]
;
(* This paradigm could be obtained by implementing Macdonell§59, see Phonetics.diphthong_split
and the code commented out in Int_sandhi *)
value build_rai g stem entry = (* stem = raa g = Mas or Fem (rare) *)
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun g
    [ (Singular,
          [ decline Voc "s"
          ; decline Nom "s"
          ; decline Acc "yam"
          ; decline Ins "yaa"
          ; decline Dat "ye"
          ; decline Abl "yas"
          ; decline Gen "yas"
```

```
          ; decline Loc "yi"
          ])
    ; (Dual,
          [ decline Voc "yau"
          ; decline Nom "yau"
          ; decline Acc "yau"
          ; decline Ins "bhyaam"
          ; decline Dat "bhyaam"
          ; decline Abl "bhyaam"
          ; decline Gen "yos"
          ; decline Loc "yos"
          ])
    ; (Plural,
          [ decline Voc "yas"
          ; decline Nom "yas"
          ; decline Acc "yas"
          ; decline Ins "bhis"
          ; decline Dat "bhyas"
          ; decline Abl "bhyas"
          ; decline Gen "yaam"
          ; decline Loc "su"
          ])
    ]
    ; Avyayaf (code "ri")
    ]
;
value build_e g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun g
    [ (Singular,
          [ decline Voc "es"
          ; decline Voc "e" (* Kale 33 *)
          ; decline Nom "es"
          ; decline Acc "am"
          ; decline Ins "ayaa"
          ; decline Dat "aye"
          ; decline Abl "es"
          ; decline Gen "es"
          ; decline Loc "ayi"
```

```
        ])
    ; (Dual,
        [ decline Voc "ayau"
        ; decline Nom "ayau"
        ; decline Acc "ayau"
        ; decline Ins "ebhyaam"
        ; decline Dat "ebhyaam"
        ; decline Abl "ebhyaam"
        ; decline Gen "ayos"
        ; decline Loc "ayos"
        ])
    ; (Plural,
        [ decline Voc "ayas"
        ; decline Nom "ayas"
        ; decline Acc "ayas"
        ; decline Ins "ebhis"
        ; decline Dat "ebhyas"
        ; decline Abl "ebhyas"
        ; decline Gen "ayaam"
        ; decline Loc "e.su"
        ])
    ]
    ; Bare Noun (fix stem "aya")
    ; Avyayaf (fix stem "i")
    ]
;
value build_o g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun g
    [ (Singular,
        [ decline Voc "aus"
        ; decline Nom "aus"
        ; decline Acc "aam"
        ; decline Ins "avaa"
        ; decline Dat "ave"
        ; decline Abl "os"
        ; decline Gen "os"
        ; decline Loc "avi"
        ])
```

```
    ; (Dual,
         [ decline Voc "aavau"
         ; decline Nom "aavau"
         ; decline Acc "aavau"
         ; decline Ins "obhyaam"
         ; decline Dat "obhyaam"
         ; decline Abl "obhyaam"
         ; decline Gen "avos"
         ; decline Loc "avos"
         ])
    ; (Plural,
         [ decline Voc "aavas"
         ; decline Nom "aavas"
         ; decline Acc "aas"
         ; decline Ins "obhis"
         ; decline Dat "obhyas"
         ; decline Abl "obhyas"
         ; decline Gen "avaam"
         ; decline Loc "o.su"
         ])
    ]
    ; Bare Noun ((mirror stem) @ (code "o")) (* go- *)
    ; Bare Noun ((mirror stem) @ (code "ava")) (* go -¿ gava- *)
    ; Avyayaf (fix stem "u") (* upagu *)
    ]
;
value build_div g stem entry = (* stem = "d" *)
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun g
    [ (Singular,
         [ decline Voc "yaus"
         ; decline Nom "yaus"
         ; decline Acc "ivam"
         ; decline Acc "yaam"
         ; decline Ins "ivaa"
         ; decline Dat "ive"
         ; decline Dat "yave"
         ; decline Abl "ivas"
         ; decline Abl "yos"
```

```
          ; decline Gen "ivas"
          ; decline Gen "yos"
          ; decline Loc "ivi"
          ; decline Loc "yavi"
          ])
    ; (Dual,
          [ decline Nom "yaavau"
          ; decline Nom "ivau" (* Renou *)
          ; decline Acc "yaavau"
          ; decline Acc "ivau" (* Renou *)
          ])
    ; (Plural,
          [ decline Voc "ivas"
          ; decline Nom "ivas"
          ; decline Nom "yaavas"
          ; decline Acc "ivas"
          ; decline Ins "yubhis"
          ; decline Dat "yubhyas"
          ; decline Abl "yubhyas"
          ; decline Gen "ivaam"
          ; decline Loc "yu.su"
          ])
    ]
    ; Avyayaf (fix stem "iv")
    ]
;
value build_diiv entry = (* diiv#2 *)
  let decline case form = (case, code form) in
  enter entry
    [ Declined Noun Fem
    [ (Singular,
          [ decline Acc "dyuvam"
          ; decline Ins "diivnaa" (* for pratidiivnaa (par l'adversaire) *)
          ; decline Dat "diive"
          ; decline Dat "dyuve"
          ; decline Loc "diivi"
          ])
    ] ]
;
value build_au g stem entry =
```

```
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Noun g
   [ (Singular,
         [ decline Voc "aus"
         ; decline Nom "aus"
         ; decline Acc "aavam"
         ; decline Ins "aavaa"
         ; decline Dat "aave"
         ; decline Abl "aavas"
         ; decline Gen "aavas"
         ; decline Loc "aavi"
         ])
   ; (Dual,
         [ decline Voc "aavau"
         ; decline Nom "aavau"
         ; decline Acc "aavau"
         ; decline Ins "aubhyaam"
         ; decline Dat "aubhyaam"
         ; decline Abl "aubhyaam"
         ; decline Gen "aavos"
         ; decline Loc "aavos"
         ])
   ; (Plural,
         [ decline Voc "aavas"
         ; decline Nom "aavas"
         ; decline Acc "aavas"
         ; decline Ins "aubhis"
         ; decline Dat "aubhyas"
         ; decline Abl "aubhyas"
         ; decline Gen "aavaam"
         ; decline Loc "au.su"
         ])
   ]
   ; Avyayaf (fix stem "u")
   ]
;
value build_ap entry =
  enter entry
   [ Declined Noun Fem
```

```
    [ (Plural,
          [ register Voc "aapas"
          ; register Nom "aapas"
          ; register Acc "apas"
          ; register Ins "adbhis"
          ; register Dat "adbhyas"
          ; register Abl "adbhyas"
          ; register Gen "apaam"
          ; register Loc "apsu"
          ])
    ]
    ; Bare Noun (code "ap")
    ; Avyayaf (code "apam")
    ]
;
```

(* Root word declension. Finalization ensures the initial aspiration by Phonetics.asp, in order to transform eg duk in dhuk (Whitney §155) *)

```
value build_root g stem entry =
  let decline case suff = (case, fix stem suff)
  and decline_nasalise case suff =
        let nstem = match stem with
          [ [ c :: r ] → if nasal c then stem else
                                   try [ c :: [ (homonasal c) :: r ]]
                                   with [ Failure _ → stem ]
          | _ → failwith "build_root"
          ] in (case, fix nstem suff)
  and declfin case suff =
        (* finalize_r for doubling of vowel in r roots Whitney §245b *)
        (case, fix (finalize_r stem) suff)
  and bare = mirror (finalize_r stem) in
  enter entry
    [ Declined Noun g
    [ (Singular,
          [ declfin Voc ""
          ; declfin Nom ""
          ; if g = Neu then declfin Acc "" else decline Acc "am"
          ; decline Ins "aa"
          ; decline Dat "e"
          ; decline Abl "as"
          ; decline Gen "as"
```

```
        ; decline Loc "i"
        ])
  ; (Dual,
        [ decline Voc (if g = Neu then "ii" else "au")
        ; decline Nom (if g = Neu then "ii" else "au")
        ; decline Acc (if g = Neu then "ii" else "au")
        ; declfin Ins "bhyaam"
        ; declfin Dat "bhyaam"
        ; declfin Abl "bhyaam"
        ; decline Gen "os"
        ; decline Loc "os"
        ])
  ; (Plural,
        [ if g = Neu then decline_nasalise Voc "i" else decline Voc "as"
        ; if g = Neu then decline_nasalise Nom "i" else decline Nom "as"
        ; if g = Neu then decline_nasalise Acc "i" else decline Acc "as"
  (* Voc Nom Acc Neu ought to have nasal : v.rnti Whitney§389c p. 145 *)
  (* Acc. vaacas with accent on aa or on a Whitney§391 p. 147 *)
        ; declfin Ins "bhis"
        ; declfin Dat "bhyas"
        ; declfin Abl "bhyas"
        ; decline Gen "aam"
        ; declfin Loc "su"
         (* viz2 -¿ vi.tsu but also véd. vik.su Whitney§218a compute_extra *)
        ])
  ]
  ; Bare Noun bare (* thus hutabhuj -¿ hutabhuk+dik -¿ ...gdik *)
  ; Avyayaf bare
  ]
;
value build_root_m g trunc stem entry = (* Kale§107 prazaam *)
 let decline case suff = (case, fix stem suff)
 and declcon case suff = (case, fix [ 36 (* n *) :: trunc ] suff) in
 enter entry
  [ Declined Noun g
  [ (Singular,
        [ declcon Voc ""
        ; declcon Nom ""
        ; if g = Neu then declcon Acc "" else decline Acc "am"
        ; decline Ins "aa"
```

```
            ; decline Dat "e"
            ; decline Abl "as"
            ; decline Gen "as"
            ; decline Loc "i"
            ])
    ; (Dual,
            [ decline Voc (if g = Neu then "ii" else "au")
            ; decline Nom (if g = Neu then "ii" else "au")
            ; decline Acc (if g = Neu then "ii" else "au")
            ; declcon Ins "bhyaam"
            ; declcon Dat "bhyaam"
            ; declcon Abl "bhyaam"
            ; decline Gen "os"
            ; decline Loc "os"
            ])
    ; (Plural,
            [ decline Voc (if g = Neu then "i" else "as")
            ; decline Nom (if g = Neu then "i" else "as")
            ; decline Acc (if g = Neu then "i" else "as")
            ; declcon Ins "bhis"
            ; declcon Dat "bhyas"
            ; declcon Abl "bhyas"
            ; decline Gen "aam"
            ; declcon Loc "su"
            ])
    ]
    ]
;
value build_archaic_yuj stem (* yu nj remnant nasal Kale§97 *) g entry =
  let decline case suff = (case, fix stem suff)
  and declfinal case = (case, [ 42; 5; 21 (* yuf *) ]) in (* Whitney§386 *)
  enter entry
    [ Declined Noun g
    [ (Singular,
            [ declfinal Voc
            ; declfinal Nom
            ; if g = Neu then declfinal Acc else decline Acc "am"
            ])
    ; (Dual,
            [ decline Voc "au" (* Kale§97 but Whitney§386 "aa" ? *)
```

```
        ; decline Nom "au"
        ; decline Acc "au"
        ])
  ; (Plural,
        [ decline Voc "as"
        ; decline Nom "as"
        ])
  ]
  ]
;
(* Root words opt. substitutes in weak cases P{6,1,63} Whitney§397 *)
value build_root_weak g stem entry =
 let decline case suff = (case, fix stem suff)
 and bare = mirror (finalize stem) in
 enter entry (* strong stem entry paada danta etc. *)
  [ Declined Noun g
  [ (Singular,
        [ decline Ins "aa"
        ; decline Dat "e"
        ; decline Abl "as"
        ; decline Gen "as"
        ; decline Loc "i"
        ])
  ; (Dual,
        [ decline Ins "bhyaam"
        ; decline Dat "bhyaam"
        ; decline Abl "bhyaam"
        ; decline Gen "os"
        ; decline Loc "os"
        ])
  ; (Plural,
        [ decline Acc "as"
        ; decline Ins "bhis"
        ; decline Dat "bhyas"
        ; decline Abl "bhyas"
        ; decline Gen "aam"
        ; decline Loc "su"
        ])
  ]
  ; Bare Noun bare
```

```
    ; Avyayaf bare
    ]
;
value build_pad g stem entry = (* for catu.spad and other -pad compounds *)
  let decline case form = (case, fix stem form)
  and bare = fix stem "pat" in
  enter entry
   [ Declined Noun g
   [ (Singular,
          [ decline Nom "paat"
          ; decline Voc "paat"
          ; decline Acc "paadam"
          ; decline Ins "padaa"
          ; decline Dat "pade"
          ; decline Abl "padas"
          ; decline Gen "padas"
          ; decline Loc "padi"
          ] @ if g = Fem then
          [ decline Nom "padii" ] else [])
   ; (Dual,
          [ decline Nom (if g = Neu then "paadii" else "paadau")
          ; decline Voc (if g = Neu then "paadii" else "paadau")
          ; decline Acc (if g = Neu then "paadii" else "paadau")
          ; decline Ins "paadbhyaam"
          ; decline Dat "paadbhyaam"
          ; decline Abl "paadbhyaam"
          ; decline Gen "paados"
          ; decline Loc "paados"
          ])
   ; (Plural,
          [ decline Nom "paadas"
          ; decline Voc "paadas"
          ; decline Acc "paadas"
          ; decline Ins "paadbhis"
          ; decline Dat "paadbhyas"
          ; decline Abl "paadbhyas"
          ; decline Gen "paadaam"
          ; decline Loc "paatsu"
          ])
   ]
```

```
    ; Bare Noun bare
    ; Avyayaf bare
    ]
;
value build_sap g st entry = (* MW saap in strong cases *)
  let decline case suff = (case, fix [ 37 :: [ 1 :: [ 48 :: st ] ] ] suff)
  and declinestr case suff = (case, fix [ 37 :: [ 2 :: [ 48 :: st ] ] ] suff) in
  enter entry
    [ Declined Noun g
    [ (Singular,
          [ decline Voc ""
          ; declinestr Nom ""
          ; declinestr Acc "am"
          ; decline Ins "aa"
          ; decline Dat "e"
          ; decline Abl "as"
          ; decline Gen "as"
          ; decline Loc "i"
          ])
    ; (Dual,
          [ decline Voc (if g = Neu then "ii" else "au")
          ; declinestr Nom (if g = Neu then "ii" else "au")
          ; declinestr Acc (if g = Neu then "ii" else "au")
          ; decline Ins "bhyaam"
          ; decline Dat "bhyaam"
          ; decline Abl "bhyaam"
          ; decline Gen "os"
          ; decline Loc "os"
          ])
    ; (Plural,
          [ decline Voc (if g = Neu then "i" else "as")
          ; declinestr Nom (if g = Neu then "i" else "as")
          ; decline Acc (if g = Neu then "i" else "as")
          ; decline Ins "bhis"
          ; decline Dat "bhyas"
          ; decline Abl "bhyas"
          ; decline Gen "aam"
          ; decline Loc "su"
          ])
    ]
```

```
  ]
;
value build_dam entry = (* vedic *)
  let decline case form = (case, code form) in
  enter entry
   [ Declined Noun Mas (* arbitrary *)
   [ (Singular,
        [ decline Gen "dan" ])
   ; (Plural,
        [ decline Gen "damaam" ])
   ]
   ; Bare Noun (revcode "dam")
   ]
;
value build_upaanah trunc stem entry = (* Kale§101 trunc = mirror(upaana) *)
  let bare = [ 32 (* t *) :: trunc ] (* upaanat *) in
  let declineh case suff = (case, fix stem suff)
  and declinet case suff = (case, fix bare suff) in
  enter entry
   [ Declined Noun Fem
   [ (Singular,
        [ declinet Voc ""
        ; declinet Nom ""
        ; declineh Acc "am"
        ; declineh Ins "aa"
        ; declineh Dat "e"
        ; declineh Abl "as"
        ; declineh Gen "as"
        ; declineh Loc "i"
        ])
   ; (Dual,
        [ declineh Voc "au"
        ; declineh Nom "au"
        ; declineh Acc "au"
        ; declinet Ins "bhyaam"
        ; declinet Dat "bhyaam"
        ; declinet Abl "bhyaam"
        ; declineh Gen "os"
        ; declineh Loc "os"
        ])
```

```
  ; (Plural,
       [ declineh Voc "as"
       ; declineh Nom "as"
       ; declineh Acc "as"
       ; declinet Ins "bhis"
       ; declinet Dat "bhyas"
       ; declinet Abl "bhyas"
       ; declineh Gen "aam"
       ; declinet Loc "su"
       ])
  ]
  ; Bare Noun (mirror bare)
  ]
;
(* reduplicated ppr of class 3 verbs or intensives: no nasal in strong stem *)
(* should be replaced by proper tag, rather than matching stem *)
value is_redup = fun (* reduplicating roots, possibly with preverb *)
    [ [ 41 :: [ 3 :: [ 41 :: r ] ] ] when r = revstem "raz"
        → False (* razmimat protected from compounds of mimat *)
    | [ 34 :: [ 1 :: [ 34 :: _ ] ] ] (* daa#1 -¿ dadat *)
    | [ 35 :: [ 1 :: [ 34 :: _ ] ] ] (* dhaa#1 -¿ dadhat *)
    | [ 41 :: [ 3 :: [ 41 :: _ ] ] ] (* maa#1 -¿ mimat *)
    | [ 42 :: [ 5 :: [ 42 :: _ ] ] ] (* yu#2 -¿ yuyat *)
    | [ 43 :: [ 19 :: [ 2 :: [ 24 :: _ ] ] ] ] (* g.r int -¿ jaagrat *)
    | [ 43 :: [ 20 :: [ 3 :: [ 24 :: _ ] ] ] ] (* gh.r -¿ jighrat *)
    | [ 43 :: [ 37 :: [ 3 :: [ 37 :: _ ] ] ] ] (* p.r#1 -¿ piprat *)
    | [ 43 :: [ 40 :: [ 3 :: [ 39 :: _ ] ] ] ] (* bh.r -¿ bibhrat *)
    | [ 45 :: [ 49 :: [ 5 :: [ 24 :: _ ] ] ] ] (* hu -¿ juhvat *)
    | [ 46 :: [ 3 :: [ 46 :: _ ] ] ] (* zaa -¿ zizat *)
    | [ 48 :: [ 3 :: [ 48 :: _ ] ] ] (* s.r -¿ sisrat *)
    | [ 49 :: [ 1 :: [ 24 :: _ ] ] ] (* haa#1 -¿ jahat *)
(* — 49 :: [ 3 :: [ 24 :: _ ] ] (* haa#? -¿ jihat *) ? *)
    | [ 49 :: [ 12 :: [ 24 :: _ ] ] ] (* hu int. -¿ johvat *)
    | [ 41 :: [ 1 :: [ 43 :: [ 17 :: [ 21 :: [ 1 :: [ 22 :: _ ] ] ] ] ] ] ]
          (* kram int. -¿ cafkramat *)
    | [ 34 :: [ 1 :: [ 45 :: [ 2 :: [ 45 :: _ ] ] ] ] ](* vad int. -¿ vaavadat *)
        → True
(* Whitney says add: cak.sat daazat daasat zaasat sazcat dhak.sat vaaghat *)
    | _ → False
    ]
```

```
;
value build_auduloma g stem pstem entry = (* au.duloma Kale 26 *)
  let decline case suff = (case, fix stem suff)
  and declinep case suff = (case, fix pstem suff) in
  enter entry
   [ Declined Noun g
   [ (Singular,
         [ decline Voc "e"
         ; decline Nom "is"
         ; decline Acc "im"
         ; decline Ins "inaa"
         ; decline Dat "aye"
         ; decline Abl "es"
         ; decline Gen "es"
         ; decline Loc "au"
         ])
   ; (Dual,
         [ decline Voc "ii"
         ; decline Nom "ii"
         ; decline Acc "ii"
         ; decline Ins "ibhyaam"
         ; decline Dat "ibhyaam"
         ; decline Abl "ibhyaam"
         ; decline Gen "yos"
         ; decline Loc "yos"
         ])
   ; (Plural,
         [ declinep Voc "aas"
         ; declinep Nom "aas"
         ; declinep Acc "aan"
         ; declinep Ins "ais"
         ; declinep Dat "ebhyas"
         ; declinep Abl "ebhyas"
         ; declinep Gen "aanaam"
         ; declinep Loc "esu"
         ])
   ] ]
;
```

Pronouns

```
value build_sa_tad g stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry (
   [ Declined Pron g
   [ (Singular, let l =
         [ decline Nom (if g = Mas then "sas" (* gives e.sa.h for etad *)
                                    else "tat") (* final *)
         ; decline Acc (if g = Mas then "tam" else "tat")
         ; decline Ins "tena"
         ; decline Dat "tasmai"
         ; decline Abl "tasmaat"
         ; decline Abl "tatas"
         ; decline Gen "tasya"
         ; decline Loc "tasmin"
         ] in if g = Mas then
         [ decline Nom "sa" :: l ] (* usable before consonants, see Dispatcher *)
              else l)
   ; (Dual,
         [ decline Nom (if g = Mas then "tau" else "te")
         ; decline Acc (if g = Mas then "tau" else "te")
         ; decline Ins "taabhyaam"
         ; decline Dat "taabhyaam"
         ; decline Abl "taabhyaam"
         ; decline Abl "tatas"
         ; decline Gen "tayos"
         ; decline Loc "tayos"
         ])
   ; (Plural,
         [ decline Nom (if g = Mas then "te" else "taani")
         ; decline Acc (if g = Mas then "taan" else "taani")
         ; decline Ins "tais"
         ; decline Dat "tebhyas"
         ; decline Abl "tebhyas"
         ; decline Abl "tatas"
         ; decline Gen "te.saam"
         ; decline Loc "te.su"
         ])
   ] ] @ (if g = Neu ∧ stem = [ 10 ] then [ Bare Pron (code "etat") ]
            else []))
;
```

```
value build_sya_tyad g entry = (* Vedic Whitney §499a *)
  let decline case form = (case, code form) in
  enter entry
   [ Declined Pron g
   [ (Singular, let l =
          [ decline Nom (if g = Mas then "syas" else "tyat")
          ; decline Acc (if g = Mas then "tyam" else "tyat")
          ; decline Ins "tyena"
          ; decline Dat "tyasmai"
          ; decline Abl "tyasmaat"
          ; decline Abl "tyatas"
          ; decline Gen "tyasya"
          ; decline Loc "tyasmin"
          ] in if g = Mas then
          [ decline Nom "sya" :: l ]
                              else l)
   ; (Dual,
          [ decline Nom (if g = Mas then "tyau" else "tye")
          ; decline Acc (if g = Mas then "tyau" else "tye")
          ; decline Ins "tyaabhyaam"
          ; decline Dat "tyaabhyaam"
          ; decline Abl "tyaabhyaam"
          ; decline Abl "tyatas"
          ; decline Gen "tyayos"
          ; decline Loc "tyayos"
          ])
   ; (Plural,
          [ decline Nom (if g = Mas then "tye" else "tyaani")
          ; decline Acc (if g = Mas then "tyaan" else "tyaani")
          ; decline Ins "tyais"
          ; decline Dat "tyebhyas"
          ; decline Abl "tyebhyas"
          ; decline Abl "tyatas"
          ; decline Gen "tye.saam"
          ; decline Loc "tye.su"
          ])
   ] ]
;
```

(* Pronominal stems (mirror+lopa) of pronouns usable as nominals. When used as pronouns, they denote the relative position. *)

```
(* Bhat: puurva, para, avara, dak.si.na, uttara, apara, adhara, sva, antara: when used
pronominally use optionally pronominal endings. Missing from his list: aneka, pazcima,
nema, ubhaya, sarva, vizva *)
value pseudo_nominal_basis = fun
  [ [ 17; 10; 36; 1 ] (* aneka *) (* possibly also eka, anya ? *)
  | [ 31; 3; 47; 17; 1; 34 ] (* dak.si.na *)
  | [ 41; 3; 22; 46; 1; 37 ] (* pazcima *)
  | [ 41; 10; 36 ] (* nema Whitney§525c *)
  | [ 42; 1; 40; 5 ] (* ubhaya *)
  | [ 43; 1; 32; 32; 5 ] (* uttara *)
  | [ 43; 1; 32; 36; 1 ] (* antara *)
  | [ 43; 1; 35; 1 ] (* adhara *)
  | [ 43; 1; 37 ] (* para *)
  | [ 43; 1; 37; 1 ] (* apara *)
  | [ 43; 1; 45; 1 ] (* avara *)
  | [ 45; 43; 1; 48 ] (* sarva *)
  | [ 45; 43; 6; 37 ] (* puurva Whitney§524 *)
(* NB ga.na puurva, paraavara, dak.si.na, uttara, apara, adhara -: paraavara +: aneka,
pazcima, nema, ubhaya, antara, para, avara, sarva, vizva, sva *)
  | [ 45; 46; 3; 45 ] (* vizva *)
  | [ 45; 48 ] (* sva *) → True
  | _ → False
  ]
;
value build_pron_a g stem entry = (* g=Mas ou g=Neu *)
  let pseudo_nominal = pseudo_nominal_basis stem
  and neu_nom_acc = match stem with
      [ [ 17 ] → (* kim *) "im"
      | [ 42 ] (* yad *)
      | [ 43; 1; 32; 1; 17 ] (* katara *)
      | [ 41; 1; 32; 1; 17 ] (* katama *)
      | [ 43; 1; 32; 3 ] (* itara *)
      | [ 42; 36; 1 ] (* anya *)
      | [ 43; 1; 32; 1; 42; 36; 1 ] (* anyatara *)
      | [ 45; 32 ] (* tva *)→ "at" (* Whitney§523 *)
      | _ → (* eka, ekatara, vizva, sva, sarva, ... *) "am"
      ] in
  let decline case suff = (case, fix stem suff)
  and phase = if pseudo_nominal then Noun else Pron in
  enter entry (
```

```
[ Declined phase g
[ (Singular, let l =
      [ decline Nom (if g = Mas then "as" else neu_nom_acc)
      ; decline Acc (if g = Mas then "am" else neu_nom_acc)
      ; decline Ins "ena"
      ; decline Dat "asmai"
      ; decline Abl "asmaat"
      ; decline Gen "asya"
      ; decline Loc "asmin"
      ] in if pseudo_nominal then
      [ decline Abl "aat" :: [ decline Loc "e" ::
      [ decline Voc "a" :: l ] ] ] else l)
; (Dual, if entry = "ubhaya" (* no dual - dubious *) then []
         else let l =
      [ decline Nom (if g = Mas then "au" else "e")
      ; decline Acc (if g = Mas then "au" else "e")
      ; decline Ins "aabhyaam"
      ; decline Dat "aabhyaam"
      ; decline Abl "aabhyaam"
      ; decline Gen "ayos"
      ; decline Loc "ayos"
      ] in if pseudo_nominal then
      [ decline Voc (if g = Mas then "au" else "e") :: l ] else l)
; (Plural, let l =
      [ decline Nom (if g = Mas then "e" else "aani")
      ; decline Acc (if g = Mas then "aan" else "aani")
      ; decline Ins "ais"
      ; decline Dat "ebhyas"
      ; decline Abl "ebhyas"
      ; decline Gen "e.saam"
      ; decline Loc "e.su"
      ] in if pseudo_nominal then
            if g = Mas then [ decline Nom "aas" :: [ decline Voc "aas" :: l ] ]
            else (* g=Neu *) [ decline Voc "aani" :: l ]
          else l)
] ] @ (if g = Neu then
          let iic = match stem with
                    [ [ 17 ] (* kim *) → code "kim"
                    | [ 42 ] (* yad *) → code "yat"
                    | [ 42; 36; 1 ] (* anyad *) → code "anyat"
```

```
                              | _ → mirror [ 1 :: stem ]
                              ] in
                [ Bare phase iic ]
           else if g = Mas ∧ stem = [ 42; 36; 1 ] (* anya *)
                then [ Bare phase (code "anya") ] (* optional anya- *)
           else if pseudo_nominal ∧ g = Mas then
                   [ Avyayaf (fix stem "am"); Avyayaf (fix stem "aat")
                   ; Indecl Tas (fix stem "atas")
                   ]
           else [])
       @ (if g = Mas then match entry with
                              [ "eka" → [ Cvi (code "ekii") ]
                              | "sva" → [ Cvi (code "svii") ]
                              | _ → []
                              ]
           else [] ))
;
value build_saa stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Pron Fem
   [ (Singular,
         [ decline Nom "saa"
         ; decline Acc "taam"
         ; decline Ins "tayaa"
         ; decline Dat "tasyai"
         ; decline Abl "tasyaas"
         ; decline Abl "tatas"
         ; decline Gen "tasyaas"
         ; decline Loc "tasyaam"
         ])
   ; (Dual,
         [ decline Nom "te"
         ; decline Acc "te"
         ; decline Ins "taabhyaam"
         ; decline Dat "taabhyaam"
         ; decline Abl "taabhyaam"
         ; decline Abl "tatas"
         ; decline Gen "tayos"
         ; decline Loc "tayos"
```

```
        ])
   ; (Plural,
        [ decline Nom "taas"
        ; decline Acc "taas"
        ; decline Ins "taabhis"
        ; decline Dat "taabhyas"
        ; decline Abl "taabhyas"
        ; decline Abl "tatas"
        ; decline Gen "taasaam"
        ; decline Loc "taasu"
        ])
   ] ]
;
value build_syaa stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry
   [ Declined Pron Fem
   [ (Singular,
        [ decline Nom "syaa"
        ; decline Acc "tyaam"
        ; decline Ins "tyayaa"
        ; decline Dat "tyasyai"
        ; decline Abl "tyasyaas"
        ; decline Abl "tyatyas"
        ; decline Gen "tyasyaas"
        ; decline Loc "tyasyaam"
        ])
   ; (Dual,
        [ decline Nom "tye"
        ; decline Acc "tye"
        ; decline Ins "tyaabhyaam"
        ; decline Dat "tyaabhyaam"
        ; decline Abl "tyaabhyaam"
        ; decline Abl "tyatyas"
        ; decline Gen "tyayos"
        ; decline Loc "tyayos"
        ])
   ; (Plural,
        [ decline Nom "tyaas"
        ; decline Acc "tyaas"
```

```
          ; decline Ins "tyaabhis"
          ; decline Dat "tyaabhyas"
          ; decline Abl "tyaabhyas"
          ; decline Abl "tyatas"
          ; decline Gen "tyaasaam"
          ; decline Loc "tyaasu"
          ])
    ] ]
;
value build_pron_aa stem entry =
  let pseudo_nominal = pseudo_nominal_basis stem in
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Pron Fem
    [ (Singular, let l =
          [ decline Nom "aa"
          ; decline Acc "aam"
          ; decline Ins "ayaa"
          ; decline Dat "asyai"
          ; decline Abl "asyaas"
          ; decline Gen "asyaas"
          ; decline Loc "asyaam"
          ] in if pseudo_nominal then
          [ decline Voc "e" :: l ] else l)
    ; (Dual, let l =
          [ decline Nom "e"
          ; decline Acc "e"
          ; decline Ins "aabhyaam"
          ; decline Dat "aabhyaam"
          ; decline Abl "aabhyaam"
          ; decline Gen "ayos"
          ; decline Loc "ayos"
          ] in if pseudo_nominal then
          [ decline Voc "e" :: l ] else l)
    ; (Plural, let l =
          [ decline Nom "aas"
          ; decline Acc "aas"
          ; decline Ins "aabhis"
          ; decline Dat "aabhyas"
          ; decline Abl "aabhyas"
```

```
            ; decline Gen "aasaam"
            ; decline Loc "aasu"
            ] in if pseudo_nominal then
            [ decline Voc "aas" :: l ] else l)
    ] ]
;
value build_ayam_idam g = (* g=Mas or Neu *)
  enter "idam"
    [ Declined Pron g
    [ (Singular,
            [ register Nom (if g = Mas then "ayam" else "idam")
            ; register Acc (if g = Mas then "imam" else "idam")
            ; register Ins "anena"
            ; register Dat "asmai" (* also "atas" *)
            ; register Abl "asmaat"
            ; register Gen "asya"
            ; register Loc "asmin"
            ])
    ; (Dual,
            [ register Nom (if g = Mas then "imau" else "ime")
            ; register Acc (if g = Mas then "imau" else "ime")
            ; register Ins "aabhyaam"
            ; register Dat "aabhyaam"
            ; register Abl "aabhyaam"
            ; register Gen "anayos"
            ; register Loc "anayos"
            ])
    ; (Plural,
            [ register Nom (if g = Mas then "ime" else "imaani")
            ; register Acc (if g = Mas then "imaan" else "imaani")
            ; register Ins "ebhis"
            ; register Dat "ebhyas"
            ; register Abl "ebhyas"
            ; register Gen "e.saam"
            ; register Loc "e.su"
            ])
    ] ]
;
value build_iyam () =
  enter "idam"
```

```
  [ Declined Pron Fem
  [ (Singular,
       [ register Nom "iyam"
       ; register Acc "imaam"
       ; register Ins "anayaa"
       ; register Dat "asyai"
       ; register Abl "asyaas"
       ; register Gen "asyaas"
       ; register Loc "asyaam"
       ])
  ; (Dual,
       [ register Nom "ime"
       ; register Acc "ime"
       ; register Ins "aabhyaam"
       ; register Dat "aabhyaam"
       ; register Abl "aabhyaam"
       ; register Gen "anayos"
       ; register Loc "anayos"
       ])
  ; (Plural,
       [ register Nom "imaas"
       ; register Acc "imaas"
       ; register Ins "aabhis"
       ; register Dat "aabhyas"
       ; register Abl "aabhyas"
       ; register Gen "aasaam"
       ; register Loc "aasu"
       ])
  ] ]
;
value build_asau_adas g =
 enter "adas"
  [ Declined Pron g
  [ (Singular, let accu =
       [ register Nom (if g = Mas then "asau" else "adas")
       ; register Acc (if g = Mas then "amum" else "adas")
       ; register Ins "amunaa"
       ; register Dat "amu.smai"
       ; register Abl "amu.smaat"
       ; register Gen "amu.sya"
```

```
          ; register Loc "amu.smin"
          ] in if g = Mas then [ register Nom "asakau" :: accu ]
                                  (* Pan7,2,107 with yaka.h/yakaa *)
               else accu)
    ; (Dual,
          [ register Nom "amuu"
          ; register Acc "amuu"
          ; register Ins "amuubhyaam"
          ; register Dat "amuubhyaam"
          ; register Abl "amuubhyaam"
          ; register Gen "amuyos"
          ; register Loc "amuyos"
          ])
    ; (Plural,
          [ register Nom (if g = Mas then "amii" else "amuuni")
          ; register Acc (if g = Mas then "amuun" else "amuuni")
          ; register Ins "amiibhis"
          ; register Dat "amiibhyas"
          ; register Abl "amiibhyas"
          ; register Gen "amii.saam"
          ; register Loc "amii.su"
          ])
    ] ]
;
value build_asau_f () =
  enter "adas"
    [ Declined Pron Fem
    [ (Singular,
          [ register Nom "asau"
          ; register Nom "asakau" (* Pan7,2,107 with yaka.h/yakaa *)
          ; register Acc "amuum"
          ; register Ins "amuyaa"
          ; register Dat "amu.syai"
          ; register Abl "amu.syaas"
          ; register Gen "amu.syaas"
          ; register Loc "amu.syaam"
          ])
    ; (Dual,
          [ register Nom "amuu"
          ; register Acc "amuu"
```

```
        ; register Ins "amuubhyaam"
        ; register Dat "amuubhyaam"
        ; register Abl "amuubhyaam"
        ; register Gen "amuyos"
        ; register Loc "amuyos"
        ])
  ; (Plural,
        [ register Nom "amuus"
        ; register Acc "amuus"
        ; register Ins "amuubhis"
        ; register Dat "amuubhyas"
        ; register Abl "amuubhyas"
        ; register Gen "amuu.saam"
        ; register Loc "amuu.su"
        ])
  ] ]
;
value build_ena g entry =
 enter "idam" (* Whitney§500 *)
  [ Declined Pron g
  [ (Singular,
   (* No nominative - anaphoric pronoun - in non accented position *)
        [ register Acc (match g with
            [ Mas → "enam"
            | Neu → "enat"
            | Fem → "enaam"
            | _ → raise (Control.Anomaly "Nouns")
            ])
        ; register Ins (match g with
            [ Mas → "enena"
            | Neu → "enena"
            | Fem → "enayaa"
            | _ → raise (Control.Anomaly "Nouns")
            ])
        ])
  ; (Dual,
        [ register Acc (match g with
            [ Mas → "enau"
            | Neu → "ene"
            | Fem → "ene"
```

```
            | _ → raise (Control.Anomaly "Nouns")
            ])
        ; register Gen "enayos"
        ; register Loc "enayos"
        ])
  ; (Plural,
        [ register Acc (match g with
            [ Mas → "enaan"
            | Neu → "enaani"
            | Fem → "enaas"
            | _ → raise (Control.Anomaly "Nouns")
            ])
        ])
  ] ]
;
value build_aham () =
  let decline case form = (case, code form) in
  enter "asmad" (* Paninian entry *)
  [ Declined Pron (Deictic Speaker)
  [ (Singular,
        [ decline Nom "aham"
        ; decline Acc "maam"
        ; decline Acc "maa" (* encl *)
        ; decline Ins "mayaa"
        ; decline Dat "mahyam"
        ; decline Dat "me" (* encl *)
        ; decline Abl "mat"
        ; decline Abl "mattas"
        ; decline Gen "mama"
        ; decline Gen "me" (* encl *)
        ; decline Loc "mayi"
        ])
  ; (Dual,
        [ decline Nom "aavaam" (* Vedic "aavam" P{7.2.88} Burrow p267 *)
        ; decline Acc "aavaam"
        ; decline Acc "nau" (* encl *)
        ; decline Ins "aavaabhyaam"
        ; decline Dat "aavaabhyaam"
        ; decline Dat "nau" (* encl *)
        ; decline Abl "aavaabhyaam"
```

```
        ; decline Gen "aavayos"
        ; decline Gen "nau" (* encl *)
        ; decline Loc "aavayos"
        ])
   ; (Plural,
        [ decline Nom "vayam"
        ; decline Acc "asmaan"
        ; decline Acc "nas" (* encl *)
        ; decline Ins "asmaabhis"
        ; decline Dat "asmabhyam"
        ; decline Dat "nas" (* encl *)
        ; decline Abl "asmat"
        ; decline Abl "asmattas"
        ; decline Gen "asmaakam"
        ; decline Gen "nas" (* encl *)
        ; decline Loc "asmaasu"
        ])
   ]
   ; Bare Pron (code "aham")
   ; Bare Pron (code "mat") (* P{7,2,98} when meaning is singular *)
   ; Bare Pron (code "asmat") (* P{7,2,98} when meaning is plural *)
   ]
;
value build_tvad () =
  let decline case form = (case, code form) in
  enter "yu.smad" (* Paninian entry *)
   [ Declined Pron (Deictic Listener)
   [ (Singular,
        [ decline Nom "tvam"
        ; decline Acc "tvaam"
        ; decline Acc "tvaa" (* encl *)
        ; decline Ins "tvayaa"
        ; decline Dat "tubhyam"
        ; decline Dat "te" (* encl *)
        ; decline Abl "tvat"
        ; decline Abl "tvattas"
        ; decline Gen "tava"
        ; decline Gen "te" (* encl *)
        ; decline Loc "tvayi"
        ])
```

```
    ; (Dual,
        [ decline Nom "yuvaam" (* Vedic "yuvam" P{7.2.88} Burrow p267 *)
        ; decline Acc "yuvaam"
        ; decline Acc "vaam" (* encl *)
        ; decline Ins "yuvaabhyaam"
        ; decline Dat "yuvaabhyaam"
        ; decline Dat "vaam" (* encl *)
        ; decline Abl "yuvaabhyaam"
        ; decline Gen "yuvayos"
        ; decline Gen "vaam" (* encl *)
        ; decline Loc "yuvayos"
        ])
    ; (Plural,
        [ decline Nom "yuuyam"
        ; decline Acc "yu.smaan"
        ; decline Acc "vas" (* encl *)
        ; decline Ins "yu.smaabhis"
        ; decline Dat "yu.smabhyam"
        ; decline Dat "vas" (* encl *)
        ; decline Abl "yu.smat"
        ; decline Abl "yu.smattas"
        ; decline Gen "yu.smaakam"
        ; decline Gen "vas" (* encl *)
        ; decline Loc "yu.smaasu"
        ])
    ]
    ; Bare Pron (code "tvad") (* P{7,2,98} when meaning is singular *)
    ; Bare Pron (code "yu.smat") (* P{7,2,98} when meaning is plural *)
    ]
;
(* Numerals *)
value build_dvi entry =
  let stem = revcode "dv" in
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun Mas
    [ (Dual,
        [ decline Voc "au"
        ; decline Nom "au"
        ; decline Acc "au"
```

```
          ; decline Ins "aabhyaam"
          ; decline Dat "aabhyaam"
          ; decline Abl "aabhyaam"
          ; decline Gen "ayos"
          ; decline Loc "ayos"
          ])
    ]
    ; Declined Noun Neu
   [ (Dual,
          [ decline Voc "e"
          ; decline Nom "e"
          ; decline Acc "e"
          ; decline Ins "aabhyaam"
          ; decline Dat "aabhyaam"
          ; decline Abl "aabhyaam"
          ; decline Gen "ayos"
          ; decline Loc "ayos"
          ])
    ]
    ; Declined Noun Fem
   [ (Dual,
          [ decline Voc "e"
          ; decline Nom "e"
          ; decline Acc "e"
          ; decline Ins "aabhyaam"
          ; decline Dat "aabhyaam"
          ; decline Abl "aabhyaam"
          ; decline Gen "ayos"
          ; decline Loc "ayos"
          ])
    ]
    ; Bare Noun (code "dvaa")
    ; Bare Noun (code "dvi")
    ]
;
value build_tri entry =
   let decline case suff =
       (case, fix (revcode "tr") suff)
   and declinf case suff =
       (case, fix (revcode "tis") suff) in
```

```
enter entry
 [ Declined Noun Mas
 [ (Plural,
       [ decline Voc "ayas"
       ; decline Nom "ayas"
       ; decline Acc "iin"
       ; decline Ins "ibhis"
       ; decline Dat "ibhyas"
       ; decline Abl "ibhyas"
       ; decline Gen "ayaa.naam"
       ; decline Loc "i.su"
       ])
 ]
 ; Declined Noun Neu
 [ (Plural,
       [ decline Voc "ii.ni"
       ; decline Nom "ii.ni"
       ; decline Acc "ii.ni"
       ; decline Ins "ibhis"
       ; decline Dat "ibhyas"
       ; decline Abl "ibhyas"
       ; decline Gen "ayaa.naam"
       ; decline Loc "i.su"
       ])
 ]
 ; Declined Noun Fem
 [ (Plural,
       [ declinf Voc "ras"
       ; declinf Nom "ras"
       ; declinf Acc "ras"
       ; declinf Ins ".rbhis"
       ; declinf Dat ".rbhyas"
       ; declinf Abl ".rbhyas"
       ; declinf Gen ".r.naam"
       ; declinf Loc ".r.su"
       ])
 ]
 ; Bare Noun (code "tri")
 ; Bare Noun (code "tis.r") (* tis.rdhanva Whitney§482f *)
 ]
```

```
;
value build_catur entry =
  let decline case suff =
    (case, fix (revcode "cat") suff)
  and declinf case suff =
    (case, fix (revcode "catas") suff) in
  enter entry
   [ Declined Noun Mas
   [ (Plural,
         [ decline Voc "vaaras"
         ; decline Nom "vaaras"
         ; decline Acc "uras"
         ; decline Ins "urbhis"
         ; decline Dat "urbhyas"
         ; decline Abl "urbhyas"
         ; decline Gen "ur.naam"
         ; decline Loc "ur.su"
         ])
   ]
   ; Declined Noun Neu
   [ (Plural,
         [ decline Voc "vaari"
         ; decline Nom "vaari"
         ; decline Acc "vaari"
         ; decline Ins "urbhis"
         ; decline Dat "urbhyas"
         ; decline Abl "urbhyas"
         ; decline Gen "ur.naam"
         ; decline Loc "ur.su"
         ])
   ]
   ; Declined Noun Fem
   [ (Plural,
         [ declinf Voc "ras"
         ; declinf Nom "ras"
         ; declinf Acc "ras"
         ; declinf Ins ".rbhis"
         ; declinf Dat ".rbhyas"
         ; declinf Abl ".rbhyas"
         ; declinf Gen ".r.naam"
```

```
        ; declinf Loc ".r.su"
        ])
    ]
    ; Bare Noun (code "catur")
    ; Avyayaf (code "caturam")
    ]
;
value build_sat entry =
  let stem = revcode ".sa" in
  let decline case suff = (case, fix stem suff) in
  enter entry
    [ Declined Noun (Deictic Numeral)
    [ (Plural,
        [ decline Voc ".t"
        ; decline Nom ".t"
        ; decline Acc ".t"
        ; decline Ins ".dbhis"
        ; decline Dat ".dbhyas"
        ; decline Abl ".dbhyas"
        ; decline Gen ".n.naam"
        ; decline Loc ".tsu"
        ])
    ]
    ; Bare Noun (code ".sa.t")
    ]
;
(* Numerals 5, 7, 8, 9, 10, 11-19 *)
value build_num stem entry =
  let decline case suff = (case, fix stem suff) in
  enter entry (
    [ Declined Noun (Deictic Numeral)
    [ (Plural, let l =
        [ decline Nom "a" (* plural although no proper plural form Whitney§483 *)
        ; decline Acc "a"
        ; decline Ins "abhis"
        ; decline Dat "abhyas"
        ; decline Abl "abhyas"
        ; decline Gen "aanaam"
        ; decline Loc "asu"
        ] in if entry = "a.s.tan" then
```

```
        [ decline Nom "au" (* remains of dual form 8 as a pair of 4 (Vedic) *)
        ; decline Acc "au"
        ; decline Ins "aabhis"
        ; decline Dat "aabhyas"
        ; decline Abl "aabhyas"
        ; decline Loc "aasu"
        ] @ l else l)
    ]
    ; Bare Noun (wrap stem 1)
    ; Cvi (wrap stem 4)
    ] @ (if entry = "a.s.tan" then
             [ Bare Noun (wrap stem 2) (* a.s.taa *) ]
         else []))
;
value build_kati entry =
  let decline case suff =
      (case, fix (revcode "kat") suff) in
  enter1 entry
   ( Declined Noun (Deictic Numeral)
   [ (Plural,
        [ decline Voc "i"
        ; decline Nom "i"
        ; decline Acc "i"
        ; decline Ins "ibhis"
        ; decline Dat "ibhyas"
        ; decline Abl "ibhyas"
        ; decline Gen "iinaam"
        ; decline Loc "i.su"
        ])
   ]
   )
;
(* Here end the declension tables *)
```

The next two functions, as well as the special cases for -vas ought to disappear, when declension will be called with a fuller morphological tag, and not just the gender

```
value pprvat = fun
  [ "avat" | "aapnuvat" | "kurvat" | "jiivat" | "dhaavat" | "dhaavat#1"
  | "dhaavat#2" | "bhavat#1" | "z.r.nvat" | "zaknuvat" → True
  | _ → False
```

```
  ]
and pprmat = fun
  [ "jamat" | "dyumat" | "bhaamat" → True
  | _ → False
  ]
;
(* tad -¿ tat yad -¿ yat cid -¿ cit etc mais pas de visarga pour r ou s *)
value terminal_form = fun
  [ [ 34 :: w ] → [ 32 :: w ]
  | w → w
  ]
;
(* Big switch between paradigms. e :string is the entry, stem : word one of its (reversed)
stems, d : declension_class gives gender or indeclinable, p :string provides morphology or is
empty if not known *)
value compute_nouns_stem_form e stem d p =
  try match d with
  [ Gender g → match g with
    [ Mas → match stem with
      [ [ 1 :: r1 ] (* -a *) → match r1 with
            [ [ 17 ] (* ka as mas stem of kim *)
            | [ 17; 10 ] (* eka *)
            | [ 17; 10; 36; 1 ] (* aneka *)
            | [ 31; 3; 47; 17; 1; 34 ] (* dak.si.na *)
            | [ 41; 1; 32; 1; 17 ] (* katama *)
            | [ 41; 3; 22; 46; 1; 37 ] (* pazcima *)
            | [ 41; 10; 36 ] (* nema Whitney§525c *)
            | [ 42 ] (* ya#1 *)
            | [ 42; 1; 40; 5 ] (* ubhaya *)
            | [ 42; 36; 1 ] (* anya *)
            | [ 43; 1; 32; 1; 17 ] (* katara *)
            | [ 43; 1; 32; 1; 17; 10 ] (* ekatara *)
            | [ 43; 1; 32; 3 ] (* itara *)
            | [ 43; 1; 32; 1; 42; 36; 1 ] (* anyatara *) (* Whitney§523 *)
            | [ 43; 1; 32; 32; 5 ] (* uttara *)
            | [ 43; 1; 32; 36; 1 ] (* antara *)
            | [ 43; 1; 35; 1 ] (* adhara *)
            | [ 43; 1; 37 ] (* para *)
            | [ 43; 1; 37; 1 ] (* apara *)
            | [ 43; 1; 45; 1 ] (* avara *)
```

```
          | [ 45; 43; 1; 48 ] (* sarva *)
          | [ 45; 43; 6; 37 ] (* puurva *)
          | [ 45; 46; 3; 45 ] (* vizva *)
          | [ 45; 32 ] (* tva *)
          | [ 45; 48 ] (* sva *) → build_pron_a Mas r1 e
          | [ 36; 10 ] (* ena *) → build_ena Mas "idam"
          | [ 47; 10 ] (* e.sa *) when (e ="etad" ∨ e ="e.sa#1" ∨ e ="e.sa")
               → build_sa_tad Mas [ 10 ] e
          | [ 48 ] (* sa *) when (e ="tad" ∨ e ="sa#2" ∨ e ="sa")
               → build_sa_tad Mas [] e
          | [ 42; 48 ] (* sya *) → build_sya_tyad Mas e
          | [ 41; 12; 44; 5; 29; 13 ] (* au.duloma *) → (* Kale 26 *)
            let ps = revcode "u.duloma" in build_auduloma Mas r1 ps e
          | _ → build_mas_a r1 e
          ]
  | [ 2 :: r1 ] (* -aa - rare *) → match r1 with
        [ [ 19 :: [ 1 :: [ 41 :: [ 2 :: [ 48 ] ] ] ] ] (* saamagaa *)
        | [ 28 :: [ 47 :: _ ] ] (* -.s.thaa savya.s.thaa *)
        | [ 33 :: [ 48 :: _ ] ] (* -sthaa (?) *)
        | [ 34 :: _ ] (* -daa yazodaa *)
        | [ 35 :: _ ] (* -dhaa yazodhaa *)
        | [ 37 :: _ ] (* -paa gopaa vizvapaa dhenupaa somapaa etc Kale *)
        | [ 40 :: _ ] (* vibhaa2 *)
        | [ 41 :: _ ] (* pratimaa and -dhmaa: pa.nidhmaa zafkhadhmaa mukhadhmaa
agnidhmaa *)
        | [ 42 :: [ 14 :: _ ] ] (* zubha.myaa *)
        | [ 43 :: [ 17 :: _ ] ] (* -kraa dadhikraa *)
        | [ 43 :: _ ] (* -raa2 *)
            → build_mono_aa Mas r1 e
        | [ 49; 2; 49 ] (* haahaa *)
        | [ 31; 2; 43 ] (* raa.naa *) → build_mas_aa_no_root r1 e
        | _ → report stem g (* monitoring *)
        ]
  | [ 3 :: r1 ] (* -i *) → match e with
        [ "sakhi" → build_sakhi r1 e True
        | "pati" → (* P{I.4.8,9} optional ghi *)
                     do { build_sakhi r1 e False; build_mas_i stem r1 e }
        | _ → build_mas_i stem r1 e (* agni, etc (ghi) *)
        ]
  | [ 4 :: r1 ] (* -ii - rare *) →
```

```
      if bi_consonant r1 then build_bicons_ii Mas r1 e (* yavakrii *)
      else if monosyl r1 ∨ compound_monosyl_ii r1
           then build_mono_ii Mas r1 e
           else build_poly_ii Mas r1 e (* rathii sudhii *)
  | [ 5 :: r1 ] (* -u *) → match r1 with
      [ [ 27; 47; 12; 43; 17 ] → build_krostu r1 e (* = kro.s.t.r *)
      | _ → build_mas_u stem r1 e (* vaayu, etc (ghi) *)
      ]
  | [ 6; 49; 6; 49 ] (* huuhuu *) → build_huuhuu e
  | [ 6 :: r1 ] (* -uu - rare *) →
      if monosyl r1 then build_mono_uu Mas r1 e (* puu2 *)
      else build_poly_uu Mas r1 e (* sarvatanuu khalapuu pratibhuu *)
         (* vedic polysyllabic in uu are of utmost rarity - Whitney §355 *)
  | [ 7 :: r1 ] (* -.r *) → match r1 with
      [ [ 27; 47; 12; 43; 17 ] → build_krostu r1 e (* kro.s.t.r Muller §236 *)
      | [ 32 :: r2 ] (* -t.r *) → match r2 with
           [ [ 3; 37 ] (* pit.r *) (* relationships McDonell §101 *)
           | [ 2; 41; 2; 24 ] (* jaamaat.r *)
           | [ 36; 1; 42; 1; 37; 3 ] (* upayant.r *)
           | [ 2; 43; 40 ] (* bhraat.r *) → build_mas_ri_g r1 e
           (* napt.r bhart.r pari.net.r dev.r: parenthood relation follow: *)
           | _ → (* dhaat.r general agent paradigm *) build_mas_ri_v r1 e
         ]
      | [ 36 ] (* n.r *) → build_nri r1 e
      | _ → build_mas_ri_v r1 e
      ]
  | [ 8 :: _ ]
  | [ 9 :: _ ] → report stem g
  | [ 10 :: r1 ] (* -e *) → build_e Mas r1 e (* apte (?) *)
  | [ 11 :: r1 ] → match r1 with
        [ [ 43 ] (* rai *) → build_rai Mas [ 2; 43 ] e
        | _ → report stem g
        ]
  | [ 12 :: r1 ] (* -o *) → build_o Mas r1 e
  | [ 13 :: r1 ] (* -au *) → match r1 with
        [ [ 48; 1 ] (* asau *) → build_asau_adas Mas
        | _ → build_au Mas r1 e
        ]
  | [ 22 :: r1 ] (* -c *) → match r1 with
        [ [ 1 :: r2 ] (* -ac *) → match r2 with
```

```
                [ [] → () (* ac utilisé seulement avec px *)
                | [ 42 :: r3 ] (* yac *) → build_mas_yac r3 e
                | [ 45 :: r3 ] (* vac *) → build_mas_vac r3 e
                | _ (* udac ... *) → build_mas_ac r2 e
                ]
          | [ 2 :: r2 ] (* -aac *) → match r2 with
                [ [ 37; 1 ] (* apa-ac *)
                | [ 42; 48; 1; 17 ] (* kasya-ac *)
                | [ 43; 1; 37 ] (* para-ac *)
                | [ 43; 37 ] (* pra-ac *)
                | [ 45; 1 ] (* ava-ac *)
                | [ 45; 34; 1; 10; 34 ] (* devadra-ac *)
                | [ 45; 43; 1 ] (* arva-ac *)
                | [ 45; 43; 1; 48 ] (* sarva-ac *)
                  → build_mas_aac r1 e
                | _ → build_root Mas stem e
                ]
          | _ → build_root Mas stem e
          ]
  | [ 24 :: r1 ] (* -j *) → match r1 with (* m.rjify *)
          [ [ 1 :: [ 43 :: _ ] ] (* -yaj2 upaya.t *)
          | [ 2 :: [ 43 :: _ ] ] (* -raaj2 viraaj2 *)
          | [ 2 :: [ 42 :: _ ] ] (* -yaaj2 *)
          | [ 7; 48 ] (* s.rj2 *) → build_root Mas [ 124 (* j' *) :: r1 ] e
          | [ 5; 42 ] (* yuj2 *) → do
               { build_root Mas stem e
               ; build_archaic_yuj [ 24; 26; 5; 42 ] (* yu nj *) Mas e
               }
          | _ → build_root Mas stem e
          ]
  | [ 32 :: r1 ] (* -t *) → match r1 with
          [ [ 1 :: r2 ] (* -at *) → if is_redup r2 then build_mas_red r1 e
                                        else match r2 with
                [ [ 41 :: r3 ] (* -mat *) →
                  if p ="Ppra" ∨ pprmat e then build_mas_at r1 e
                                               else build_mas_mat r2 e
                  (* Whitney§451 : yat iyat kiyat *)
                | [ 42 ] | [ 42; 3 ] | [ 42; 3; 17 ] →
                  if p ="Ppra" then build_mas_at r1 e (* yat2 *)
                  else build_mas_mat r2 e
```

```
          | [ 45 :: r3 ] (* -vat *) →
            if p ="Ppra" ∨ pprvat e then build_mas_at r1 e
            else if e ="maghavat" then build_mas_maghavan e else build_mas_mat r2 e
          | [ 49 :: [ 1 :: [ 41 :: _ ] ] ] (* mahat, sumahat *)
                  → build_mas_mahat r2 e
          | [ 34 ] (* dat *) → build_root_weak Mas stem "danta"
          | _ → build_mas_at r1 e (* p.r.sat, jagat, like ppr *)
          ]
      | [ 2 :: r2 ] (* -aat *) → match r2 with
          [ [ 37; 1; 36 ] (* vedic napaat *) → build_root Mas stem e
          | _ → build_mas_at r1 e (* ppr in aat/aant ? *)
          ]
      | _ → build_root Mas stem e
      ]
  | [ 34 :: r1 ] (* -d *) → match r1 with
      [ [ 1; 37 ] (* pad *) → build_root_weak Mas stem "paada"
      | [ 1 :: [ 37 :: s ] ] (* -pad *) → build_pad Mas s e
      | _ → build_root Mas stem e
      ]
  | [ 36 :: r1 ] (* -n *) → match r1 with
        [ [ 1 :: r2 ] (* -an *) → match r2 with
            [ [ 47 :: [ 6 :: [ 37 ] ] ] (* puu.san *)
                → build_an_god r2 e (* Whitney §426a *)
            | [ 41 :: r3 ] (* -man *) → match r3 with
                  [ [ 1 :: [ 42 :: [ 43 :: [ 1 ] ] ] ] (* aryaman *)
                      → build_man_god r3 e (* Whitney §426a *)
                  | _ → build_man Mas r3 e
                  ]
            | [ 45 :: ([ 46 :: _ ] as r3) ] (* -zvan *) → build_mas_zvan r3 e
                                                    (* takes care of eg dharmazvan *)
            | [ 45 :: r3 ] (* -van *) → match e with
                [ "yuvan" → build_mas_yuvan e
                | "maghavan" → build_mas_maghavan e
                  (* NB: entry is maghavat but interface allows maghavan *)
                | _ → build_van Mas r3 e
                ]
            | [ 49 :: r3 ] (* -han *) → build_han r3 e
            | _ → build_an Mas r2 e (* raajan *)
            ]
      | [ 3 :: r2 ] (* -in *) → match r2 with
```

```
          [ [ 33 :: r3 ] (* -thin *)→ match r3 with
              [ [ 1 :: [ 37 :: _ ] ] (* -pathin *) (* P{7,1,85} *)
              | [ 1 :: [ 41 :: _ ] ] (* -mathin *)
                  → build_athin r3 e
              | _ → build_mas_in r2 e
              ]
          | [ 47; 17; 5; 40; 7 ] (* -.rbhuk.sin *) (* P{7,1,85} *)
              → build_ribhuksin r2 e
          | _ → build_mas_in r2 e
          ]
      | _ → report stem g
      ]
| [ 37 :: [ 1 :: [ 48 :: r ] ] ] (* -sap *) → build_sap Mas r e
| [ 41 :: r1 ] (* -m *) → match r1 with
      [ [ 1; 42; 1 ] (* ayam *) → build_ayam_idam Mas
      | [ 1; 34 ] (* dam2 *) → (* build_dam e *)
            () (* skipped - only gen. vedic forms except dam-pati *)
      | _ → build_root_m Mas r1 stem e (* was report stem g *)
      ]
| [ 45 :: r1 ] (* -v *) → match r1 with
      [ [ 3; 34 ] (* div *) → build_div Mas [ 34 ] e
      | [ 4; 34 ] (* diiv *) → () (* avoids reporting bahu *)
      | _ → report stem g
      ]
| [ 47 :: r1 ] (* .s *) → match r1 with
        [ [ 3 :: r2 ] → match r2 with
            [ [ 45; 1; 19 ] (* gavi.s *)
            | [ 45; 34 ] (* dvi.s *)
            | [ 45; 34; 3; 45 ] (* vidvi.s *)
            | [ 45; 34; 1; 32; 1; 49 ] (* hatadvi.s *)
            | [ 28; 1; 37; 3; 37 ] (* pipa.thi.s *)
                → build_is Mas r2 e (* Kale §114 *)
            | _ → build_root Mas stem e
            ]
        | [ 5 :: r2 ] → match r2 with
            [ [ 24 :: [ 1 :: [ 48 ] ] ] (* saju.s *)
                → build_us Mas r2 e (* Kale§114 *)
            | _ → build_root Mas stem e
            ]
        | _ → build_root Mas stem e
```

```
          ]
    | [ 48 :: r1 ] (* -s *) → match r1 with
          [ [ 1 :: r2 ] (* -as *) → match r2 with
              [ [ 42 :: _ ] (* -yas *) → build_mas_yas r2 e
              | [ 45 :: r3 ] (* -vas *) →
                if p = "Ppfta" then build_mas_vas r3 e
                else match r3 with
                  [ [ 1 :: [ 43 :: _ ] ] (* -ravas *) → build_as Mas r2 e
                    (* uccaisravas, puruuravas, ugrazravas, vizravas non ppf *)
                  | [ 3 :: r4 ] (* -ivas *) → build_mas_ivas r4 e
                  | [ 35 :: _ ] (* -dhvas *)
                  | [ 5 :: [ 48 :: _ ] ] (* -suvas *) → build_root Mas stem e
                  | _ (* other ppf *) → build_mas_vas r3 e
                  ]
              | [ 43 :: [ 48 :: _ ]] (* -sras *) → build_root Mas stem e
(* — [ 46; 1; 33; 17; 5 ] (× ukthazas × ) → build_ukthazas Mas e *)
(* — [ 46 :: _ ] (× − zas × ) → build_root Mas stem e *)
              | _ → build_as Mas r2 e
              ]
          | [ 2; 41 ] (* maas *) → build_maas ()
          | [ 2 :: _ ] (* -aas *) → () (* avoids reporting bahu aas bhaas *)
          | [ 3 :: r2 ] (* -is *) → match r2 with
              [ [ 46; 2 :: _ ] (* niraazis *) → build_aazis Mas r2 e
              | _ → build_is Mas r2 e (* udarcis *)
              ]
          | [ 5 :: r2 ] (* -us *) → build_us Mas r2 e (* acak.sus *)
          | [ 12; 34 ] (* dos *) → build_dos Mas e
          | [ 14; 5; 37 ] (* pu.ms *) → build_pums [ 41; 5; 37 ] stem e
          | [ 14; 5; 37; 1; 36 ] (* napu.ms *)
            → build_pums [ 41; 5; 37; 1; 36 ] stem e
          | [ 14; 2; 41 ] (* maa.ms *) → () (* avoids reporting bahu *)
          | _ → report stem g
          ]
    | [ 49 :: r1 ] (* -h *) → match r1 with
          [ [ 1 :: [ 45 :: r3 ] ] (* vah2 *) → match e with
              [ "ana.dvah" → build_anadvah r3 e
              | _ → build_mas_vah r3 e
              ]
          | [ 1; 34 ] (* dah2 *) (* mandatory duhify *)
          | [ 5; 34 ] (* duh2 *) → build_root Mas [ 149 (* h' *) :: r1 ] e
```

```
            | [ 3 :: [ 36 :: [ 48 :: _ ] ] ] (* -snih2 *)
            | [ 5 :: [ 36 :: [ 48 :: _ ] ] ] (* -snuh2 *)
            | [ 5 :: [ 43 :: [ 34 :: _ ] ] ] (* -druh2 *) → do
                { build_root Mas [ 149 (* h' *) :: r1 ] e
                ; build_root Mas stem e (* optionally duhify *)
                }
            | _ → build_root Mas stem e
            ]
        | _ → build_root Mas stem e
        ]
  | Neu → match stem with
    [ [ 1 :: r1 ] (* -a *) → match r1 with
            [ [ 17; 10 ] (* eka *) (* pronouns *)
            | [ 17; 10; 36; 1 ] (* aneka *)
            | [ 31; 3; 47; 17; 1; 34 ] (* dak.si.na *)
            | [ 41; 1; 32; 1; 17 ] (* katama *)
            | [ 41; 3; 22; 46; 1; 37 ] (* pazcima *)
            | [ 42; 1; 40; 5 ] (* ubhaya *)
            | [ 43; 1; 32; 1; 17 ] (* katara *)
            | [ 43; 1; 32; 1; 17; 10 ] (* ekatara *)
            | [ 43; 1; 32; 3 ] (* itara *)
            | [ 43; 1; 32; 32; 5 ] (* uttara *)
            | [ 43; 1; 32; 36; 1 ] (* antara *)
            | [ 43; 1; 35; 1 ] (* adhara *)
            | [ 43; 1; 37 ] (* para *)
            | [ 43; 1; 37; 1 ] (* apara *)
            | [ 43; 1; 45; 1 ] (* avara *)
            | [ 45; 43; 6; 37 ] (* puurva *)
            | [ 45; 46; 3; 45 ] (* vizva *)
            | [ 45; 43; 1; 48 ] (* sarva *)
            | [ 45; 48 ] (* sva *)
            (* — 45; 32 cf tvad clash with tva taddhita ending *)
                → build_pron_a Neu r1 e
            | _ → build_neu_a r1 e
            ]
    | [ 2 :: _ ] → report stem Neu (* (missing) ahigopaa raa vibhaa sthaa *)
    | [ 3 :: r1 ] (* -i *)
    | [ 4 :: r1 ] (* -ii - rare *) → build_neu_i r1 e
    | [ 5 :: r1 ] (* -u *)
    | [ 6 :: r1 ] (* -uu - rare *) → build_neu_u r1 e
```

```
| [ 7 :: r1 ] (* -.r *) → build_neu_ri r1 e
| [ 11; 43 ] (* rai *)
| [ 12; 19 ] (* go *)
| [ 13; 36 ] (* nau *)
| [ 13; 44; 19 ] (* glau *)
| [ 13; 48; 1 ] (* asau *) → () (* avoids reporting bahu *)
| [ 8 :: _ ]
| [ 9 :: _ ]
| [ 10 :: _ ]
| [ 11 :: _ ]
| [ 12 :: _ ]
| [ 13 :: _ ] → report stem g
| [ 22 :: r1 ] (* -c *) → match r1 with
      [ [ 1 :: r2 ] (* -ac *) → match r2 with
            [ [] → () (* ac utilisé seulement avec px *)
            | [ 42 :: r3 ] → build_neu_yac r3 e
            | [ 45 :: r3 ] → build_neu_vac r3 e
            | _ (* udac ... *) → build_neu_ac r2 e
            ]
      | [ 2 :: _ ] (* -aac *) → build_neu_aac r1 e
      | _ → build_root Neu stem e
      ]
| [ 24 :: r1 ] (* -j *) → match r1 with (* m.rjify *)
      [ [ 2 :: [ 43 :: _ ] ] (* -raaj2 viraaj2 *)
      | [ 2 :: [ 42 :: _ ] ] (* -yaaj2 *)
           → build_root Neu [ 124 (* j' *) :: r1 ] e
      | [ 5; 42 ] (* yuj2 *) → do
           { build_root Neu stem e
           ; build_archaic_yuj [ 24; 26; 5; 42 ] (* yu nj *) Neu e
           }
      | _ → build_root Neu stem e (* -s.rk as.rjk *)
      ]
 | [ 32 :: r1 ] (* -t *) → match r1 with
      [ [ 1 :: r2 ] (* -at *) → if is_redup r2 then build_neu_red r1 e
                                    else match r2 with
          [ [ 49 :: [ 1 :: [ 41 :: _ ] ] ] (* mahat, sumahat *)
               → build_neu_mahat r2 e
          | _ → build_neu_at r1 e (* e.g. jagat *)
          ]
      | [ 2 :: r2 ] (* -aat *) → build_neu_at r1 e (* ppr in aat/aant ? *)
```

```
          | _ → build_root Neu stem e
          ]
  | [ 34 :: r1 ] (* -d *) → match r1 with
      [ [ 1 :: r2 ] (* -ad *) → match r2 with
        [ [ 32 ] (* tad *) → do
              { build_sa_tad Neu [] e
              ; enter e [ Bare Noun (code "tat") ]
              }
        | [ 32; 10 ] (* etad *) → build_sa_tad Neu [ 10 ] e
        | [ 42; 32 ] (* tyad *) → build_sya_tyad Neu e
        | [ 36; 10 ] (* enad *) → build_ena Neu "idam"
        | [ 37 ] (* pad *) → build_root_weak Neu stem "paada"
        | [ 37 :: s ] (* -pad *) → build_pad Neu s e
        | [ 42 ] (* yad *)
        | [ 45; 32 ] (* tvad *)
        | [ 42; 36; 1 ] (* anyad *)
        | [ 43; 1; 32; 1; 42; 36; 1 ] (* anyatarad *) (* Whitney§523 *)
             → build_pron_a Neu r2 e
        | _ → build_root Neu stem e
        ]
      | [ 7; 49 ] (* h.rd *)
           → build_root_weak Neu stem "h.rdaya" (* P{6,1,63} Whitney§397 *)
      | _ → build_root Neu stem e
      ]
  | [ 36 :: r1 ] (* -n *) → match r1 with
        [ [ 1 :: r2 ] (* -an *) → match r2 with
            [ [ 33; 17; 1; 48 ] (* sakthan *)
            | [ 33; 48; 1 ] (* asthan *)
            | [ 47; 17; 1 ] (* ak.san *)
            | [ 35; 1; 34 ] (* dadhan *) → build_aksan r2 e
            | [ 17; 1; 42 ] (* yakan *)
            | [ 17; 1; 46 ] (* zakan *)
            | [ 34; 5 ] (* udan *)
            | [ 47; 6; 42 ] (* yuu.san *)
            | [ 47; 12; 34 ] (* do.san *)
            | [ 48; 1 ] (* asan *)
            | [ 48; 2 ] (* aasan *) → build_sp_an r2 e (* Whitney§432 *)
            | [ 35; 6 ] (* uudhan *) → build_uudhan r2 e
            | [ 41 :: r3 ] (* -man *) → match e with
                [ "brahman" → build_neu_brahman e
```

```
            | _ → build_man Neu r3 e
            ]
          | [ 45 :: r3 ] (* -van *) → match e with
            [ "yuvan" → build_neu_yuvan e
            | _ → build_van Neu r3 e
            ]
          | [ 49 :: r3 ] (* -han *) → match r3 with
            [ [ 1 :: _ ] (* -ahan *)
            | [ 2; 42; 2; 48 ] (* saayaahan *) → build_ahan r2 e
            | _ (* -han2 *) → build_an Neu r2 e
            ]
          | _ → build_an Neu r2 e
          ]
        | [ 3 :: r2 ] (* -in *) → build_neu_in r2 e
        | _ → report stem g
        ]
    | [ 37 :: [ 1 :: [ 48 :: r ] ] ] (* -sap *) → build_sap Neu r e
    | [ 41 :: r1 ] (* -m *) → match r1 with
        [ [ 1; 34; 3 ] (* idam *) → build_ayam_idam Neu
        | [ 3; 17 ] (* kim *) → build_pron_a Neu [ 17 ] e
        | _ → build_root_m Neu r1 stem e (* was report stem g *)
        ]
    | [ 45 :: r1 ] (* -v *) → match r1 with
        [ [ 3; 34 ] (* div *) → build_div Neu [ 34 ] e
        | [ 4; 34 ] (* diiv *) → () (* avoids reporting bahu *)
        | _ → report stem g
        ]
    | [ 47 :: r1 ] (* .s *) → match r1 with
        [ [ 3 :: r2 ] → match r2 with
            [ [ 45; 1; 19 ] (* gavi.s *)
            | [ 45; 34; 1; 32; 1; 49 ] (* hatadvi.s *)
            | [ 28; 1; 37; 3; 37 ] (* pipa.thi.s *)
                → build_is Neu r2 e
            | _ → build_root Neu stem e
            ]
        | [ 5 :: r2 ] → match r2 with
            [ [ 24 :: [ 1 :: [ 48 ]]] (* saju.s *)
                → build_us Neu r2 e
            | _ → build_root Neu stem e
            ]
```

```
        | _ → build_root Neu stem e
        ]
  | [ 48 :: r1 ] (* -s *) → match r1 with
        [ [ 1 :: r2 ] (* -as *) → match r2 with
            [ [ 34; 1 ] (* adas *) → build_asau_adas Neu
            | [ 42 :: _ ] (* -yas *) → build_neu_yas r2 e
            | [ 45 :: r3 ] (* -vas *) →
              if p = "Ppfta" then build_neu_vas r3 e
              else match r3 with
              [ [ 1 ] (* avas1 - non ppf *)
              | [ 1 :: [ 43 :: _ ] ] (* -ravas eg zravas, sravas - non ppf *)
              | [ 5 :: [ 48 :: _ ] ] (* -suvas *)
              | [ 3; 43; 1; 45 ] (* varivas *) → build_as Neu r2 e
              | [ 3 :: r4 ] (* ivas *) → build_neu_ivas r4 e
              | [ 35 :: _ ] (* -dhvas *) → build_root Neu stem e
              | _ (* other ppf *) → build_neu_vas r3 e
              ]
            | [ 43 :: [ 48 :: _ ]] (* -sras *) → build_root Neu stem e
            | _ (* manas, ziras, ... *) → build_as Neu r2 e
            ]
        | [ 2 :: r2 ] (* -aas *) → match r2 with
            [ [] → build_neu_aas stem e (* aas3 irregular *)
            | [ 17 ] (* kaas2 *)
            | [ 41 ] (* maas *) → () (* avoids reporting bahu *)
            | [ 40 :: _ ] (* bhaas aabhaas *) → () (* missing paradigm *)
            | _ → report stem Neu
            ]
        | [ 3 :: r2 ] (* -is *) → build_is Neu r2 e (* jyotis havis *)
        | [ 5 :: r2 ] (* -us *) → build_us Neu r2 e (* cak.sus dhanus *)
        | [ 12; 34 ] (* dos *) → build_dos Neu e
        | _ → build_root Neu stem e
        ]
  | [ 49 :: r1 ] (* -h *) → match r1 with
        [ [ 1; 34 ] (* dah2 *) (* duhify *)
        | [ 5; 43; 34 ] (* druh2 *) → do
            { build_root Neu [ 149 (* h' *) :: r1 ] e (* optionally duhify *)
            ; build_root Neu stem e
            }
        | _ → build_root Neu stem e
        ]
```

```
    | _ → build_root Neu stem e
    ]
  | Fem → match stem with
    [ [ 1 :: _ ] → report stem g
    | [ 2 :: r1 ] (* -aa *) → match r1 with
                    [ [ 42 ] (* yaa *) → match e with
              [ "ya#1" | "yad" | "yaa#2" → build_pron_aa r1 e (* pn yaa#2 *)
              | "ya#2" | "yaa#3" → build_fem_aa r1 e (* ifc. -yaa#3 *)
              | _ → report stem g
              ]
        | [ 17 ] (* kaa *)
        | [ 17; 10 ] (* ekaa *)
        | [ 17; 10; 36; 1 ] (* anekaa *)
        | [ 31; 3; 47; 17; 1; 34 ] (* dak.si.naa *)
        | [ 41; 1; 32; 1; 17 ] (* katamaa *)
        | [ 41; 3; 22; 46; 1; 37 ] (* pazcimaa *)
        | [ 42; 36; 1 ] (* anyaa *)
        | [ 43; 1; 32; 1; 17 ] (* kataraa *)
        | [ 43; 1; 32; 1; 17; 10 ] (* ekataraa *)
        | [ 43; 1; 32; 1; 42; 36; 1 ] (* anyataraa *) (* Whitney§523 *)
        | [ 43; 1; 32; 3 ] (* itaraa *)
        | [ 43; 1; 32; 32; 5 ] (* uttaraa *)
        | [ 43; 1; 32; 36; 1 ] (* antaraa *)
        | [ 43; 1; 35; 1 ] (* adharaa *)
        | [ 43; 1; 37 ] (* paraa *)
        | [ 43; 1; 37; 1 ] (* aparaa *)
        | [ 43; 1; 45; 1 ] (* avaraa *)
        | [ 45; 43; 1; 48 ] (* sarvaa *)
        | [ 45; 43; 6; 37 ] (* puurvaa *)
        | [ 45; 46; 3; 45 ] (* vizvaa *)
        | [ 45; 48 ] (* svaa *)
        | [ 45; 32 ] (* tvaa *)
              → build_pron_aa r1 e
        | [ 36; 10 ] (* enaa *) → build_ena Fem "idam"
        | [ 47; 10 ] (* e.saa *) when e ="etad" ∨ e ="e.saa"
              → build_saa [ 10 ] e
        | [ 48 ] (* saa *) → build_saa [] e
        | [ 42 ; 48 ] (* syaa *) → build_syaa [] e
        | _ → build_fem_aa r1 e
        ]
```

```
| [ 3 :: r1 ] (* -i *) → build_fem_i stem r1 e
| [ 4 :: r1 ] (* -ii *) →
      (* match r1 with [ [ 37 :: [ 2 :: _ ] ] (× − aapii × ) | _ → ] *)
  if monosyl r1 ∨ compound_monosyl_ii r1 then match r1 with
    [ [ 43; 32; 48 ] (* strii *) → build_strii r1 e
    | [ 43; 46 ] (* zrii *) → do
      { build_mono_ii Fem r1 e (* nom. ii.h *)
      ; build_fem_ii r1 e (* MW *) (* nom. ii Pan6,1,68 sulopa *)
      }
    | _ → build_mono_ii Fem r1 e (* dhii hrii bhii2 *)
    ]
  else do
      { if r1 = [ 22; 1 ] (* -acii *) then () (* seulement avec px *)
        else build_fem_ii r1 e (* nom. ii Pan6,1,68 sulopa *)
      ; match r1 with (* vedic forms Whitney§355-356 *)
        [ [ 45; 1 ] (* avii *)
        | [ 34; 1; 36 ] (* nadii *)
        | [ 41; 43; 6; 48 ] (* suurmii *)
        | [ 41; 47; 17; 1; 44 ] (* lak.smii *)
        | [ 43; 1; 32 ] (* tarii *) (* Whitney§363a *)
        | [ 43; 32; 36; 1; 32 ] (* tantrii *)
        | [ 43; 1; 32; 48 ] (* starii *) (* Deshpande u.naadisuutra *)
(* HN Bhat: avii tantrii tarii lak.smii hrii dhii zrii in u.naadi *)
(* autre liste: tantrii starii lak.smii tarii dhii hrii zrii *)
(* ci-dessus: + nadii suurmii - dhii hrii traités par build_mono_ii *)
          → build_poly_ii Fem r1 e (* nom. ii.h *)
        | _ → ()
        ]
      }
| [ 5 :: r1 ] (* u *) → build_fem_u stem r1 e
| [ 6 :: r1 ] (* -uu *) →
  if monosyl r1 ∨ compound_monosyl_uu r1 then build_mono_uu Fem r1 e
  else do
      { build_fem_uu r1 e
      ; match r1 with (* vedic forms Whitney§355-356 *)
        [ [ 35; 1; 45 ] (* vadhuu *)
        | [ 36; 1; 32 ] (* tanuu *)
        | [ 41; 1; 22 ] (* camuu *)
          → build_poly_uu Fem r1 e
        | _ → ()
```

```
                ]
              }
    | [ 7 :: r1 ] (* -.r *) → match r1 with
          [ [ 32 :: r2 ] (* -t.r *) → match r2 with
              [ [ 2; 41 ] (* maat.r *) (* relationships McDonnel §101 *)
              | [ 3; 49; 5; 34 ] (* duhit.r *) → build_fem_ri_g r1 e
              | _ → build_fem_ri_v r1 e
              ]
          | [ 34; 36; 2; 36; 1; 36 ] (* nanaand.r *)
          | [ 34; 36; 1; 36; 1; 36 ] (* nanaand.r *)
                → build_fem_ri_g r1 e
          | _ → build_fem_ri_v r1 e (* including relationship svas.r *)
          ]
    | [ 8 :: _ ]
    | [ 9 :: _ ]
    | [ 10 :: _ ] → report stem Fem
    | [ 11 :: r1 ](* -ai *) → match r1 with
          [ [ 43 ] (* rai *) → build_rai Fem [ 2; 43 ] e
(* — 39; 41; 5; 41 (* mumbai *) -¿ (* TO DO *) *)
(* — 48; 32; 32; 1; 48 (* sattasai *) -¿ (* idem *) *)
          | _ → report stem Fem
          ]
    | [ 12 :: r1 ] (* -o *) → build_o Fem r1 e
    | [ 13 :: r1 ] (* -au *) → match r1 with
          [ [ 48; 1 ] (* asau *) → build_asau_f ()
          | _ → build_au Fem r1 e
          ]
    | [ 24 :: r1 ] (* -j *) → match r1 with (* m.rjify *)
          [ [ 2 :: [ 43 :: _ ] ] (* -raaj2 viraaj2 *)
          | [ 2 :: [ 42 :: _ ] ] (* -yaaj2 *)
          | [ 7; 48 ] (* s.rj2 *) → build_root Fem [ 124 (* j' *) :: r1 ] e
          | [ 5; 42 ] (* yuj2 *) → do
                { build_root Fem stem e
                ; build_archaic_yuj [ 24; 26; 5; 42 ] (* yu nj *) Fem e
                }
          | _ → build_root Fem stem e
          ]
    | [ 32; 7; 37 ] (* p.rt *) → build_root_weak Fem stem "p.rtanaa"
    | [ 34 :: r1 ] (* -d *) → match r1 with
          [ [ 1; 37 ] (* pad *) → build_root_weak Fem stem "paada"
```

```
          | [ 1; 37; 2 ] (* aapad *)
          | [ 1; 37; 3; 45 ] (* vipad *)
          | [ 1; 37; 41; 1; 48 ] (* sampad *) → build_root Fem stem e
          | [ 1 :: [ 37 :: s ] ] (* -pad *) → build_pad Fem s e
          | _ → build_root Fem stem e
          ]
    | [ 36 :: r1 ] (* -n *) → match r1 with
          [ [ 1 :: r2 ] (* -an *) → match r2 with
              [ [ 41 :: r3 ] (* man *) → match r3 with
                  [ [ 2; 48 ] (* saaman *)
                  | [ 4; 48 ] (* siiman *) → build_man Fem r3 e (* check *)
                  | _ → report stem Fem
                  ]
              | _ → report stem Fem
              ]
          | _ → report stem Fem
          ]
    | [ 37; 1 ] (* ap *) → build_ap e
    | [ 37 :: [ 1 :: [ 48 :: r ] ] ] (* -sap *) → build_sap Fem r e
    | [ 41 :: r1 ] (* -m *) → match r1 with
          [ [ 1; 42; 3 ] (* iyam *) → build_iyam ()
          | _ → build_root_m Fem r1 stem e (* was report stem g *)
          ]
    | [ 43 :: r1 ] (* -r *) → match r1 with
          [ [ 2 :: _ ] (* -aar *) → build_root Fem stem e (* dvaar *)
          | [ 3 :: r2 ] (* -ir *) → build_fem_ir r2 e (* gir *)
          | [ 5 :: r2 ] (* -ur *) → build_fem_ur r2 e
          | [ 1 :: _ ] (* -praatar -sabar *) → ()
          | _ → report stem g
          ]
    | [ 45 :: r1 ] (* -v *) → match r1 with
          [ [ 3; 34 ] (* div *) → build_div Fem [ 34 ] e
          | [ 4; 34 ] (* diiv#2 *) → build_diiv e
          | _ → report stem g
          ]
    | [ 46; 3; 36 ] (* niz *) → build_root_weak Fem stem "nizaa"
    | [ 47 :: r1 ] (* -.s *) → match r1 with
          [ [ 3 :: r2 ] → match r2 with
                [ [ 28 :: [ 1 :: [ 37 :: [ 3 :: [ 37 ] ] ] ] ] (* pipa.thi.s *)
                    → build_is Fem r2 e
```

```
          | _ → build_root Fem stem e
          ]
      | [ 5 :: r2 ] → match r2 with
          [ [ 24 :: [ 1 :: [ 48 ] ] ] (* saju.s *)
              → build_us Fem r2 e
          | _ → build_root Fem stem e
          ]
      | _ → build_root Fem stem e
      ]
  | [ 48 :: r1 ] (* -s *) → match r1 with
      [ [ 1; 36 ] (* nas *) → build_nas e
      | [ 1 :: r2 ] (* -as *) → match r2 with
          [ [ 45 :: [ 35 :: _ ] ] (* -dhvas *)
          | [ 43 :: [ 48 :: _ ] ] (* -sras *) → build_root Fem stem e
          | [ 34; 1 ] (* adas *) → build_asau_f ()
          | _ → build_as Fem r2 e
          ]
      | [ 2 :: r2 ] (* -aas *) → build_root Fem stem e (* bhaas *)
      | [ 3 :: r2 ] (* -is *) → match r2 with
          [ [ 46; 2 :: _ ] (* -aazis *) → build_aazis Fem r2 e
          | _ → build_is Fem r2 e
          ]
      | [ 5 :: r2 ] (* -us *) → build_us Fem r2 e
      | [ 12; 34 ] (* dos *) → build_dos Fem e
      | [ 14; 2; 41 ] (* maa.ms *) → () (* avoids reporting bahu *)
      | [ 14 :: [ 5 :: _ ] ] → () (* -pu.ms *)
      | _ → report stem g
      ]
  | [ 49 :: r1 ] (* -h *) → match r1 with
      [ [ 1 :: [ 34 :: _ ] ] (* dah2 -dah *)
      | [ 5 :: [ 34 :: _ ] ] (* duh2 -duh *)
      | [ 3; 31; 47; 5 ] (* u.s.nih *) →
        build_root Fem [ 149 (* h' *) :: r1 ] e (* duhify *)
      | [ 3; 36; 48 ] (* snih2 *)
      | [ 5; 36; 48 ] (* snuh2 *)
      | [ 5 :: [ 43 :: [ 34 :: _ ] ] ] (* druh2 -druh *) → do
          { build_root Fem [ 149 (* h' *) :: r1 ] e (* optionally duhify *)
          ; build_root Fem stem e
          }
      | [ 1; 36; 2; 37; 5 ] → build_upaanah r1 stem e (* Kale§101 *)
```

```
          | _ → build_root Fem stem e
          ]
      | _ → build_root Fem stem e
      ]
  | Deictic _ → match stem with
      [ (* aham *) [ 41; 1; 49; 1 ] (* Dico *)
      | (* asmad *) [ 34; 1; 41; 48; 1 ] (* tradition *) → build_aham ()
      | (* tvad *) [ 34; 1; 45; 32 ] (* Dico *)
      | (* yu.smad *) [ 34; 1; 41; 47; 5; 42 ] (* tradition *) → build_tvad ()
      | (* aatman *) [ 36; 1; 41; 32; 2 ] → build_aatman e
      | (* eka *) [ 1; 17; 10 ] → warn stem "a␣Mas␣or␣Neu" (* pn in Dico *)
      | (* dvi *) [ 3; 45; 34 ] → build_dvi e
      | (* tri *) [ 3; 43; 32 ] → build_tri e
      | (* tis.r *) [ 7; 48; 3; 32 ]
      | (* trayas *) [ 48; 1; 42; 1; 43; 32 ]
      | (* trii.ni *) [ 3; 31; 4; 43; 32 ] → warn stem "tri"
      | (* catur *) [ 43; 5; 32; 1; 22 ] → build_catur e
      | (* catas.r *) [ 7; 48; 1; 32; 1; 22 ]
      | (* catvaari *) [ 3; 43; 2; 45; 32; 1; 22 ] → warn stem "catur"
      | (* .sa.s *) [ 47; 1; 47 ] → build_sat e
      | (* -an (numbers) *) [ 36 :: [ 1 :: st ] ] → match st with
             [ (* pa ncan *) [ 22; 26; 1; 37 ]
             | (* saptan *) [ 32; 37; 1; 48 ]
             | (* a.s.tan *) [ 27; 47; 1 ]
             | (* navan *) [ 45; 1; 36 ]
             | (* .so.dazan *) [ 46; 1; 29; 12; 47 ]
             | (* -dazan *) [ 46 :: [ 1 :: [ 34 :: _ ] ] ] → build_num st e
             | _ → report stem g
             ]
      | (* kati *) [ 3; 32; 1; 17 ] → build_kati e
      | (* vi.mzati *) [ 3; 32; 1; 46; 14; 3; 45 ]
      | (* .sa.s.ti *) [ 3; 27; 47; 1; 47 ]
      | (* saptati *) [ 3; 32; 1; 32; 37; 1; 48 ]
      | (* aziiti *) [ 3; 32; 4; 46; 1 ]
      | (* navati *) [ 3; 32; 1; 45; 1; 36 ]
      | (* -zat *) [ 32 :: [ 1 :: [ 46 :: _ ] ] ]
                    (* -tri.mzat -catvaari.mzat -pa ncaazat *)
                    → warn stem "a␣Fem"
      | (* zata *) [ 1; 32; 1; 46 ] (* actually also Mas *)
      | (* dvizata *) [ 1; 32; 1; 46; 3; 45; 34 ]
```

```
        | (* sahasra *) [ 1; 43; 48; 1; 49; 1; 48 ] → warn stem "a Neu"
        | (* adhika *) [ 1; 17; 3; 35; 1 ] → warn stem "an adj"
        | _ → report stem g
        ]
      ]
  | Ind k → let form = mirror (terminal_form stem) in
              enter e [ Indecl k form ]
  ] with
  [ Failure s → do
      { output_string stdout "\n\n"
      ; flush stdout
      ; Printf.eprintf "Declension error for stem %s in entry %s\n%!"
                       (Canon.decode (mirror stem)) e
      ; failwith s
      }
  ]
;
(* Main procedure, invoked by compute_decls and fake_compute_decls with entry e :string,
d : declension_class which gives the gender g, s : skt is a stem of e, p :string is a participle
name or "" *)
value compute_decls_stem e (s, d) p =
  let rstem = revstem s in (* remove homonym index if any *)
  compute_nouns_stem_form e rstem d p
  (* Only the normalized form is stored and thus extra sandhi rules such as m+n-¿nn must
be added in Compile_sandhi *)
;
(* We keep entries with only feminine stems, in order to put them in Iic *)
value extract_fem_stems = extract_rec []
  where rec extract_rec acc = fun
    [ [] → acc
    | [ (s, Gender Fem) :: rest ] → extract_rec [ s :: acc ] rest
    | [ _ :: rest ] → [] (* Beware: ind subentry of fstem will kill its iic *)
    ]
;
value enter_iic_stem entry (stem : string) = do
  { enter1 entry (Bare Noun (mirror (finalize (revstem stem)))) (* horror *)
  ; match entry with (* extra forms *)
    [ "viz#2" → enter1 entry (Bare Noun (normal_stem entry)) (* vizpati *)
    | _ → ()
    ]
```

```
  }
;
(* called by Make_nouns.genders_to_nouns twice, for nouns and then ifcs *)
value compute_decls word genders =
  let entry = Canon.decode word in
  let compute_gender gen = compute_decls_stem entry gen ""
                               (* we do not know the morphology *) in do
  { try List.iter compute_gender genders
    with [ Report s → print_report s
         | Failure s → print_report ("Anomaly: " ^ entry ^ " " ^ s)
         ]
  ; match extract_fem_stems genders with
    [ [] → ()
    | fem_stems → iter (enter_iic_stem entry) fem_stems
    ]
  }
;
value iic_indecl = (* should be lexicalized *)
(* indeclinable stems used as iic of non-avyayiibhaava cpd *)
  [ "atra" (* atrabhavat *)
  ; "adhas" (* adha.hzaakha adhazcara.nam *)
  ; "antar" (* antarafga *)
  ; "arvaak" (* arvaakkaalika *)
  ; "alam" (* (gati) ala.mk.rta *)
  ; "iti" (* ityukta *)
  ; "upari" (* uparicara *)
  ; "ubhayatas" (* ubhayata.hsasya - tasil *)
  ; "evam" (* eva.mvid *)
  ; "tatra" (* tatrabhavat *)
  ; "divaa" (* divaanidraa *)
  ; "na~n" (* na nvaada *)
  ; "naanaa" (* naanaaruupa *)
  ; "param" (* para.mtapa *)
  ; "punar" (* punarukta *)
  ; "puras" (* (gati) pura.hstha *)
  ; "praayas" (* praayazcitta *)
  ; "mithyaa" (* mithyaak.rta *)
  ; "tathaa" (* tathaagata *)
  ; "yathaa" (* yathaanirdi.s.ta *)
  ; "vinaa" (* vinaabhava *)
```

```
  ; "satraa" (* satraajit *)
  ; "sadaa" (* sadaananda *)
  ; "saha#2" (* problematic – overgenerates *)
  ; "saak.saat"
  ; "saaci"
  ; "svayam"
  ]
;
(* Feminine stems iic for productive adjectives *)
(* This is a generic weakness, to be remedied. *)
(* Generative stems are not inspected for feminine stems *)
(* attested as substantives, and thus incurring a feminine iic stem. *)
(* This concerns privative compounds and participles. *)
value iicf_extra =
  [ "abalaa" (* a-bala with fem abalaa *)
  ; "kaantaa" (* kaanta pp *)
  ]
;
value iic_avya =
(* indeclinable stems used as iic of avyayiibhaava cpd *)
  [ "ati" (* atikambalam atinidram atyaasam *)
  ; "adhas" (* adhazcara.nam *)
  ; "adhi" (* adhipaa.ni adhistri adhihari adhihasti adhyaatmam *)
  ; "abhi" (* abhyagni abhipuurvam *)
  ; "anu" (* anujye.s.tham anuk.sa.nam anugu anu.svadham (.) *)
  ; "antaraa" (* antaraaz.fgam *)
  ; "apa"
(*; "aa" – overgenerates *)
  ; "iti"
  ; "upa" (* upakumbham upak.r.s.naat upagafgam upanadam upaagni *)
  ; "upari" (* uparibhuumi *)
  ; "dus" (* durbhik.sam *)
  ; "nis" (* nirmak.sikam *)
  ; "pari"
  ; "prati" (* pratyaham prativar.sam *)
  ; "paare" (* paaregafgam *)
  ; "praak"
  ; "bahir" (* bahirgraamam *)
  ; "madhye" (* madhyegafgam madhyejalaat *)
  ; "yathaa" (* yathaazakti yathaakaamam yathaagatam yathaanyaasam yathaav.rddham
```

```
yathaazraddham yathaasthaanam ... *)
  ; "yaavat" (* yaavacchakyam yaavajjiivam P{2,1,8} *)
  ; "sa#1" (* sak.satram sacakram sat.r.nam saak.siptam saak.saat *)
  ; "su#1"
(* "dvyaha" (* dvyahatar.sam (adv+namul) dvyahaatyaasam (adv) *) *)
  ]
(* Avyayiibhaava compounds not recognized as such: those should not be marked as avya (and thus skipped) in the lexicon 1. missing iic: iic aa-: aakar.namuulam aacandram aadvaadazam aamuulam aasa.msaaram aasamudram iic. a-yathaa-: ayathaamaatram iic. ubhayatas-: ubhayata.hkaalam iic. dvyaha-: dvyahatar.sam dvyahaatyaasam iic. para-: parazvas iic. paras-: parovaram iic. uccais-: uccai.hzabdam iic. mithyaa-: mithyaaj naanam 2. missing ifc: ifc. -prati: sukhaprati zaakaprati ifc. kridanta yathaav.rddham yathe.s.tam yaavacchakyam (TODO) ifc. also pv-kridanta (-aagata) yathaagatam ifc. yatham: yathaayatham 3. misc: ti.s.thadgu anu.svadham var.sabhogye.na (retroflexion) *)
;

value enter_iic entry =
  enter1 entry (Bare Noun (normal_stem entry)) (* stripped entry *)
  (* NB This assumes the iic to be the entry stem - unsafe *)
;
value compute_extra_iic = iter enter_iic
;
(* Glitch to allow Cvi construction to kridanta entries, even though Inflected.enter_form called from Parts does not allow it. *)
(* Incomplete for compounds anyway: "si.mh'avyaaghraami.siik.r" *)
value iiv_krids =
  [ "gupta"
  ; "yuddha"
  ; "lak.sya"
  ; "vibhinna"
  ; "vyakta"
  ; "ziir.na"
  ; "zuddha"
  ; "spa.s.ta"
  ; "saaci" (* ind *)
  ]
;
value enter_iiv entry =
  match revstem entry with
  [ [ _ :: stem ] -> enter1 entry (Cvi (wrap stem 4 (* ii *)))
```

```
  | _ → failwith "wrong␣stem␣enter_iiv"
  ]
;
value compute_extra_iiv = iter enter_iiv
;
(* Gati forms used with auxiliary verbs, like Iiv – form Absya *)
value gatis = (* Gsaak.sat Wh§1092 *)
  [ "saak.saat" (* Pan1,4,74 in the sense of cvi - becoming Wh§1078a *)
  ; "mithyaa" (* Gsaak.saat *)
  ; "cintaa"
  ; "bhadraa"
  ; "locanaa"
  ; "vibhaa.saa" (* sampatkaa ? *)
  ; "aasthaa"
  ; "amaa"
  ; "zraddhaa" (* praajaryaa praajaruhaa viijaryaa viijaruhaa sa.msaryaa *)
  ; "arthe"
  ; "lava.nam"
  ; "u.s.nam"
  ; "ziitam"
  ; "udakam"
  ; "aardram"
  ; "agnau"
  ; "vaze" (* vikampate vihasane prahasane pratapane *)
  ; "praadur" (* Wh§1078 *)
  ; "namas" (* namask.rtya Wh§1092a *)
  ; "aavis" (* namask.rtya Wh§1078 *)
  ; "urasi" (* Pan1,4,75 in the sense of anatyaadhaana cf Sharma *)
  ; "manasi"
  ; "anye" (* Pan1,4,76 id *)
  ; "pade"
  ; "madhye"
  ; "nivacane"
  ; "haste" (* Pan1,4,77 upayamana (mariage) *)
  ; "paa.nau"
  ; "svayam"
  ; "uurii" (* Pan1,4,61 Guurii uuriik.rtya but Wh§1094b says uriik.r *)
  (* other Guurii: yadurii,urarii,yadurarii,paapii,laalii,aattaalii,vetaalii, dhuurii,zakalii,sa.mzaklii,phalu
etc. ignored *)
  (* vinaa Wh§1078a ignored *)
```

(∗ The following gatis are treated as preverbs, since they apply to roots other than the 3 auxiliaries: ; "astam" (∗ gam,i Pan1,4,68 asta.mgatya Wh§1092b ∗) ; "puras" (∗ k.r1,dhaa1,i Pan1,4,67 Wh§1078 ∗) ; "tiras" (∗ k.r1,dhaa1 Pan1,4,71-72 Wh§1078 ∗) ; "alam" (∗ ala.mk.rtya Pan1,4,64 Wh§1078a ∗) ; "bahis" (∗ k.r1 bhuu1 Wh§1078a ∗) ; "zrat" (∗ dhaa1 Wh§1079 ∗) ∗)
(∗ Not taken into account at present: sat/asat satk.rtya Pan1,4,63 antar antarhatya Pan1,4,65 ka.ne/manas ka.nehatya Pan1,4,66 accha acchagatya acchodya Pan1,4,69 Wh§1078 adas ada.hk.rtya Pan1,4,70 also ignored onomatopeae pa.tapa.taakaroti etc. .daac Pan5,4,57-67 ∗)
  ]
;
value enter_gati gati = (∗ assumes gati has lexical entry ∗)
  let *stem* = *normal_stem gati* in
  enter1 gati (*Cvi stem*)
;
value enter_iiy entry =
  enter1 entry (*Avyayai* (*normal_stem entry*)) (∗ stripped entry ∗)
;
(∗ Tasils are treated as adverbs. Here are the lexicalized ones: Whitney§1098 First tasils of pronouns, not needed if lexicalised ; enter1 "tad" (Indecl Tas (code "tatas")) (∗ tasil on tad **P**{5,3,7} ∗) ; enter1 "ya#1" (Indecl Tas (code "yatas")) (∗ tasil on ya **P**{5,3,7} ∗) ; enter1 "ku#1" (Indecl Tas (code "kutas")) (∗ tasil on ku **P**{5,3,7-8} ∗) ; enter1 "abhi" (Indecl Tas (code "abhitas")) (∗ tasil on abhi **P**{5,3,9} ∗) ; enter1 "pari" (Indecl Tas (code "paritas")) (∗ tasil on pari **P**{5,3,9} ∗) ; enter1 "anti" (Indecl Tas (code "antitas")) (∗ tasil on pn **P**{5,3,7} ∗) ; enter1 "ayam" (Indecl Tas (code "atas")) (∗ tasil on ayam **P**{5,3,5} ∗) ; enter1 "idam" (Indecl Tas (code "itas")) (∗ tasil on idam id ∗) ; enter1 "adas" (Indecl Tas (code "amutas")) (∗ id ∗) ; enter1 "anya" (Indecl Tas (code "anyatas")) (∗ id ∗) ; enter1 "para" (Indecl Tas (code "paratas")) (∗ id ∗) ; enter1 "vizva" (Indecl Tas (code "vizvatas")) (∗ id ∗) ; enter1 "puurva" (Indecl Tas (code "puurvatas")) (∗ id ∗) ; enter1 "sarva" (Indecl Tas (code "sarvatas")) (∗ id ∗) ; enter1 "eka" (Indecl Tas (code "ekatas")) (∗ id ∗) ; enter1 "sva" (Indecl Tas (code "svatas")) (∗ id ∗) ; enter1 "anyatara" (Indecl Tas (code "anyataratas")) (∗ id ∗) ; enter1 "dak.si.na" (Indecl Tas (code "dak.si.natas")) (∗ id ∗) ; enter1 "avara" (Indecl Tas (code "avaratas")) (∗ **P**{5,3,29} ∗) ; enter1 "uttara#1" (Indecl Tas (code "uttaratas")) (∗ on pn **P**{5,3,7} ? ∗) ; enter1 "ubhaya" (Indecl Tas (code "ubhayatas")) (∗ on pn **P**{5,3,7} ? ∗) ∗)
value tasil_extra () = do (∗ add non-generative tasils - ad-hoc ∗)
  { *enter1* "aze.sa" (*Indecl Tas* (*code* "aze.satas")) (∗ tasil on privative cpd ∗)
  ; *enter1* "ekaruupa" (*Indecl Tas* (*code* "ekaruupatas")) (∗ tasil on cpd ∗)
  ; *enter1* "d.r.s.taanta" (*Indecl Tas* (*code* "d.r.s.taantatas"))(∗ tasil on cpd ∗)
  ; *enter1* "guruvaktra" (*Indecl Tas* (*code* "guruvaktratas")) (∗ id ∗)

```
  ; enter1 "paramaartha" (Indecl Tas (code "paramaarthatas")) (* id *)
  ; enter1 "praagbhaava" (Indecl Tas (code "praagbhaavatas")) (* id *)
  ; enter1 "svaravar.na" (Indecl Tas (code "svaravar.natas")) (* id *)
  ; enter1 "bhasad" (Indecl Tas (code "bhasattas")) (* tasil on consonant stem *)
(*; enter1 "nas#2" (Indecl Tas (code "nastas")) - idem but lexicalized *)
  ; enter1 "yad.rcchaa" (Indecl Tas (code "yad.rcchaatas")) (* tasil on fstem *)
(* NB bhii.smadro.napramukhatas BhG1,25 treated in enter_extra_ifcs below *)
  }
;
(* Supplementary forms - called by Make_nouns.genders_to_nouns with argument iic_stems
contents of iic_stems_file dumped from Subst.iic_stems built by calling Subst.record_iic for
iic only entries. *)
value compute_extra iic_only_stems = do
  { enter1 "maas" (* Siddhaanta kaumudii *) decl where decl = Declined Noun Mas [ (Dual, [ (Ins, code
  ; enter1 "yuu.sa" (* Siddhaanta kaumudii *) decl
     where decl = Declined Noun Mas [ (Plural, [ (Loc, code "yuu.h.su") ]) ]
  ; enter1 "avanam" (Cvi (code "avanamii")) (* exception *)
 (* For the moment, computed as form of n.r but skipped ; enter1 "nara" decl (× overgenerates badly ! ×
) where decl = Declined Noun Mas [ (Singular, [ (Nom, code "naa") ]) ] *)
  ; enter1 "nara" decl
     where decl = Declined Noun Mas [ (Plural, [ (Gen, code "n.rr.naam") ]) ]
  ; enter1 "nara" decl (* P{6,4,6} *)
     where decl = Declined Noun Mas [ (Plural, [ (Gen, code "n.r.naam") ]) ]
  ; enter1 "nara" decl
     where decl = Bare Noun (code "n.r")
  ; enter1 "bhagavat" decl (* archaic vocative bhagavas *)
     where decl = Declined Noun Mas [ (Singular, [ (Voc, code "bhagavas") ]) ]
  ; enter1 "tak.san" decl (* P{6,4,9} *)
     where decl = Declined Noun Mas [ (Singular, [ (Acc, code "tak.sa.nam") ]) ]
  ; enter1 "bhuuman" decl (* dhruvaaya bhuumaaya nama.h *)
     where decl = Declined Noun Mas [ (Singular, [ (Dat, code "bhuumaaya") ]) ]
  ; enter1 "sudhii" (* Monier *) decl
     where decl = Declined Noun Mas [ (Singular, [ (Nom, code "sudhi") ]) ]
  ; enter1 "viz#2" (* Vedic Whitney§218a *) decl
     where decl = Declined Noun Fem [ (Plural, [ (Loc, code "vik.su") ]) ]
  ; iter enter_iiy iic_avya
  ; enter1 "giri" (Avyayaf (code "giram")) (* P{5,4,112} upagiram *)
  ; iter enter_gati gatis
  ; tasil_extra ()
  ; compute_extra_iic iic_indecl (* antar *)
```

```
  ; compute_extra_iic iic_only_stems (* aajaanu etc. *)
  ; compute_extra_iic iicf_extra (* abalaa etc. *)
  ; compute_extra_iiv iiv_krids (* zuddhii *)
  ; enter1 "u" (* Vedic *) decl
    where decl = Indecl Interj [ 5 ] (* u *)
    (* Unplugged presently because of overgeneration ; compute_extra_iic gen_prefixes ;
compute_extra_ifc bahu_suffixes eg Fem -padaa for meter formation *)
  }
;
(* Used in Make_nouns in Phase 2 ifc pass *)
value enter_extra_ifcs () = do (* archaic retroflexion in cpds P{8,4,13} *)
  { let entry = "bhogya" in (* var.sabhogye.na Meghaduuta 1b *)
        let ins_sg = [ (Singular, [ (Ins, code "bhogye.na") ]) ]
        and gen_pl = [ (Plural, [ (Gen, code "bhogyaa.naam") ]) ] in do
        { enter1 entry (Declined Noun Mas ins_sg)
        ; enter1 entry (Declined Noun Mas gen_pl)
        ; enter1 entry (Declined Noun Neu ins_sg)
        ; enter1 entry (Declined Noun Neu gen_pl)
        ; enter1 entry (Declined Noun Fem gen_pl)
        }
  ; let entry = "yogin" in (* pu.spayogi.nah Renou yogi-fleur? *)
        let form = code "yogi.nas" in do
        { enter1 entry (Declined Noun Mas [ (Singular, [ (Gen, form) ]) ])
        ; enter1 entry (Declined Noun Mas [ (Singular, [ (Abl, form) ]) ])
        ; enter1 entry (Declined Noun Neu [ (Singular, [ (Gen, form) ]) ])
        ; enter1 entry (Declined Noun Neu [ (Singular, [ (Abl, form) ]) ])
        ; enter1 entry (Declined Noun Mas [ (Plural, [ (Nom, form) ]) ])
        ; enter1 entry (Declined Noun Mas [ (Plural, [ (Acc, form) ]) ])
        }
  ; let entry = "yuga" in do
      { let form = code "yugaa.ni" in do (* v.r.sabhayugaa.ni vastrayugaa.ni *)
        { enter1 entry (Declined Noun Neu [ (Plural, [ (Nom, form) ]) ])
        ; enter1 entry (Declined Noun Neu [ (Plural, [ (Acc, form) ]) ])
        }
      ; let form = code "yuge.na" in do (* vastrayuge.na kharayuge.na *)
        { enter1 entry (Declined Noun Neu [ (Singular, [ (Ins, form) ]) ])
        ; enter1 entry (Declined Noun Mas [ (Singular, [ (Ins, form) ]) ])(*bahu*)
        }
      } (* NB "vastrayugi.nas", "vastrayugi.nau" etc. OK since stem autonomous *)
  ; let entry = "kaamin" in (* svargakaami.nau *)
```

```
        let form = code "kaami.nau" in do
        { enter1 entry (Declined Noun Mas [ (Dual, [ (Nom, form) ]) ])
        ; enter1 entry (Declined Noun Mas [ (Dual, [ (Acc, form) ]) ])
        ; enter1 entry (Declined Noun Mas [ (Dual, [ (Voc, form) ]) ])
        }
  ; let entry = "gaamin" in (* v.r.sagaami.nau *)
        let form = code "gaami.nau" in do
        { enter1 entry (Declined Noun Mas [ (Dual, [ (Nom, form) ]) ])
        ; enter1 entry (Declined Noun Mas [ (Dual, [ (Acc, form) ]) ])
        ; enter1 entry (Declined Noun Mas [ (Dual, [ (Voc, form) ]) ])
        }
  ; let entry = "dh.rt" in (* ruupadh.rk *)
        let form = code "dh.rk" in do
        { enter1 entry (Declined Noun Mas [ (Singular, [ (Nom, form) ]) ])
        ; enter1 entry (Declined Noun Neu [ (Singular, [ (Nom, form) ]) ])
        }
   (* Now ad-hoc provision for specific adverbs usable as ifcs *)
   ; let entry = "k.rtvas" in
        enter_ind_ifc entry (Indecl Adv (code entry))
   ; let entry ="pramukha" in
        enter_ind_ifc entry (Indecl Tas (code "pramukhatas")) (* ifc tasil *)
   }
;
value enter_extra_iifcs () = do
  { let entry = "ahan" in (* for -aha- like pu.nyaahavaacanam *)
     enter1 entry (Bare Noun (code "aha"))
  ; let entry = "aakyaa#2" in (* for -aakhya- like pu.nyaahavaacanam *)
     enter1 entry (Bare Noun (code "aakhya"))
  ; let entry = "senaa" in (* for zuklasenadeva.h *)
     enter1 entry (Bare Noun (code "sena"))
     (* more entries are potentially concerned - for bahus of X-Y with Y fstem *)
  }
;
(* called by Declension.emit_decls and Morpho_debug.emit_decls *)
value fake_compute_decls ((s, _ (* forget decli *)) as nmorph) part =
  let entry = s in do (* fake entry made from stem s - cheat *)
  { reset_nominal_databases ()
  ; morpho_gen.val := False
  ; compute_decls_stem entry nmorph part
  ; nominal_databases ()
```

```
  }
;
```

For Interface - cache management

```
open Bank_lexer;
module Gram = Camlp4.PreCast.MakeGram Bank_lexer
;
open Bank_lexer.Token;
open Skt_morph;

value full_entry = Gram.Entry.mk "full_entry"
and entry = Gram.Entry.mk "entry"
and gen = Gram.Entry.mk "gen"
;
EXTEND Gram
  full_entry :
    [ [ e = entry; g = gen → (e, g) ] ] ;
  entry :
    [ [ "["; t = TEXT; "]" → t ] ];
  gen :
    [ [ "("; t = TEXT; ")" →
      let gender_of = fun
        [ "m." → Mas
        | "f." → Fem
        | "n." → Neu
        | s → failwith ("Weird␣gender" ^ s)
        ] in
      Gender (gender_of t) ] ];
END
;
value parse_entry s =
  try Gram.parse_string full_entry Loc.ghost s with
  [ Loc.Exc_located loc e → do
      { Format.eprintf "Wrong␣input:␣%s\n,␣at␣location␣%a:@." s Loc.print loc
      ; raise e
      }
  ]
;
value update_index ic =
  try read_from_ic ic
      where rec read_from_ic ic =
```

```
          let s = input_line ic in do
          { let ((entry, gender) as eg) = parse_entry s in
            try compute_decls_stem entry eg ""
            with [ Sys_error m → print_string ("Sys_error" ^ m)
                 | _ → print_string "Wrong␣input"
                 ]
          ; read_from_ic ic
          }
  with [ End_of_file → close_in ic ]
;
(* Cache forms computation - used in Interface and Restore_caches *)
value extract_current_caches cache_txt_file = do
  { nouns.val := Deco.empty
  ; iics.val := Deco.empty
  ; morpho_gen.val := False
  ; let ic = open_in cache_txt_file in update_index ic
  ; (nouns.val, iics.val)
  }
;
```

Interface for module Verbs

```
open Skt_morph;

value compute_conjugs : Word.word → Conj_infos.root_infos → unit;
value compute_conjugs_stems : string → Conj_infos.root_infos → unit;
value compute_extra : unit → unit;
value fake_compute_conjugs : int (* pr_class *) → string (* entry *) → unit;
```

Module Verbs

Verbs defines the conjugation paradigms, and computes conjugated forms
Computed forms comprise finite verbal forms of roots, but also derived nominal forms (participles), infinitives and absolutives
Terminology. record functions will build the forms needed by Conjugation and Stemming. After change of this file, and "make releasecgi", these tables are updated. But the Reader/Parser needs a full pass of generation, with "make scratch" from Dictionary, in order to rebuild the full automata.

```
open List; (* map, length, rev *)
open Phonetics; (* vowel, homonasal, duhify, mrijify, nahify, light, nasal, gana, mult, aug, trunc_a,
*)
open Skt_morph;
open Inflected; (* Conju, Invar, Inftu, roots, enter1, morpho_gen, admits_aa *)
open Parts; (* memo_part, record_part, cau_gana, fix, fix_augment, rfix, compute_participles
*)
(* This module also uses modules List2 Word Control Canon Encode Int_sandhi and in-
terface Conj_infos *)
open Pada; (* voices_of_gana *)
```

In the grinding phase, we record for each root entry its class and its stem for 3rd present. In the declination phase, we compute the inflected forms and we record them with a pair (*entry*, *conjugs*) in verbs.rem, parts.rem, etc.

```
exception Not_attested (* No attested form *)
;
(* Present system - we give vmorph info Prim root_class pada third_conjug where third_conjug
is a word, used for checking the 3rd sg Para *)
value present = Present
and imperfect = Imperfect
and optative = Optative
and imperative = Imperative
;
(* Paradigms *)
value vpa cl = Presenta cl Present
and vpm cl = Presentm cl Present
and vpp = Presentp Present
and via cl = Presenta cl Imperfect
and vim cl = Presentm cl Imperfect
and vip = Presentp Imperfect
and voa cl = Presenta cl Optative
and vom cl = Presentm cl Optative
and vop = Presentp Optative
and vma cl = Presenta cl Imperative
and vmm cl = Presentm cl Imperative
and vmp = Presentp Imperative
and vfa = Conjug Future Active
and vfm = Conjug Future Middle
and vca = Conjug Conditional Active
and vcm = Conjug Conditional Middle
```

```
and vfp = Conjug Future Passive
and vpfa = Conjug Perfect Active
and vpfm = Conjug Perfect Middle
and vpfp = Conjug Perfect Passive
and vbena = Conjug Benedictive Active
and vbenm = Conjug Benedictive Middle
and vaa cl = Conjug (Aorist cl) Active
and vam cl = Conjug (Aorist cl) Middle
and vja cl = Conjug (Injunctive cl) Active
and vjm cl = Conjug (Injunctive cl) Middle
and vap1 = Conjug (Aorist 1) Passive (* passive of root aorist *)
and vjp1 = Conjug (Injunctive 1) Passive (* passive of root injunctive *)
;
(* Finite verbal forms of roots *)
value fpresa cl conj = (conj, vpa cl)
and fpresm cl conj = (conj, vpm cl)
and fpresp conj = (conj, vpp)
and fimpfta cl conj = (conj, via cl)
and fimpftm cl conj = (conj, vim cl)
and fimpftp conj = (conj, vip)
and fopta cl conj = (conj, voa cl)
and foptm cl conj = (conj, vom cl)
and foptp conj = (conj, vop)
and fimpera cl conj = (conj, vma cl)
and fimperm cl conj = (conj, vmm cl)
and fimperp conj = (conj, vmp)
and ffutura conj = (conj, vfa)
and ffuturm conj = (conj, vfm)
and fconda conj = (conj, vca)
and fcondm conj = (conj, vcm)
and fperfa conj = (conj, vpfa)
and fperfm conj = (conj, vpfm)
and fbenea conj = (conj, vbena)
and fbenem conj = (conj, vbenm)
and faora cl conj = (conj, vaa cl)
and faorm cl conj = (conj, vam cl)
and finja cl conj = (conj, vja cl)
and finjm cl conj = (conj, vjm cl)
and faorp1 conj = (conj, vap1)
and finjp1 conj = (conj, vjp1)
```

```
;
(* Primary finite verbal forms of roots *)
value presa cl = fpresa cl Primary
and presm cl = fpresm cl Primary
and impfta cl = fimpfta cl Primary
and impftm cl = fimpftm cl Primary
and opta cl = fopta cl Primary
and optm cl = foptm cl Primary
and impera cl = fimpera cl Primary
and imperm cl = fimperm cl Primary
and futura = ffutura Primary
and futurm = ffuturm Primary
and perfa = fperfa Primary
and perfm = fperfm Primary
and aora cl = faora cl Primary
and aorm cl = faorm cl Primary
and aorp1 = faorp1 Primary
and benea = fbenea Primary
and benem = fbenem Primary
and inja cl = finja cl Primary
and injm cl = finjm cl Primary
and injp1 = finjp1 Primary
;
(* Participial forms *)
value pra k = Ppra k
and prm k = Pprm k
and prp = Pprp
and pfta = Ppfta
and pftm = Ppftm
and futa = Pfuta
and futm = Pfutm
(* Also in Part: Ppp, Pppa, Ger=Pfut Passive, Inf *)
;
(* Verbal forms of roots *)
value vppra k conj = (conj, pra k)
and vpprm k conj = (conj, prm k)
and vppfta conj = (conj, pfta)
and vppftm conj = (conj, pftm)
and vpfuta conj = (conj, futa)
and vpfutm conj = (conj, futm)
```

```
and vpprp conj = (conj, prp)
(* Also in Part: Ppp, Pppa, Ger=Pfut Passive, Inf *)
;
(* Verbal forms of roots *)
value ppra k = vppra k Primary
and pprm k = vpprm k Primary
and ppfta = vppfta Primary
and ppftm = vppftm Primary
and pfuta = vpfuta Primary
and pfutm = vpfutm Primary
and pprp = vpprp Primary
;
(* Derived verbal forms *)
value causa = fpresa cau_gana Causative
and pcausa = vppra cau_gana Causative
and causm = fpresm cau_gana Causative
and pcausm = vpprm cau_gana Causative
and causp = fpresp Causative
and causfa = ffutura Causative
and pcausfa = vpfuta Causative
and causfm = ffuturm Causative
and pcausfm = vpfutm Causative
and caaora cl = faora cl Causative
and caaorm cl = faorm cl Causative
and intensa = fpresa int_gana Intensive
and pinta = vppra int_gana Intensive
and intensm = fpresm int_gana Intensive
and pintm = vpprm int_gana Intensive
and desida = fpresa des_gana Desiderative
and pdesa = vppra des_gana Desiderative
and desidm = fpresm des_gana Desiderative
and pdesm = vpprm des_gana Desiderative
and despfa = fperfa Desiderative
and despfm = fperfm Desiderative
;
value intimpfta = fimpfta int_gana Intensive
and intopta = fopta int_gana Intensive
and intimpera = fimpera int_gana Intensive
;
value code = Encode.code_string (* normalized *)
```

```
and revcode = Encode.rev_code_string (* reversed *)
and revstem = Encode.rev_stem (* stripped of homo counter *)
;
(* Checking consistency of computed form with witness from lexicon. *)
(* Discrepancies are noted on a warnings log, written on stderr. *)
(* NB currently log dumped in (D)STAT/warnings.txt by "make roots.rem". *)
value emit_warning s =
  if morpho_gen.val then output_string stderr (s ^ "\n") else ((* cgi *))
;
value report entry gana listed computed =
  let s1 = Canon.decode computed
  and s2 = Canon.decode listed in
  let message = entry ^ " [" ^ string_of_int gana ^ "] wrong 3rd pr "
                      ^ s1 ^ " for " ^ s2 in
  emit_warning message
;
(* third is attested from Dico, form is generated by morphology *)
value check entry gana third ((_,form) as res) = do
  { if third = [] (* no checking *) ∨ third = form then ()
    else match entry with
         [ "a~nc" | "kalu.s" | "kram" | "grah" | "cam" | "tul" | "t.rr"
         | "manth" | "v.r#1" | "huu" | "putr"
           (* these roots have multiple ga.nas, i.e. different entries in DP *)
             → () (* 2 forms - avoids double warning *)
         | _ → report entry gana third form
         ]
  ; res (* Note that the computed form has priority over the listed one. *)
        (* Log inspection leads to correction of either Dico or Verbs. *)
  }
;
value warning message =
  failwith (message ^ "\n")
and error_empty n =
  failwith ("empty stem " ^ string_of_int n)
and error_suffix n =
  failwith ("empty suffix " ^ string_of_int n)
and error_vowel n =
  failwith ("no vowel in root " ^ string_of_int n)
;
```

**** Conjugation of verbal stems ****

Suffixing uses *Int_sandhi.sandhi* (through Parts.fix) for thematic conjugation and conjugation of roots of ganas 5,7,8 and 9, and the following sandhi function for athematic conjugation of roots of ganas 2 and 3 (through respectively fix2 and fix3w).
This sandhi restores initial aspiration if final one is lost – Gonda§4 note. This concerns root syllables with initial g- d- b- and final -gh -dh -bh -h where aspiration is shifted forwards. The corresponding problem is dealt in *Nouns.build_root* by *Phonetics.finalize*, so there is some redundancy. It is related to Grassmann's law and Bartholomae's law in IE linguistics.

```
value sandhi revstem wsuff =
  let aspirate w = match w with
    [ [] → w
    | [ c :: rest ] → match c with (* uses arithmetic encoding for aspiration *)
        [ 19 | 34 | 39 (* g d b *) → [ c + 1 :: rest ] (* aspiration *)
        | _ → w
        ]
    ]
  and lost = match wsuff with
    [ [] → False
    | [ c :: _ ] → match c with (* Gonda§4 note *)
      [ 48 (* s *) → (* 32 — 33 — 35 — 49 (* t th dh h *) ? *)
         match revstem with
         [ [ 20 :: _ ] | [ 35 :: _ ] | [ 40 :: _ ] | [ 49 :: _ ]
            (* gh dh bh h *)
         | [ 149 :: _ ] | [ 249 :: _ ]
            (* h' h" *)
               → True
         | _ → False
         ]
      | _ → False
      ]
    ]
  and result = Int_sandhi.int_sandhi revstem wsuff in
  if lost then aspirate result else result
;
```

Theoretical general conjugational scheme : Given the stem value, let conjug person suff = (person,fix stem suff) (*fix_augment* instead of *fix* for preterit) We enter in the roots lexicon an entry: (*Conju verbal* [(*Singular*, [*conjug First suff_s1* ; *conjug Second suff_s2* ; *conjug Third suff_s3*
Remark. More general patterns such as above could have been used, in Paninian style, but at the price of complicating internal sandhi, for instance for dropping final a of the stem in *conjug First suff_s1* (Goldman§4.22). Here instead of st-a+e -¿ st-e we compute st-e with

a shortened stem. Similarly st-a+ete -¿ st-ete -¿ in Dual, see *compute_thematic_presentm* etc.
Returns the reverse of *int_sandhi* of reversed prefix and reversed stem
But *int_sandhi* may provoke too much retroflexion, such as *si.sarti instead of sisarti for root s.r in redup3 below. Same pb to avoid *pu.sphora as perfect of sphur, instead of pusphora. Thus need of the boolean argument retr in the following:

```
value revaffix retr revpref rstem =
  let glue = if retr then Int_sandhi.int_sandhi else List2.unstack in
  rev (glue revpref (rev rstem)) ;
(* Computation of verbal stems from root *)

value final_guna v w = List2.unstack (guna v) w
and final_vriddhi v w = List2.unstack (vriddhi v) w
;
(* Strong form of reversed stem *)
value strong = fun (* follows Phonetics.gunify *)
  [ [] → error_empty 1
  | [ v :: rest ] when vowel v → final_guna v rest
  | [ c :: [ v :: rest ] ] when short_vowel v → [ c :: final_guna v rest ]
  | s → s
  ]
;
(* Lengthened form of reversed stem *)
value lengthened = fun
  [ [] → error_empty 2
  | [ v :: rest ] when vowel v → final_vriddhi v rest
  | [ c :: [ v :: rest ] ] when short_vowel v → [ c :: final_vriddhi v rest ]
  | s → s
  ]
;
value strengthen_10 rstem = fun
  [ "m.r.d" | "sp.rh" → rstem (* exceptions with weak stem *)
  | "k.sal" → lengthened rstem (* v.rddhi *)
  | _ → strong rstem (* guna *)
  ]
;
(* .r -¿ raa (Whitney§882a, Macdonell§144.4) *)
value long_metathesis = fun (* .r penultimate -¿ raa *)
  [ [ c :: [ 7 (* .r *) :: rest ] ] → [ c :: [ 2 :: [ 43 :: rest ] ] ]
  | _ → failwith "long_metathesis"
```

```
  ]
;
(* truncates an rstem eg bh.rjj -¿ bh.rj *)
value truncate = fun
  [ [] → error_empty 3
  | [ _ :: r ] → r
  ]
;
value strong_stem entry rstem = (* rstem = revstem entry *)
  match entry with
    [ "am" → revcode "amii" (* amiiti *)
    | "dah#1" | "dih" | "duh#1" | "druh#1" | "muh" | "snih#1" | "snuh#1"
                → duhify (strong rstem)
    | "nah" → nahify (strong rstem)
    | "m.rj" → mrijify (revcode "maarj") (* maar.s.ti long_metathesis *)
    | "yaj#1" | "vraj" | "raaj#1" | "bhraaj" | "s.rj#1"
                → mrijify (strong rstem)
    | "bh.rjj" → mrijify (strong (truncate rstem)) (* bh.rsj Pan8,2,29 *)
    | "nij" → revcode "ni~nj" (* nasalisation for gana 2 *)
    | "zrath" → revcode "zranth"
    | "diiv#1" → revcode "dev"
    | _ → strong rstem
    ]
;
value weak_stem entry rstem = (* rstem = revstem entry *)
  match entry with
    [ "dah#1" | "dih" | "duh#1" | "druh#1" | "muh" | "snih#1" | "snuh#1"
                → duhify rstem
    | "nah" → nahify rstem
    | "m.rj" | "yaj#1" | "vraj" | "raaj#1" | "bhraaj" | "s.rj#1"
                → mrijify rstem
    | "bh.rjj" → mrijify (truncate rstem)
    | "nij" → revcode "ni~nj" (* nasalisation *)
    | "vaz" → revcode "uz" (* but not vac ! *)
    | "zaas" → revcode "zi.s"
    | "myak.s" → revcode "mik.s"
    | _ → rstem
    ]
;
(* samprasaara.na correction - weak strong and long rev stem words of root. *)
```

```
(* Concerns 4 roots, lexicalized under their strong rather than weak stem. *)
(* Beware. The sampra correction must be effected separately when weak_stem and strong_stem
are invoked directly, rather than as components of stems. *)
value stems root =
  let rstem = revstem root in
  let sampra substitute =
       let lstem = lengthened rstem in
       (revstem substitute, rstem, lstem) in
  match root with (* This shows what ought to be the root name, its weak form *)
      [ "grah" → sampra "g.rh" (* P{6,1,15} *)
      | "vyadh" → sampra "vidh" (* P{6,1,15} *)
      | "spardh" → sampra "sp.rdh"
      | "svap" → sampra "sup" (* P{6,1,15} *)
      (* note "vac", "yaj" etc not concerned although having samprasaara.na *)
      | _ → let weak = weak_stem root rstem
             and strong = strong_stem root rstem in
             let long = lengthened weak in
             (weak, strong, long)
      ]
;
value drop_penultimate_nasal = fun
  [ [ c :: [ n :: s ] ] when nasal n → [ c :: s ]
  | _ → failwith "No␣penultimate␣nasal"
  ]
;
value passive_stem entry rstem = (* Panini yak (k : no guna, samprasaara.na) *)
  let weak = match entry with
  (* weak same as first component of stems, except praz vac etc and bh.rjj *)
    [ "dah#1" | "dih" | "duh#1" | "druh#1" | "muh" | "snih#1" | "snuh#1"
               → duhify rstem
    | "nah" → nahify rstem
    | "m.rj" | "vraj" | "raaj#1" | "bhraaj" | "s.rj#1" | "bh.rjj"
               → mrijify rstem
    | "yaj#1" → mrijify (revcode "ij") (* samprasaara.na ya-x →i-x P{6,1,15} *)
    | "vyadh" → revcode "vidh" (* id *)
    | "grah" → revcode "g.rh" (* samprasaara.na ra-x →.r-x P{6,1,16} *)
    | "vrazc" → revcode "v.rzc" (* id *)
    | "praz" → revcode "p.rcch" (* id *)
    | "svap" → revcode "sup" (* samprasaara.na va-x →u-x P{6,1,15} *)
    | "vaz" | "vac" | "vap" | "vap#1" | "vap#2" | "vad" | "vas#1" | "vas#4"
```

```
| "vah#1" (* idem - specific code for va-x roots *)
           → match rstem with
             [ [ 48 :: _ ] → [ 47 ; 5 (* u *) ] (* vas →u.s *)
             | [ c :: _ ] → [ c ; 5 (* u *) ] (* va-x →u-x *)
             | [] → failwith "Anomalous passive_stem"
             ]
| "vaa#3" → revcode "uu" (* P{6,1,15} *)
| "zaas" → revcode "zi.s" (* ambiguous zi.s.ta, zi.syate *)
| "zii#1" → revcode "zay" (* P{7,4,22} *)
| "pyaa" → revcode "pyaay" (* pyaa=pyai *)
| "indh" | "und" | "umbh" | "gumph" | "granth" | "da.mz" | "dhva.ms"
| "bandh" | "bhra.mz" | "za.ms" | "zrambh" | "skambh" | "skand"
  (* above roots have penultimate nasal and do not have i_it marker *)
| "ba.mh" | "ma.mh" | "manth" | "stambh"
  (* these four roots are listed in dhatupathas as bahi, mahi, mathi, stabhi and thus
appear here even though they admit i_it marker *)
           → drop_penultimate_nasal rstem
| _ → match rstem with
      (* -a nc -aa nc va nc a nj sa nj drop_penultimate_nasal *)
      (* doubt for pi nj and gu nj since they admit i_it marker *)
      [ [ 22 :: [ 26 :: r ] ] (* - nc *) → [ 22 :: r ] (* -ac *)
      | [ 24 :: [ 26 :: r ] ] (* - nj *) → [ 24 :: r ] (* -aj *)
      | w → w
      ]
] in
match weak with
[ [ c :: rst ] → match c with
    [ 2 (* aa *) → match rst with
        [ [ 42 (* y *) ] (* yaa1 *)
        | [ 42 (* y *); 18 (* kh *) ] (* khyaa *)
        | [ 42 (* y *); 35 (* dh *) ] (* dhyaa *) → weak
        | [ 42 (* y *) :: r ] → [ 4 (* ii *) :: r ] (* ziiyate stiiyate *)
        | _ → match entry with
          [ "j~naa#1" | "bhaa#1" | "mnaa" | "laa" | "zaa" | "haa#2"
              → weak
          | _ → [ 4 (* ii *) :: rst ]
          ]
        ]
    | 3 (* i *) → [ 4 (* ii *) :: rst ]
    | 5 (* u *) → match entry with
```

```
            [ "k.su" | "k.s.nu" | "plu" | "sru" → weak
            | _ → [ 6 (* uu *) :: rst ]
            ]
        | 7 (* .r *) → match rst with
            [ [ _ ] → [ 3 :: [ 43 :: rst ] ] (* ri- *)
            | _ (* 0 or 2 consonants *) → [ 43 :: [ 1 :: rst ] ] (* ar- *)
            ]
        | 8 (* .rr *) → match rst with
            [ [ d :: _ ] →
              if labial d then [ 43 :: [ 6 :: rst ] ] (* puuryate *)
                          else [ 43 :: [ 4 :: rst ] ] (* kiiryate ziiryate *)
            | _ → error_empty 4
            ]
        | _ → if c > 9 ∧ c < 14 (* e ai o au *) then match entry with
            [ "dhyai" → [ 2 :: rst ] (* dhyaa in Dico *)
            | "hve" → revcode "huu" (* huu in Dico, just for convenience *)
            | _ → [ 4 (* ii *) :: rst ]
            ]
              else weak
        ]
    | [] → error_empty 5
    ]
;
(* Reduplication for third class present: redup3 takes the root string and its (reversed) stem
word, and returns a triple (s, w, b) where s is the (reversed) strong stem word, w is the
(reversed) weak stem word, b is a boolean flag for special aa roots *)
value redup3 entry rstem =
  match mirror rstem with
    [ [] → failwith "Empty␣root"
    | [ 7 (* .r *) ] → (* Whitney§643d *) (revstem "iyar",revstem "iy.r",False)
    | [ c1 :: r ] → if vowel c1 then failwith "Attempt␣reduplicating␣vowel␣root"
                    else
      let v = lookvoy r
          where rec lookvoy = fun
            [ [] → failwith "Attempt␣to␣reduplicate␣root␣with␣no␣vowel"
            | [ c2 :: r2 ] → if vowel c2 then c2 else lookvoy r2
            ]
      and iflag = match entry with (* special flag for some aa roots *)
            [ "gaa#1" | "ghraa" | "maa#1" | "zaa" | "haa#2" → True
            | _ → False
```

```
        ]
and iflag2 = match entry with (* special flag for some other roots *)
        [ "maa#3" | "vac" | "vyac" → True
        | _ → False
        ] in
let c = if sibilant c1 then match r with
  (* c is reduplicating consonant candidate *)
              [ [] → failwith "Reduplicated␣root␣with␣no␣vowel"
              | [ c2 :: _ ] → if vowel c2 ∨ nasal c2 then c1
                                 else if stop c2 then c2
                                 else (* semivowel c2 *) c1
              ]
           else c1 in
let rv = (* rv is reduplicating vowel *)
   if entry ="v.rt#1" then 1 (* a *) else
   if rivarna v ∨ iflag ∨ iflag2 then 3 (* i *)
   else if entry ="nij" then 10 (* e *) (* Whitney says intensive! *)
   else short v (* reduplicated vowel is short *)
and rc = match c with (* rc is reduplicating consonant *)
   [ 17 | 18 (* k kh *) → 22 (* c *)
   | 19 | 20 | 49 (* g gh h *) → 24 (* j *)
   | 149 | 249 (* h' h2 *) → failwith "Weird␣root␣of␣class␣3"
   | 23 | 25 | 28 | 30 | 33 | 35 | 38 | 40 → c − 1 (* aspiration loss *)
   | _ → c
   ]
and iiflag = iflag ∨ entry = "haa#1" in
let (strong, weak) =
       if iiflag then match rstem with
           [ [ 2 :: rest ] → (rstem, [ 4 :: rest ]) (* aa →ii *)
           | _ → failwith "Anomaly␣Verbs"
           ]
       else let wstem = match entry with
              [ "daa#1" | "dhaa#1" → match rstem with
                  [ [ 2 :: rest ] → rest (* drop final aa *)
                  | _ → failwith "Anomaly␣Verbs"
                  ]
              | _ → rstem
              ] in
(strong rstem, wstem)
and glue = match entry with
```

```
          [ "s.r" → revaffix False [ rv; rc ] (* no retroflexion: sisarti *)
          | _ → revaffix True [ rv; rc ]
          ] in (glue strong, glue weak, iiflag)
    ]
;
```

Dhatupatha it markers (from AK's listing)
NB Use of these markers should progressively replace lists of exceptions

```
value aa_it = fun
  [ (* "muurch" — WRONG ? *)
    "phal" | "zvit#1" | "svid#2" | "tvar" | "dh.r.s" → True
  | _ → False
  ]
and i_it = fun (* unused but subset of set in intercalates *)
  [ "vand" | "bhand" | "mand#1" | "spand" | "indh" | "nind"
  | "nand" | "cand" | "zafk" | "iifkh" | "lafg" | "afg" | "ifg"
  | "gu~nj" | "laa~nch" | "vaa~nch" | "u~nch" | "ku.n.d" | "ma.n.d" | "ku.n.th"
  | "lu.n.th" | "kamp" | "lamb" | "stambh" | "j.rmbh" | "cumb" | "inv" | "jinv"
  | "ba.mh" | "ma.mh" | "ghu.s" | "kaafk.s" | "ra.mh" | "tvar"
  | "pi~nj" | "rud#1" | "hi.ms" | "chand" | "lafgh" → True
(* NB. other roots admitting set: "a~nc" | "an#2" | "arh" | "av" | "az#1" | "az#2" |
"as#2" | "aas#2" | "i.s#1" | "i.s#2" | "iik.s" | "ii.d" | "iiz#1" | "uc" | "umbh" |
"uuh" | ".rc#1" | ".rj" | ".rdh" | "edh" | "kafk" | "kam" | "ka.s" | "kup" | "krand"
| "krii.d" | "khan" | "khaad" | "gam" | "ghaat" | "ghuur.n" | "cit#1" | "jak.s" |
"jap" | "jalp" | "tak" | "tan#1" | "tan#2" | "tark" | "dagh" | "dabh" | "dham" |
"dhva.ms" | "dhvan" | "pa.th" | "pat#1" | "piz" | "bhaa.s" | "bhraaj" | "mad#1" |
"mlecch" | "yat#1" | "yaac" | "rak.s" | "raaj#1" | "ruc#1" | "lag" | "lap" |
"la.s" | "lok" | "loc" | "vad" | "vam" | "vaz" | "vaaz" | "vip" | "ven" | "vyath" |
"vraj" | "vrii.d" | "za.ms" | "zas" | "zaas" | "zuc#1" | "san#1" | "skhal" |
"spardh" | "sp.rh" | "sphu.t" | "svan" | "has" *)
  | _ → False
  ]
and ii_it = fun
  [ "hlaad" | "yat#1" | "cit#1" | "vas#4" | "jabh#1" | "kan" | "puuy" | "sphaa"
  | "pyaa" | "jan" | "n.rt" | "tras" | "diip" | "mad#1" | ".r.s" | "ju.s#1"
  | "vij" | "d.rbh" | "gur" | "k.rt#1" | "indh" | "und" | "v.rj" | "p.rc"
      → True
  | _ → False
  ]
and u_it = fun
```

```
  [ "sidh#2" | "a~nc" | "va~nc" | "zrambh" | "stubh" | "kam" | "cam" | "jam"
  | "kram" | ".s.thiiv" | "dhaav#1" | "gras" | "mi.s" | "p.r.s" | "v.r.s"
  | "gh.r.s" | "zas" | "za.ms" | "sra.ms" | "dhva.ms" | "v.rt" | "v.rdh#1"
  | "bhram" | "ram" | "m.rdh" | "khan" | "zaas" | "diiv#1" | "siiv" | "sidh#1"
  | "zam#1" | "tam" | "dam#1" | "zram" | "as#2" | "yas" | "jas" | "das"
  | "bhra.mz" | ".rdh" | "g.rdh" | "dambh" | "i.s#1" | "t.rd" | "tan#1"
  | "k.san" → True
  | _ → False
  ]
and uu_it = fun (* perstems P{7,2,44} *)
  [ "trap" | "k.sam" | "gaah" | "ak.s" | "tak.s" | "tvak.s" | "syand" | "k.rp"
  | "guh" | "m.rj" | "klid" | "az#1" | "vrazc" | "b.rh#2" | "v.rh" | "a~nj"
  | "kli.s" | "ta~nc" → True
  | _ → False
  ]
and o_it = fun (* these roots have ppp in -na P{8,2,45} - unused here *)
  [ "zuu" | "haa#1" | "haa#2" | "vij" | "vrazc" | "bhuj#1" | "bha~nj" | "lag"
 (* — "iir" — "und" — "k.rr" — "klid" — "k.sii" — "k.sud" — "k.svid" — "khid"
— "g.rr#1" — "glai" — "chad#1" — "chid#1" — "ch.rd" — "j.rr" — ".dii" —
"tud#1" — "t.rd" — "t.rr" — "dagh" — "d.rr" — "dev" — "draa#1" — "draa#2"
— "nud" — "pad#1" — "pii" — "p.rr" — "pyaa" — "bhid#1" — "majj" — "man" —
"mid" — "mlaa" — "ri" — "lii" — "luu#1" — "vid#2" — "vlii" — "zad" — "z.rr"
— "sad#1" — "skand" — "st.rr" — "styaa" — "syand" — "svid#2" — "had" *)
 (* also "suu#2" suuna and "vrii" vrii.na and "k.saa" k.saa.na *)
      → True
  | _ → False
  ]
;
(*****************)
(* Present system *)
(*****************)
```

In all such functions, (*stem* : *word*) is the code of the reversed stem.
Exemple pour cyu: stem=strong=guna=cyo et cyo+ati=cyavati par *int_sandhi*

```
value compute_thematic_presenta cl conj stem entry third =
  let conjug person suff = (person, fix stem suff) in do
  { enter1 entry (Conju (fpresa cl conj)
   [ (Singular,
        [ conjug First "aami"
        ; conjug Second "asi"
```

```
        ; check entry cl third (conjug Third "ati")
        ])
   ; (Dual,
        [ conjug First "aavas"
        ; conjug Second "athas"
        ; conjug Third "atas"
        ])
   ; (Plural,
        [ conjug First "aamas"
        ; conjug Second "atha"
        ; conjug Third "anti"
        ])
   ])
  ; let m_stem = match entry with (* Whitney§450 *)
        [ "b.rh#1" → revcode "b.rh" (* not b.r.mh *)
        | _ → stem
        ] in
    let f_stem = match entry with (* Whitney§450f *)
        [ "j.rr" | "p.r.s" | "b.rh#1" (* — "mah" *) | "v.rh" → rfix m_stem "at"
        | _ → rfix m_stem "ant"
        ] in
    if cl = 4 ∧ entry ="daa#2" ∨ entry ="mah" then () (* to avoid dyat mahat *)
    else record_part (Ppra_ cl conj m_stem f_stem entry)
  }
;
value compute_thematic_presentm cl conj stem entry third =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju (fpresm cl conj)
   [ (Singular,
        [ conjug First "e"
        ; conjug Second "ase"
        ; check entry cl third (conjug Third "ate")
        ])
   ; (Dual,
        [ conjug First "aavahe"
        ; conjug Second "ethe"
        ; conjug Third "ete"
        ])
   ; (Plural,
        [ conjug First "aamahe"
```

```
        ; conjug Second "adhve"
        ; conjug Third "ante"
        ])
    ])
;
value thematic_preterit_a conjug =
    [ (Singular,
        [ conjug First "am"
        ; conjug Second "as"
        ; conjug Third "at"
        ])
    ; (Dual,
        [ conjug First "aava"
        ; conjug Second "atam"
        ; conjug Third "ataam"
        ])
    ; (Plural,
        [ conjug First "aama"
        ; conjug Second "ata"
        ; conjug Third "an"
        ])
    ]
;
value compute_thematic_impfta cl conj stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (fimpfta cl conj) (thematic_preterit_a conjug))
;
value thematic_preterit_m conjug =
    [ (Singular,
        [ conjug First "e"
        ; conjug Second "athaas"
        ; conjug Third "ata"
        ])
    ; (Dual,
        [ conjug First "aavahi"
        ; conjug Second "ethaam"
        ; conjug Third "etaam"
        ])
    ; (Plural,
        [ conjug First "aamahi"
```

```
        ; conjug Second "adhvam"
        ; conjug Third "anta"
        ])
    ]
;
value compute_thematic_impftm cl conj stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (fimpftm cl conj) (thematic_preterit_m conjug))
;
value compute_thematic_optativea cl conj stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju (fopta cl conj)
    [ (Singular,
        [ conjug First "eyam"
        ; conjug Second "es"
        ; conjug Third "et"
        ])
    ; (Dual,
        [ conjug First "eva"
        ; conjug Second "etam"
        ; conjug Third "etaam"
        ])
    ; (Plural,
        [ conjug First "ema"
        ; conjug Second "eta"
        ; conjug Third "eyur"
        ])
    ])
;
value compute_thematic_optativem cl conj stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju (foptm cl conj)
    [ (Singular,
        [ conjug First "eya"
        ; conjug Second "ethaas"
        ; conjug Third "eta"
        ])
    ; (Dual,
        [ conjug First "evahi"
        ; conjug Second "eyaathaam"
```

```
        ; conjug Third "eyaataam"
        ])
    ; (Plural,
        [ conjug First "emahi"
        ; conjug Second "edhvam"
        ; conjug Third "eran"
        ])
    ])
;
value compute_thematic_imperativea cl conj stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju (fimpera cl conj)
    [ (Singular,
        [ conjug First "aani"
        ; conjug Second "a"
        ; conjug Third "atu"
        ])
    ; (Dual,
        [ conjug First "aava"
        ; conjug Second "atam"
        ; conjug Third "ataam"
        ])
    ; (Plural,
        [ conjug First "aama"
        ; conjug Second "ata"
        ; conjug Third "antu"
        ])
    ])
;
value compute_thematic_imperativem cl conj stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju (fimperm cl conj)
    [ (Singular,
        [ conjug First "ai"
        ; conjug Second "asva"
        ; conjug Third "ataam"
        ])
    ; (Dual,
        [ conjug First "aavahai"
        ; conjug Second "ethaam"
```

```
          ; conjug Third "etaam"
          ])
    ; (Plural,
          [ conjug First "aamahai"
          ; conjug Second "adhvam"
          ; conjug Third "antaam"
          ])
    ])
;
value record_part_m (conj, part_kind) stem entry = match part_kind with
  [ Pprm k → record_part (Pprm_ k conj stem entry)
  | Pprp → record_part (Pprp_ conj stem entry)
  | Ppfta → record_part (Ppfta_ conj stem entry)
  | Ppftm → record_part (Ppftm_ conj stem entry)
  | Pfutm → record_part (Pfutm_ conj stem entry)
  | _ → failwith "Unexpected␣participle"
  ]
;
value record_part_m_th verbal stem entry =
  match entry with
  [ "cint" → let pprm = Pprm_ 10 Primary (revcode "cintayaan") entry in
                record_part pprm (* irregular *)
  | "muc#1" | "sp.rz#1" →
          let mid_stem = rfix stem "aana" in (* Whitney§752 *)
          record_part_m verbal mid_stem entry
  | _ → let mid_stem = trunc_a (rfix stem "amaana") (* -maana *) in
          (* trunc_a needed because possible retroflexion in amaa.na *)
          record_part_m verbal mid_stem entry
  ]
and record_part_m_ath verbal stem entry =
  let suff = if entry = "aas#2" then "iina" (* McDonell§158a *)
             else "aana" (* -aana *) in
  let mid_stem = match rfix stem suff with
                 [ [ 1 :: r ] → r | _ → failwith "Anomaly␣Verbs" ] in
  (* rare (Whitney). Creates bizarre forms such as plu -¿ puplvaana *)
  record_part_m verbal mid_stem entry
;
(* Thematic present system - gana is root's present class *)
value compute_thematic_active gana conj stem entry third = do
  { compute_thematic_presenta gana conj stem entry third
```

```
  ; compute_thematic_impfta gana conj stem entry
  ; compute_thematic_optativea gana conj stem entry
  ; compute_thematic_imperativea gana conj stem entry
  }
and compute_thematic_middle gana conj stem entry third = do
  { compute_thematic_presentm gana conj stem entry third
  ; compute_thematic_impftm gana conj stem entry
  ; compute_thematic_optativem gana conj stem entry
  ; compute_thematic_imperativem gana conj stem entry
  ; record_part_m_th (vpprm gana conj) stem entry
  }
;
value compute_causativea = compute_thematic_active cau_gana Causative
and compute_causativem = compute_thematic_middle cau_gana Causative
and compute_desiderativea = compute_thematic_active des_gana Desiderative
and compute_desiderativem = compute_thematic_middle des_gana Desiderative
;
```

** Gana 2 (root conjugation) **
fix2 : *Word.word* → *string* → *string* → *Word.word*
set indicates connecting vowel string of se.t root

```
value fix2 stem suff set =
  let codesf = code suff in
  let wsfx = match codesf with
      [ [] → error_suffix 1
      | [ c :: _ ] → if vowel c ∨ c = 42 (* y *) then codesf
                     else if set then [ 3 :: codesf ] (* pad with initial i *)
                     else codesf
      ] in
  sandhi stem wsfx
;
(* correction for i, ii, u, uu roots of gana 2 *)
value correct2 weak = match weak with
    [ [ 3 ] (* i *) → weak (* eg ppr yat P{6,4,81} *)
    | [ 3 (* i *) :: rest ] → [ 42 :: weak ]
    | [ 4; 46 ] (* zii *) → [ 42; 1; 46 ] (* zay *)
    | [ 4 (* ii *) :: rest ] → [ 42 :: [ 3 :: rest ] ] (* iy *)
    | [ 5 (* u *) :: rest ] → [ 45 :: weak ]
    | [ 6 (* uu *) :: rest ] → [ 45 :: [ 5 :: rest ] ] (* uv *)
    | _ → weak
```

```
    ]
;
value fix2w weak suff set =
  let weakv = correct2 weak
  and weakc = match weak with
    [ [ 4; 46 ] (* zii *) → [ 10; 46 ] (* ze *)
    | _ → weak
    ] in
  match code suff with
    [ [ c :: _ ] → fix2 (if vowel c then weakv else weakc) suff set
    | [] → error_suffix 7
    ]
;
value fix2w_augment weak suff set = aug (fix2w weak suff set)
;
value fix2wi suff = (* special for root i Atma ii -¿ iiy ai -¿ aiy *)
  match code suff with (* P{6,4,77} MacDonell§134.3d *)
    [ [ c :: _ ] → fix2 (if vowel c then [ 42; 3 ] else [ 3 ]) suff False
    | [] → error_suffix 15
    ]
;
value fix2whan suff =
  let codesf = code suff in
  let stem = match codesf with
     [ [] → error_suffix 2
     | [ c :: _ ] → if vowel c then "ghn"
                     else if c = 41 ∨ c = 42 ∨ c = 45 (* m y v *) then "han"
                     else "ha"
     ] in
  sandhi (revcode stem) codesf
;
value fix2whan_augment suff =
  let codesf = code suff in
  let stem = match codesf with
     [ [] → error_suffix 3
     | [ c :: _ ] → if vowel c then "aghn"
                     else if c = 41 ∨ c = 42 ∨ c = 45 (* m y v *) then "ahan"
                     else "aha"
     ] in
  sandhi (revcode stem) codesf
```

```
;
(* correction for u roots *)
value fix2s strong suff set = match strong with
  [ [ 12 (* o *) :: rest ] → match code suff with
      [ [ c :: _ ] → if vowel c then fix2 strong suff set
                       else fix2 [ 13 (* au *) :: rest ] suff set
      | [] → error_suffix 4
      ]
  | _ → fix2 strong suff set
  ]
;
value fix2s_augment strong suff set = aug (fix2s strong suff set)
;
value fix2sbruu suff =
  let strong = revcode "bro" in
  match code suff with
      [ [ c :: _ ] → let suff' = if vowel c then suff else "ii" ^ suff in
                       fix2 strong suff' False
      | [] → error_suffix 5
      ]
;
value fix2sbruu_augment suff = aug (fix2sbruu suff)
;
(* P{6,1,6} reduplicated roots dropping the n of 3rd pl -anti *)
value abhyasta = fun
  [ "jak.s" | "jaag.r" | "cakaas" → True (* zaas has special treatment *)
  | _ → False
  ]
;
value compute_athematic_present2a strong weak set entry third =
  let conjugs person suff =
      (person,if entry = "bruu" then fix2sbruu suff
              else fix2s strong suff set)
  and conjugw person suff =
      (person,if entry = "han#1" then fix2whan suff
              else fix2w weak suff set) in do
  { enter1 entry (Conju (presa 2)
   [ (Singular, let l =
        [ conjugs First "mi"
        ; if entry = "as#1" then (Second, code "asi")
```

```
          else conjugs Second "si"
        ; check entry 2 third (conjugs Third "ti")
        ] in if entry ="bruu" then [ conjugw First "mi" :: l ]
             else if entry ="stu" then [ (First, code "staviimi") :: l ]
             else l (* bruumi Whitney§632 staviimi Whitney§633 *))
  ; (Dual,
        [ conjugw First "vas"
        ; conjugw Second "thas"
        ; conjugw Third "tas"
        ])
  ; (Plural, let l =
        [ conjugw First "mas"
        ; conjugw Second "tha"
        ; if entry = "zaas" then conjugs Third "ati" (* P{7,1,4} *)
          else conjugw Third (if abhyasta entry then "ati" else "anti")
        ] in if entry = "m.rj" then [ conjugs Third "anti" :: l ]
             else l (* Whitney§627 *))
  ])
 }
;
value compute_athematic_present2m strong weak set entry third =
  let conjugs person suff =
      (person,if entry = "bruu" then fix2sbruu suff
                else fix2s strong suff set)
  and conjugw person suff =
      (person,if entry = "han#1" then fix2whan suff
                else if entry = "i" then fix2wi suff (* Gonda§64.III *)
                else fix2w weak suff set) in
  enter1 entry (Conju (presm 2)
   [ (Singular, let l =
        [ if entry = "as#1" then (First, code "he") else
          conjugw First "e"
        ; conjugw Second "se"
        ; check entry 2 third (conjugw Third "te")
        ] in if entry = "m.rj" then [ conjugs First "e" :: l ]
             else l (* Whitney§627 *))
  ; (Dual, let l =
        [ conjugw First "vahe"
        ; conjugw Second "aathe"
        ; conjugw Third "aate"
```

```
        ] in if entry = "m.rj" then
                [ conjugs Second "aathe"
                ; conjugs Third "aate"
                ] @ l
              else l (* Whitney§627 *))
  ; (Plural, let l =
        [ conjugw First "mahe"
        ; if entry = "as#1" then (Second, code "dhve") else
          if entry = "aas#2" then (Second, code "aadhve") else (* -Whitney§612 *)
          conjugw Second "dhve"
        ; if entry = "zii#1" then conjugw Third "rate" (* P{7,1,6} *)
          else conjugw Third "ate"
        ] in if entry = "m.rj" then [ conjugs Third "ate" :: l ]
              else l (* Whitney§627 *))
  ])
;
value compute_athematic_impft2a strong weak set entry =
  let conjugs person suff =
      (person,if entry = "bruu" then fix2sbruu_augment suff
                else fix2s_augment strong suff set)
  and conjugw person suff =
      (person,if entry = "han#1" then fix2whan_augment suff
                else fix2w_augment weak suff set) in
  enter1 entry (Conju (impfta 2)
   [ (Singular, let l =
        [ conjugs First "am"
        ; if set then conjugs Second "as"
          else if entry = "as#1" then conjugs Second "iis" (* Whitney§621c *)
          else if entry = "ad#1" then conjugs Second "as" (* Whitney§621c *)
          else conjugs Second "s"
        ; if set then conjugs Third "at"
          else if entry = "as#1" then conjugs Third "iit" (* idem aasiit *)
          else if entry = "ad#1" then conjugs Third "at" (* idem aadat *)
          else conjugs Third "t"
        ] in if set then [ conjugs Second "iis"; conjugs Third "iit" ] @ l
              else if entry = "bruu"
                    then [ (First, code "abruvam") (* Whitney§632 *) :: l ]
              else l)
  ; (Dual,
        [ conjugw First "va"
```

```
          ; conjugw Second "tam"
          ; conjugw Third "taam"
          ])
    ; (Plural, let l =
          [ conjugw First "ma"
          ; conjugw Second "ta"
          ; if entry = "i" then conjugs Third "an" (* aayan *)
            else match entry with (* Kane§429 *)
                 [ "cakaas" | "jak.s" | "jaag.r"
              (* — "daridraa" - should concern "draa#1" TODO *)
                 | "zaas" → conjugw Third "us"
                 | _ → conjugw Third "an"
                 ]
          ] in if entry = "m.rj"
                    then [ conjugs Third "an" :: l ] (* Whitney§627 *)
               else if entry = "bruu"
                    then [ (Third, code "abruuvan") :: l ] (* Whitney§632 *)
               else match weak with (* Kale§420 optional -us for roots in -aa *)
                    [ [ 2 :: s ] → [ (Third, aug (sandhi s (code "us"))) :: l ]
                    | _ → l
                    ])
    ])
;
value compute_athematic_impft2m strong weak set entry =
  let conjugs person suff =
      (person,if entry = "bruu" then fix2sbruu_augment suff
              else fix2s_augment strong suff set)
  and conjugw person suff =
      (person,if entry = "han#1" then fix2whan_augment suff
              else fix2w_augment weak suff set) in
  enter1 entry (Conju (impftm 2)
   [ (Singular, let l =
          [ if entry = "i" then conjugw First "yi" (* adhyaiyi Bucknell 128 *)
            else conjugw First "i"
          ; conjugw Second "thaas"
          ; conjugw Third "ta"
          ] in if entry = "m.rj" then [ conjugs First "i" :: l ]
               else l (* Whitney§627 *))
   ; (Dual, let l =
          [ conjugw First "vahi"
```

```
        ; conjugw Second "aathaam"
        ; conjugw Third "aataam"
        ] in if entry = "m.rj" then
                [ conjugs Second "aathaam"
                ; conjugs Third "aataam"
                ] @ l else l (* Whitney§627 *))
    ; (Plural, let l =
        [ conjugw First "mahi"
        ; if entry = "aas#2" then (Second, code "aadhvam") (* -Whitney§620 *)
          else conjugw Second "dhvam"
        ; if entry = "zii#1" then conjugw Third "rata" (* P{7,1,6} *) else
          if entry = "i" then conjugw Third "yata" (* Bucknell 128 *) else
          conjugw Third "ata"
        ] in if entry = "m.rj" then [ conjugs Third "ata" :: l ] else
             if entry ="duh#1" then [ conjugw Third "ra" :: l ]
             (* aduhata -¿ aduha-a = P{7,1,41} aduha -¿ aduhra P{7,1,8} *)
             else l (* Whitney§627 *))
   ])
;
value compute_athematic_optative2a weak set entry =
  let conjugw person suff =
      (person,if entry = "han#1" then fix2whan suff
                                  else fix2w weak suff set) in
  enter1 entry (Conju (opta 2)
   [ (Singular, let l =
        [ conjugw First "yaam"
        ; conjugw Second "yaas"
        ; conjugw Third "yaat"
        ] in if entry = "bruu"
             then [ (Third, code "bruyaat") (* Whitney§632 *) :: l ]
             else l)
   ; (Dual,
        [ conjugw First "yaava"
        ; conjugw Second "yaatam"
        ; conjugw Third "yaataam"
        ])
   ; (Plural,
        [ conjugw First "yaama"
        ; conjugw Second "yaata"
        ; conjugw Third "yur"
```

```
        ])
    ])
;
value compute_athematic_optative2m weak set entry =
  let conjugw person suff =
      (person,if entry = "han#1" then fix2whan suff
              else if entry = "i" then fix2wi suff (* adhiiyiita *)
              else fix2w weak suff set)
  and conjugwmrij person suff = (person, fix2 (revcode "maarj") suff set) in
  enter1 entry (Conju (optm 2)
   [ (Singular, let l = (* ii below replaced by iyii for root i ? *)
        [ conjugw First "iiya"
        ; conjugw Second "iithaas"
        ; conjugw Third "iita"
        ] in if entry = "m.rj" then
                [ conjugwmrij First "iiya"
                ; conjugwmrij Second "iithaas"
                ; conjugwmrij Third "iita"
                ] @ l
             else l (* Whitney§627 *))
   ; (Dual, let l =
        [ conjugw First "iivahi"
        ; conjugw Second "iiyaathaam"
        ; conjugw Third "iiyaataam"
        ] in if entry = "m.rj" then
                [ conjugwmrij First "iivahi"
                ; conjugwmrij Second "iiyaathaam"
                ; conjugwmrij Third "iiyaataam"
                ] @ l
             else l (* Whitney§627 *))
   ; (Plural, let l =
        [ conjugw First "iimahi"
        ; conjugw Second "iidhvam"
        ; conjugw Third "iiran" (* special dropping n TODO see Kane§429 *)
        ] in if entry = "m.rj" then
                [ conjugwmrij First "iimahi"
                ; conjugwmrij Second "iidhvam"
                ; conjugwmrij Third "iiran"
                ] @ l
             else l (* Whitney§627 *))
```

```
    ])
;
value compute_athematic_imperative2a strong weak set entry =
  let conjugs person suff =
      (person,if entry = "bruu" then fix2sbruu suff
                                 else fix2s strong suff set)
  and conjugw person suff =
      (person,if entry = "han#1" then fix2whan suff
                                  else fix2w weak suff set) in
  enter1 entry (Conju (impera 2)
   [ (Singular, let l =
        [ conjugs First "aani"
        ; (Second, match entry with
           [ "as#1" → code "edhi"
           | "zaas" → code "zaadhi"
 (* above leads to conflict between P{6,4,35} (zaa+hi) and P{6,4,101} (zaas+dhi) asiddhavat
=¿ we operate in parallel zaa+dhi= zaadhi *)
           | "cakaas" → code "cakaadhi" (* Kane§429 *)
           | _ → let w = if entry = "han#1" then revcode "ja" else weak in
                 match w with
             [ [ c :: _ ] → fix2 w suff set
               where suff = if vowel c ∨ set then "hi" else "dhi"
             | _ → error_empty 6
             ] (* "dhi" or "hi" after vowel *)
           ])
        ; conjugs Third "tu"
        ] in if entry = "vac" then
                 [ (Second, code "voci"); (Third, code "vocatu") ] @ l
             else if entry ="bruu" then [ conjugs Second "hi" :: l ]
                    (* braviihi Whitney§632 *)
             else if entry ="cakaas" then [ (Second, code "cakaadvi") :: l ]
                    (* Kane§429 *)
             else l)
   ; (Dual,
        [ conjugs First "aava"
        ; conjugw Second "tam"
        ; conjugw Third "taam"
        ])
   ; (Plural, let l =
        [ conjugs First "aama"
```

```
        ; conjugw Second "ta"
        ; if entry = "zaas" then conjugs Third "atu" (* P{7,1,4} *)
        else conjugw Third (if abhyasta entry then "atu" else "antu")
        ] in if entry = "m.rj" then [ conjugs Third "antu" :: l ]
             else l (* Whitney§627 *))
    ])
;
value compute_athematic_imperative2m strong weak set entry =
  let conjugs person suff =
      (person,if entry = "bruu" then fix2sbruu suff
                  else fix2s strong suff set)
  and conjugw person suff =
      (person,if entry = "han#1" then fix2whan suff
                  else fix2w weak suff set) in
  enter1 entry (Conju (imperm 2)
   [ (Singular,
        [ conjugs First "ai"
        ; conjugw Second "sva"
        ; conjugw Third "taam"
        ])
   ; (Dual, let l =
        [ conjugs First "aavahai"
        ; conjugw Second "aathaam"
        ; conjugw Third "aataam"
        ] in if entry = "m.rj" then
                  [ conjugs Second "aathaam"
                  ; conjugs Third "aataam"
                  ] @ l
             else l (* Whitney§627 *))
   ; (Plural, let l =
        [ conjugs First "aamahai"
        ; if entry = "aas#2" then (Second, code "aadhvam") (* -Whitney§617 *)
          else conjugw Second "dhvam"
        ; if entry = "zii#1" then conjugw Third "rataam" (* P{7,1,6} *)
          else conjugw Third "ataam"
        ] in if entry = "m.rj" then [ conjugs Third "ataam" :: l ]
             else l (* Whitney§627 *))
   ])
;
value compute_active_present2 sstem wstem set entry third = do
```

```
  { compute_athematic_present2a sstem wstem set entry third
  ; let weak = if entry = "as#1" then [ 48; 1 ] else wstem in
     compute_athematic_impft2a sstem weak set entry
  ; compute_athematic_optative2a wstem set entry
  ; compute_athematic_imperative2a sstem wstem set entry
  ; match wstem with
     [ [ 2 :: _ ] → (* Ppr of roots in -aa is complex and overgenerates *)
        match entry with
        [ "bhaa#1" | "maa#1" | "yaa#1" → () (* no known ppra *)
        | _ → let m_pstem = wstem and f_pstem = rev (fix2w wstem "at" set) in
              record_part (Ppra_ 2 Primary m_pstem f_pstem entry)
        ]
     | _ → let m_pstem = if entry = "han#1" then revstem "ghn"
                         else correct2 wstem in
           let f_pstem = if entry = "han#1" then revstem "ghnat"
                         else rev (fix2w wstem "at" set) in
           record_part (Ppra_ 2 Primary m_pstem f_pstem entry)
     ]
  ; if entry = "m.rj" then let m_pstem = revstem "maarj" in
                           let f_pstem = revstem "maarjat" in
                           record_part (Ppra_ 2 Primary m_pstem f_pstem entry)
     else ()
  }
and compute_middle_present2 sstem wstem set entry third = do
  { compute_athematic_present2m sstem wstem set entry third
  ; compute_athematic_impft2m sstem wstem set entry
  ; compute_athematic_optative2m wstem set entry
  ; compute_athematic_imperative2m sstem wstem set entry
  ; match entry with
     [ "maa#1" → () (* no pprm *)
     | "i" → record_part_m_ath (pprm 2) [ 42; 3 ] entry (* iyaana *)
     | _ → record_part_m_ath (pprm 2) (correct2 wstem) entry
     ]
  }
;
```

** Gana 3 **

```
value strip_ii = fun
  [ [ 4 :: w ] → w (* ii disappears before vowels in special roots *)
  | _ → failwith "Wrong␣weak␣stem␣of␣special␣3rd␣class␣root"
```

```
  ]
;
value fix3w wstem iiflag dadh suff =
  let codesf = code suff in
  let short = if iiflag then strip_ii wstem else wstem in
  let stem = match codesf with
      [ [] → error_suffix 8
      | [ 5; 43 ] (* ur *) → if iiflag then short else strong wstem (* guna *)
      | [ c :: _ ] → if dadh then match c with (* Gonda§66 *)
              [ 32 | 33 | 35 | 48 | 49 (* t th dh s h *) → revstem "dhad"
                  (* aspirate correction of sandhi not enough : dh+t=ddh not tt *)
              | _ → short
              ] else if vowel c then short else wstem
      ] in
  sandhi stem codesf
;
value fix3w_augment wstem iiflag dadh suff = aug (fix3w wstem iiflag dadh suff)
;
value compute_athematic_present3a strong weak iiflag entry third =
  let dadh_flag = (entry ="dhaa#1") in
  let conjugs person suff = (person, fix strong suff) (* gu.na *)
  and conjugw person suff = (person, fix3w weak iiflag dadh_flag suff)
  and conjughaa person suff = (person, fix (revstem "jahi") suff)
                                      (* weak = jahii but optionally jahi *)
  and haa_flag = (entry ="haa#1") in do
  { enter1 entry (Conju (presa 3)
   [ (Singular,
          [ conjugs First "mi" (* Panini mip, where p indicates guna *)
          ; conjugs Second "si" (* sip *)
          ; check entry 3 third (conjugs Third "ti") (* tip *)
          ])
    ; (Dual, let l =
          [ conjugw First "vas"
          ; conjugw Second "thas"
          ; conjugw Third "tas"
          ] in if haa_flag then l @
                  [ conjughaa First "vas"
                  ; conjughaa Second "thas"
                  ; conjughaa Third "tas"
                  ]
```

```
              else l)
    ; (Plural, let l =
          [ conjugw First "mas"
          ; conjugw Second "tha"
          ; if entry ="bhas" then (Third, code "bapsati") (* Whitney§678 MW§340 *)
            else conjugw Third "ati"
          ] in if haa_flag then l @
                    [ conjughaa First "mas"
                    ; conjughaa Second "tha"
                    ]
              else l)
    ])
  ; let wstem = if iiflag then strip_ii weak else
                    if entry ="bhas" then revcode "baps" (* Whitney§678 *)
                    else weak in (* 3rd pl weak stem *)
    record_part (Pprared_ Primary wstem entry)
  }
;
value compute_athematic_present3m conj gana weak iiflag entry third =
  let dadh_flag = (entry ="dhaa#1") in
  let conjugw person suff = (person, fix3w weak iiflag dadh_flag suff) in
  enter1 entry (Conju (fpresm gana conj)
    [ (Singular,
          [ conjugw First "e"
          ; conjugw Second "se"
          ; check entry 3 third (conjugw Third "te")
          ])
    ; (Dual,
          [ conjugw First "vahe"
          ; conjugw Second "aathe"
          ; conjugw Third "aate"
          ])
    ; (Plural,
          [ conjugw First "mahe"
          ; conjugw Second "dhve"
          ; conjugw Third "ate"
          ])
    ])
;
value compute_athematic_impft3a strong weak iiflag entry =
```

```
  let dadh_flag = (entry ="dhaa#1") in
  let conjugs person suff = (person, fix_augment strong suff)
  and conjugw person suff = (person, fix3w_augment weak iiflag dadh_flag suff)
  and conjughaa person suff = (person, fix_augment (revstem "jahi") suff)
  and haa_flag = (entry ="haa#1") in
  enter1 entry (Conju (impfta 3)
   [ (Singular, let l =
        [ conjugs First "am"
        ; conjugs Second "s"
        ; conjugs Third "t"
        ] in if haa_flag then l @
                [ conjughaa Second "s"
                ; conjughaa Third "t"
                ]
             else l)
   ; (Dual, let l =
        [ conjugw First "va"
        ; conjugw Second "tam"
        ; conjugw Third "taam"
        ] in if haa_flag then l @
                [ conjughaa First "va"
                ; conjughaa Second "tam"
                ; conjughaa Third "taam"
                ]
             else l)
   ; (Plural, let l =
        [ conjugw First "ma"
        ; conjugw Second "ta"
        ; conjugw Third "ur"
        ] in if haa_flag then l @
                [ conjughaa First "ma"
                ; conjughaa Second "ta"
                ]
             else l)
   ])
;
(* common to impft_m and root_aoristm *)
value conjugs_past_m conjug =
   [ (Singular,
        [ conjug First "i"
```

```
        ; conjug Second "thaas"
        ; conjug Third "ta"
        ])
  ; (Dual,
        [ conjug First "vahi"
        ; conjug Second "aathaam"
        ; conjug Third "aataam"
        ])
  ; (Plural,
        [ conjug First "mahi"
        ; conjug Second "dhvam"
        ; conjug Third "ata"
        ])
  ]
;
value conjug_impft_m gana conjugw = (* used by classes 3 and 9 *)
  Conju (impftm gana) (conjugs_past_m conjugw)
;
value compute_athematic_impft3m weak iiflag entry =
  let dadh_flag = (entry ="dhaa#1") in
  let conjugw person suff = (person, fix3w_augment weak iiflag dadh_flag suff) in
  enter1 entry (conjug_impft_m 3 conjugw)
;
(* Like compute_athematic_optative2a except for yan#1 and bruu *)
value conjug_optativea gana conj conjugw =
  Conju (fopta gana conj)
  [ (Singular,
        [ conjugw First "yaam"
        ; conjugw Second "yaas"
        ; conjugw Third "yaat"
        ])
  ; (Dual,
        [ conjugw First "yaava"
        ; conjugw Second "yaatam"
        ; conjugw Third "yaataam"
        ])
  ; (Plural,
        [ conjugw First "yaama"
        ; conjugw Second "yaata"
        ; conjugw Third "yur"
```

```
          ])
    ]
;
value conjug_opt_ath_a gana = conjug_optativea gana Primary
;
value compute_athematic_optative3a weak iiflag entry =
  let dadh_flag = (entry ="dhaa#1") in
  let conjugw person suff = (person,
       if entry ="haa#1" then fix (revstem "jah") suff
       else fix3w weak iiflag dadh_flag suff) in
  enter1 entry (conjug_opt_ath_a 3 conjugw)
;
value conjug_opt_ath_m gana conjugw =
  Conju (optm gana)
    [ (Singular,
          [ conjugw First "iiya"
          ; conjugw Second "iithaas"
          ; conjugw Third "iita"
          ])
    ; (Dual,
          [ conjugw First "iivahi"
          ; conjugw Second "iiyaathaam"
          ; conjugw Third "iiyaataam"
          ])
    ; (Plural,
          [ conjugw First "iimahi"
          ; conjugw Second "iidhvam"
          ; conjugw Third "iiran"
          ])
    ]
;
value compute_athematic_optative3m weak iiflag entry =
  let dadh_flag = (entry ="dhaa#1") in
  let conjugw person suff = (person, fix3w weak iiflag dadh_flag suff) in
  enter1 entry (conjug_opt_ath_m 3 conjugw)
;
value compute_athematic_imperative3a strong weak iiflag entry =
  let dadh_flag = (entry ="dhaa#1")
  and daa_flag = (entry ="daa#1")
  and haa_flag = (entry ="haa#1") in
```

```
let conjugs person suff = (person, fix strong suff)
and conjugw person suff = (person, fix3w weak iiflag dadh_flag suff)
and conjughaa person suff = (person, fix (revstem "jahi") suff) in
enter1 entry (Conju (impera 3)
 [ (Singular, let l =
      [ conjugs First "aani"
      ; (Second, if daa_flag then code "dehi" (* P{4,4,119} *)
                 else if dadh_flag then code "dhehi" (* idem ghu P{1,1,20} *)
                 else match weak with
          [ [ c :: _ ] → fix3w weak iiflag dadh_flag suff
             where suff = if vowel c then (* "dhi" or "hi" after vowel *)
                              if entry = "hu" then "dhi" else "hi"
                              (* "hu" only exception Pan6,4,101 Müller p153 *)
                          else "dhi"
          | _ → error_empty 7
          ] )
      ; conjugs Third "tu"
      ] in if haa_flag then l @
              [ conjughaa Second "hi" (* jahihi *)
              ; conjugs Second "hi" (* jahaahi *)
              ; conjughaa Third "tu" (* jahitu *)
              ]
           else l)
 ; (Dual, let l =
      [ conjugs First "aava"
      ; conjugw Second "tam"
      ; if entry ="bhas" then (Third, code "babdhaam") (* Whitney§678 MW§340 *)
        else conjugw Third "taam"
      ] in if haa_flag then l @
              [ conjughaa Second "tam"
              ; conjughaa Third "taam"
              ]
           else l)
 ; (Plural, let l =
      [ conjugs First "aama"
      ; conjugw Second "ta"
      ; conjugw Third "atu"
      ] in if haa_flag then l @ [ conjughaa Second "ta" ]
           else l)
 ])
```

```
;
value compute_imp_ath_m gana conjugs conjugw entry =
  enter1 entry (Conju (imperm gana)
   [ (Singular,
         [ conjugs First "ai"
         ; conjugw Second "sva"
         ; conjugw Third "taam"
         ])
   ; (Dual,
         [ conjugs First "aavahai"
         ; conjugw Second "aathaam"
         ; conjugw Third "aataam"
         ])
   ; (Plural,
         [ conjugs First "aamahai"
         ; conjugw Second "dhvam"
         ; conjugw Third "ataam"
         ])
   ])
;
value compute_athematic_imperative3m strong weak iiflag entry =
  let dadh_flag = (entry ="dhaa#1") in
  let conjugs person suff = (person,fix strong suff)
  and conjugw person suff = (person,fix3w weak iiflag dadh_flag suff) in
  compute_imp_ath_m 3 conjugs conjugw entry
;
value compute_active_present3 sstem wstem iiflag entry third = do
  { compute_athematic_present3a sstem wstem iiflag entry third
  ; compute_athematic_impft3a sstem wstem iiflag entry
  ; compute_athematic_optative3a wstem iiflag entry
  ; compute_athematic_imperative3a sstem wstem iiflag entry
  }
and compute_middle_present3 sstem wstem iiflag entry third = do
  { compute_athematic_present3m Primary 3 wstem iiflag entry third
  ; compute_athematic_impft3m wstem iiflag entry
  ; compute_athematic_optative3m wstem iiflag entry
  ; compute_athematic_imperative3m sstem wstem iiflag entry
  ; let short = if iiflag then strip_ii wstem else wstem in
    record_part_m_ath (pprm 3) short entry
  }
```

```
;
** Gana 5 **
value compute_athematic_present5a gana strong weak vow entry third =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = match code suff with
      [ [ c :: _ ] →
        if vowel c then
            let w = if vow then weak else [ 45 (* v *) :: weak ] in
            (person, fix w suff)
        else (person, fix weak suff)
      | [] → error_suffix 9
      ]
  and conjugw2 person suff = match weak with
      [ [ 5 :: no_u ] → (person, fix no_u suff)
      | _ → failwith "5a weak ought to end in u"
      ] in do
  { enter1 entry (Conju (presa gana)
   [ (Singular,
        [ conjugs First "mi"
        ; conjugs Second "si"
        ; check entry gana third (conjugs Third "ti")
        ])
   ; (Dual, let l =
        [ conjugw First "vas"
        ; conjugw Second "thas"
        ; conjugw Third "tas"
        ] in
        if vow then [ conjugw2 First "vas" (* optional elision of u *) :: l ]
               else l)
   ; (Plural, let l =
        [ conjugw First "mas"
        ; conjugw Second "tha"
        ; conjugw Third "anti"
        ] in
        if vow then [ conjugw2 First "mas" (* optional elision of u *) :: l ]
               else l)
   ])
  ; let m_pstem = if vow then weak else [ 45 (* v *) :: weak ] in
    let f_pstem = rfix m_pstem "at" in
```

```
    record_part (Ppra_ 5 Primary m_pstem f_pstem entry)
  }
;
value compute_athematic_present5m gana weak vow entry third =
  let conjugw person suff = match code suff with
      [ [ c :: _ ] → if vowel c then
                          let w = if vow then weak else [ 45 (* v *) :: weak ] in
                          (person, fix w suff)
                       else (person, fix weak suff)
      | [] → error_suffix 17
      ]
  and conjugw2 person suff = match weak with
      [ [ 5 :: no_u ] → (person, fix no_u suff)
      | _ → failwith "5m weak ought to end in u"
      ] in
  enter1 entry (Conju (presm gana)
   [ (Singular,
        [ conjugw First "e"
        ; conjugw Second "se"
        ; check entry gana third (conjugw Third "te")
        ])
   ; (Dual, let l =
        [ conjugw First "vahe"
        ; conjugw Second "aathe"
        ; conjugw Third "aate"
        ] in
        if vow then [ conjugw2 First "vahe" (* optional elision of u *) :: l ]
        else l)
   ; (Plural, let l =
        [ conjugw First "mahe"
        ; conjugw Second "dhve"
        ; conjugw Third "ate"
        ] in
        if vow then [ conjugw2 First "mahe" (* optional elision of u *) :: l ]
        else l)
   ])
;
value compute_athematic_impft5a gana strong weak vow entry =
  let conjugs person suff = (person, fix_augment strong suff)
  and conjugw person suff = match code suff with
```

```
        [ [ c :: _ ] →
          if vowel c then
              let w = if vow then weak else [ 45 (* v *) :: weak ] in
              (person, fix_augment w suff)
          else (person, fix_augment weak suff)
        | [] → error_suffix 10
        ]
  and conjugw2 person suff = match weak with
        [ [ 5 :: no_u ] → (person, fix_augment no_u suff)
        | _ → failwith "5a␣weak␣ought␣to␣end␣in␣u"
        ] in
  enter1 entry (Conju (impfta gana)
   [ (Singular,
         [ conjugs First "am"
         ; conjugs Second "s"
         ; conjugs Third "t"
         ])
   ; (Dual, let l =
         [ conjugw First "va"
         ; conjugw Second "tam"
         ; conjugw Third "taam"
         ] in
        if vow then [ conjugw2 First "va" (* optional elision of u *) :: l ]
                else l)
   ; (Plural, let l =
         [ conjugw First "ma"
         ; conjugw Second "ta"
         ; conjugw Third "an"
         ] in
        if vow then [ conjugw2 First "ma" (* optional elision of u *) :: l ]
        else l)
   ])
;
value compute_athematic_impft5m gana weak vow entry =
  let conjugw person suff = match code suff with
        [ [ c :: _ ] →
          if vowel c then
              let w = if vow then weak else [ 45 (* v *) :: weak ] in
              (person, fix_augment w suff)
          else (person, fix_augment weak suff)
```

```
        | [] → error_suffix 14
        ]
  and conjugw2 person suff = match weak with
        [ [ 5 :: no_u ] → (person, fix_augment no_u suff)
        | _ → failwith "5m weak ought to end in u"
        ] in
  enter1 entry (Conju (impftm gana)
   [ (Singular,
          [ conjugw First "i"
          ; conjugw Second "thaas"
          ; conjugw Third "ta"
          ])
   ; (Dual, let l =
          [ conjugw First "vahi"
          ; conjugw Second "aathaam"
          ; conjugw Third "aataam"
          ] in
        if vow then [ conjugw2 First "vahi" (* optional elision of u *) :: l ]
        else l)
   ; (Plural, let l =
          [ conjugw First "mahi"
          ; conjugw Second "dhvam"
          ; conjugw Third "ata"
          ] in
        if vow then [ conjugw2 First "mahi" (* optional elision of u *) :: l ]
        else l)
   ])
;
value compute_athematic_optative5a gana weak vow entry = (* gana=5 or 8 *)
  let conjugw person suff = match code suff with
        [ [ c :: _ ] →
          if vowel c then
              let w = if vow then weak else [ 45 (* v *) :: weak ] in
              (person, fix w suff)
          else (person, fix weak suff)
        | [] → error_suffix 11
        ] in
  enter1 entry (conjug_opt_ath_a gana conjugw)
;
value compute_athematic_optative5m gana weak vow entry = (* gana=5 or 8 *)
```

```
  let conjugw person suff = match code suff with
      [ [ c :: _ ] →
        if vowel c then
            let w = if vow then weak else [ 45 (* v *) :: weak ] in
            (person, fix w suff)
        else (person, fix weak suff)
      | [] → error_suffix 19
      ] in
  enter1 entry (conjug_opt_ath_m gana conjugw)
;
value compute_athematic_imperative5a gana strong weak vow entry = (* gana=5 or 8 *)
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = match code suff with
      [ [ c :: _ ] → if vowel c then
                          let w = if vow then weak else [ 45 (* v *) :: weak ] in
                          (person, fix w suff)
                      else (person, fix weak suff)
      | [] → (person, fix weak "")
      ] in
  enter1 entry (Conju (impera gana)
   [ (Singular,
        [ conjugs First "aani"
        ; conjugw Second (if vow then "" else "hi")
        ; conjugs Third "tu"
        ])
   ; (Dual,
        [ conjugs First "aava"
        ; conjugw Second "tam"
        ; conjugw Third "taam"
        ])
   ; (Plural,
        [ conjugs First "aama"
        ; conjugw Second "ta"
        ; conjugw Third "antu"
        ])
   ])
;
value compute_athematic_imperative5m gana strong weak vow entry = (* gana=5 or 8 *)
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = match code suff with
```

```
      [ [ c :: _ ] →
        if vowel c then
           let w = if vow then weak else [ 45 (* v *) :: weak ] in
              (person, fix w suff)
        else (person, fix weak suff)
      | [] → (person, fix weak "")
      ] in
  compute_imp_ath_m gana conjugs conjugw entry
;
(* Used by classes 5 and 8 *)
value compute_active_present5 gana sstem wstem vow entry third = do
  { compute_athematic_present5a gana sstem wstem vow entry third
  ; compute_athematic_impft5a gana sstem wstem vow entry
  ; compute_athematic_optative5a gana wstem vow entry
  ; compute_athematic_imperative5a gana sstem wstem vow entry
  }
and compute_middle_present5 gana sstem wstem vow entry third = do
  { compute_athematic_present5m gana wstem vow entry third
  ; compute_athematic_impft5m gana wstem vow entry
  ; compute_athematic_optative5m gana wstem vow entry
  ; compute_athematic_imperative5m gana sstem wstem vow entry
  ; record_part_m_ath (pprm 5) wstem entry
  }
;
(* Also used by gana 8 *)
value compute_present5 gana sstem wstem vow entry third pada padam =
  match voices_of_gana gana entry with
       [ Para → if pada then
            compute_active_present5 gana sstem wstem vow entry third
            else emit_warning ("Unexpected␣middle␣form:␣" ^ entry)
       | Atma → if padam then emit_warning ("Unexpected␣active␣form:␣" ^ entry)
            else compute_middle_present5 gana sstem wstem vow entry third
       | Ubha →
          let thirda = if pada then third else []
          and thirdm = if pada then [] else third in do
          { compute_active_present5 gana sstem wstem vow entry thirda
          ; compute_middle_present5 gana sstem wstem vow entry thirdm
          }
       ]
;
```

```
** Gana 7 **
value compute_athematic_present7a strong weak entry third =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = (person, fix weak suff) in do
  { enter1 entry (Conju (presa 7)
   [ (Singular,
        [ conjugs First "mi"
        ; conjugs Second "si"
        ; check entry 7 third (conjugs Third "ti")
        ])
   ; (Dual,
        [ conjugw First "vas"
        ; conjugw Second "thas"
        ; conjugw Third "tas"
        ])
   ; (Plural,
        [ conjugw First "mas"
        ; conjugw Second "tha"
        ; conjugw Third "anti"
        ])
   ])
  ; let m_pstem = weak
    and f_pstem = rfix weak "at" in
    record_part (Ppra_ 7 Primary m_pstem f_pstem entry)
  }
;
value compute_athematic_present7m weak entry third =
  let conjugw person suff = (person, fix weak suff) in
  enter1 entry (Conju (presm 7)
   [ (Singular,
        [ conjugw First "e"
        ; conjugw Second "se"
        ; check entry 7 third (conjugw Third "te")
        ])
   ; (Dual,
        [ conjugw First "vahe"
        ; conjugw Second "aathe"
        ; conjugw Third "aate"
        ])
   ; (Plural,
```

```
          [ conjugw First "mahe"
          ; conjugw Second "dhve"
          ; conjugw Third "ate"
          ])
    ])
;
value compute_athematic_impft7a strong weak entry =
  let conjugs person suff = (person, fix_augment strong suff)
  and conjugw person suff = (person, fix_augment weak suff) in
  enter1 entry (Conju (impfta 7)
    [ (Singular, let l =
          [ conjugs First "am"
          ; conjugs Second "s"
          ; conjugs Third "t"
          ] in match rev (fix_augment strong "s") with
                [ [ c :: r ] → if c = 32 (* t *) then
                                      [ (Second, rev [ 48 (* s *) :: r ]) :: l ]
                                      (* abhinad-s -¿ abhinat or abhinas *)
                                  else l (* horrible patch *)
                | _ → error_empty 8
                ])
    ; (Dual,
          [ conjugw First "va"
          ; conjugw Second "tam"
          ; conjugw Third "taam"
          ])
    ; (Plural,
          [ conjugw First "ma"
          ; conjugw Second "ta"
          ; conjugw Third "an"
          ])
    ])
;
value compute_athematic_impft7m weak entry =
  let conjugw person suff = (person, fix_augment weak suff) in
  enter1 entry (Conju (impftm 7)
    [ (Singular,
          [ conjugw First "i"
          ; conjugw Second "thaas"
          ; conjugw Third "ta"
```

```
          ])
    ; (Dual,
          [ conjugw First "vahi"
          ; conjugw Second "aathaam"
          ; conjugw Third "aataam"
          ])
    ; (Plural,
          [ conjugw First "mahi"
          ; conjugw Second "dhvam"
          ; conjugw Third "ata"
          ])
    ])
;
value compute_athematic_optative7a weak entry =
  let glue = if entry = "hi.ms" then fun w s →
                    List2.unstack w (code s) (* no retroflexion Whitney§183a *)
                else fix in
  let conjugw person suff = (person, glue weak suff) in
  enter1 entry (conjug_opt_ath_a 7 conjugw)
;
value compute_athematic_optative7m weak entry =
  let conjugw person suff = (person, fix weak suff) in
  enter1 entry (conjug_opt_ath_m 7 conjugw)
;
value compute_athematic_imperative7a strong weak entry =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = (person, fix weak suff) in
  enter1 entry (Conju (impera 7)
    [ (Singular,
          [ conjugs First "aani"
          ; (Second, match weak with
              [ [ c :: _ ] → fix weak suff
                where suff = if vowel c then "hi" else "dhi"
              | _ → error_empty 9
              ]) (* "dhi" or "hi" after vowel *)
          ; conjugs Third "tu"
          ])
    ; (Dual,
          [ conjugs First "aava"
          ; conjugw Second "tam"
```

```
          ; conjugw Third "taam"
          ])
    ; (Plural,
          [ conjugs First "aama"
          ; conjugw Second "ta"
          ; conjugw Third "antu"
          ])
    ])
;
value compute_athematic_imperative7m strong weak entry =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = (person, fix weak suff) in
  compute_imp_ath_m 7 conjugs conjugw entry
;
value compute_active_present7 sstem wstem entry third = do
  { compute_athematic_present7a sstem wstem entry third
  ; compute_athematic_impft7a sstem wstem entry
  ; compute_athematic_optative7a wstem entry
  ; compute_athematic_imperative7a sstem wstem entry
  }
and compute_middle_present7 sstem wstem entry third = do
  { compute_athematic_present7m wstem entry third
  ; compute_athematic_impft7m wstem entry
  ; compute_athematic_optative7m wstem entry
  ; compute_athematic_imperative7m sstem wstem entry
  ; record_part_m_ath (pprm 7) wstem entry
  }
;
value compute_present7 sstem wstem entry third pada padam =
  match voices_of_gana 7 entry with
  [ Para → if pada then compute_active_present7 sstem wstem entry third
             else emit_warning ("Unexpected␣middle␣form:␣" ^ entry)
  | Atma → if padam then emit_warning ("Unexpected␣active␣form:␣" ^ entry)
             else compute_middle_present7 sstem wstem entry third
  | Ubha → let thirda = if pada then third else []
             and thirdm = if pada then [] else third in do
             { compute_active_present7 sstem wstem entry thirda
             ; compute_middle_present7 sstem wstem entry thirdm
             }
  ]
```

```
;
** Gana 8 **
Conjugation of k.r
    "karo" "kuru" "kur"

value compute_athematic_presentk strong weak short entry third =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = (person, fix weak suff)
  and conjugwvm person suff = (person, fix short suff) (* -v -m suff *) in do
  { enter1 entry (Conju (presa 8)
   [ (Singular,
        [ conjugs First "mi"
        ; conjugs Second "si"
        ; check entry 8 third (conjugs Third "ti")
        ])
   ; (Dual,
        [ conjugwvm First "vas"
        ; conjugw Second "thas"
        ; conjugw Third "tas"
        ])
   ; (Plural,
        [ conjugwvm First "mas"
        ; conjugw Second "tha"
        ; conjugw Third "anti"
        ])
   ])
  ; let f_pstem = rfix weak "at" in
     record_part (Ppra_ 8 Primary weak f_pstem entry)
  ; record_part_m_ath (pprm 8) weak entry
  ; enter1 entry (Conju (presm 8)
   [ (Singular,
        [ conjugw First "e"
        ; conjugw Second "se"
        ; conjugw Third "te"
        ])
   ; (Dual,
        [ conjugwvm First "vahe"
        ; conjugw Second "aathe"
        ; conjugw Third "aate"
        ])
```

```
    ; (Plural,
          [ conjugwvm First "mahe"
          ; conjugw Second "dhve"
          ; conjugw Third "ate"
          ])
    ])
  }
;
value compute_athematic_impftk strong weak short entry =
  let conjugs person suff = (person, fix_augment strong suff)
  and conjugw person suff = (person, fix_augment weak suff)
  and conjugwvm person suff = (person, fix_augment short suff) (* -v -m suff *) in do
  { enter1 entry (Conju (impfta 8)
    [ (Singular,
          [ conjugs First "am"
          ; conjugs Second "s"
          ; conjugs Third "t"
          ])
    ; (Dual,
          [ conjugwvm First "va"
          ; conjugw Second "tam"
          ; conjugw Third "taam"
          ])
    ; (Plural,
          [ conjugwvm First "ma"
          ; conjugw Second "ta"
          ; conjugw Third "an"
          ])
    ])
  ; enter1 entry (Conju (impftm 8) (* similar to conjugs_past_m except for -v -m suff *)
    [ (Singular,
          [ conjugw First "i"
          ; conjugw Second "thaas"
          ; conjugw Third "ta"
          ])
    ; (Dual,
          [ conjugwvm First "vahi"
          ; conjugw Second "aathaam"
          ; conjugw Third "aataam"
          ])
```

```
    ; (Plural,
          [ conjugwvm First "mahi"
          ; conjugw Second "dhvam"
          ; conjugw Third "ata"
          ])
    ])
  }
;
value compute_athematic_optativek weak short entry =
  let conjugw person suff = (person, fix weak suff)
  and conjugs person suff = (person, fix short suff) in do
  { enter1 entry (conjug_opt_ath_a 8 conjugs) (* short since -y suffixes *)
  ; enter1 entry (conjug_opt_ath_m 8 conjugw)
  }
;
value compute_athematic_imperativek strong weak entry =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = (person, fix weak suff) in do
  { enter1 entry (Conju (impera 8)
    [ (Singular,
          [ conjugs First "aani"
          ; conjugw Second ""
          ; conjugs Third "tu"
          ])
    ; (Dual,
          [ conjugs First "aava"
          ; conjugw Second "tam"
          ; conjugw Third "taam"
          ])
    ; (Plural,
          [ conjugs First "aama" (* also kurma Epics *)
          ; conjugw Second "ta"
          ; conjugw Third "antu"
          ])
    ])
  ; compute_imp_ath_m 8 conjugs conjugw entry
  }
;
value compute_presentk sstem wstem short entry third = do
  { compute_athematic_presentk sstem wstem short entry third
```

```
  ; compute_athematic_impftk sstem wstem short entry
  ; compute_athematic_optativek wstem short entry
  ; compute_athematic_imperativek sstem wstem entry
  }
;
** Gana 9 **
value compute_athematic_present9a strong weak short entry third =
  let conjugs person suff = (person, fix strong suff)
  and conjugw_v person suff = (person, fix short suff) (* vowel suffix *)
  and conjugw_c person suff = (person, fix weak suff) (* consonant suffix *) in do
  { enter1 entry (Conju (presa 9)
   [ (Singular,
         [ conjugs First "mi"
         ; conjugs Second "si"
         ; check entry 9 third (conjugs Third "ti")
         ])
   ; (Dual,
         [ conjugw_c First "vas"
         ; conjugw_c Second "thas"
         ; conjugw_c Third "tas"
         ])
   ; (Plural,
         [ conjugw_c First "mas"
         ; conjugw_c Second "tha"
         ; conjugw_v Third "anti"
         ])
   ])
  ; let f_pstem = rfix short "at" in
     record_part (Ppra_ 9 Primary short f_pstem entry) (* follows 3rd pl *)
  }
;
value compute_athematic_present9m weak short entry third =
  let conjugw person suff = match code suff with
       [ [ c :: _ ] → let w = if vowel c then short else weak in
                            (person, fix w suff)
       | [] → error_suffix 16
       ] in
  enter1 entry (Conju (presm 9)
   [ (Singular,
```

```
        [ conjugw First "e"
        ; conjugw Second "se"
        ; check entry 9 third (conjugw Third "te")
        ])
   ; (Dual,
        [ conjugw First "vahe"
        ; conjugw Second "aathe"
        ; conjugw Third "aate"
        ])
   ; (Plural,
        [ conjugw First "mahe"
        ; conjugw Second "dhve"
        ; conjugw Third "ate"
        ])
   ])
;
value compute_athematic_impft9a strong weak short entry =
  let conjugs person suff = (person, fix_augment strong suff)
  and conjugw person suff = match code suff with
      [ [ c :: _ ] → let w = if vowel c then short else weak in
                         (person, fix_augment w suff)
      | [] → error_suffix 6
      ] in
  enter1 entry (Conju (impfta 9)
   [ (Singular,
        [ conjugs First "am"
        ; conjugs Second "s"
        ; conjugs Third "t"
        ])
   ; (Dual,
        [ conjugw First "va"
        ; conjugw Second "tam"
        ; conjugw Third "taam"
        ])
   ; (Plural,
        [ conjugw First "ma"
        ; conjugw Second "ta"
        ; conjugw Third "an"
        ])
   ])
```

```
;
value compute_athematic_impft9m weak short entry =
  let conjugw person suff = match code suff with
      [ [ c :: _ ] → let w = if vowel c then short else weak in
                      (person, fix_augment w suff)
      | [] → error_suffix 13
      ] in
  enter1 entry (conjug_impft_m 9 conjugw)
;
value compute_athematic_optative9a weak short entry =
  let conjugw person suff = match code suff with
      [ [ c :: _ ] → let w = if vowel c then short else weak in (* tjs y- *)
                      (person, fix w suff)
      | [] → error_suffix 14
      ] in
  enter1 entry (conjug_opt_ath_a 9 conjugw)
;
value compute_athematic_optative9m short entry =
  let conjugw person suff = (person, fix short suff) in (* suff starts with ii *)
  enter1 entry (conjug_opt_ath_m 9 conjugw)
;
value compute_athematic_imperative9a strong weak short vow root entry =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = match code suff with
      [ [ c :: _ ] → let w = if vowel c then short else weak in
                      (person, fix w suff)
      | [] → (person, fix weak "")
      ]
  and conjugw2 person suff = (person, fix root suff) in
  enter1 entry (Conju (impera 9)
   [ (Singular,
        [ conjugs First "aani"
        ; if vow then conjugw Second "hi"
          else conjugw2 Second "aana" (* no nii suffix for consonant root *)
        ; conjugs Third "tu"
        ])
   ; (Dual,
        [ conjugs First "aava"
        ; conjugw Second "tam"
        ; conjugw Third "taam"
```

```
          ])
    ; (Plural,
          [ conjugs First "aama"
          ; conjugw Second "ta"
          ; conjugw Third "antu"
          ])
    ])
;
value compute_athematic_imperative9m strong weak short root entry =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = match code suff with
      [ [ c :: _ ] → let w = if vowel c then short else weak in
                      (person, fix w suff)
      | [] → (person, fix weak "")
      ] in
  compute_imp_ath_m 9 conjugs conjugw entry
;
value compute_active_present9 sstem wstem short vow stem entry third = do
  { compute_athematic_present9a sstem wstem short entry third
  ; compute_athematic_impft9a sstem wstem short entry
  ; compute_athematic_optative9a wstem short entry
  ; compute_athematic_imperative9a sstem wstem short vow stem entry
  }
and compute_middle_present9 sstem wstem short stem entry third = do
  { compute_athematic_present9m wstem short entry third
  ; compute_athematic_impft9m wstem short entry
  ; compute_athematic_optative9m short entry
  ; compute_athematic_imperative9m sstem wstem short stem entry
  ; record_part_m_ath (pprm 9) short entry (* short and not wstem *)
  }
;
value compute_present9 sstem wstem short vow stem entry third pada padam =
  match voices_of_gana 9 entry with
  [ Para → if pada then
                compute_active_present9 sstem wstem short vow stem entry third
           else emit_warning ("Unexpected␣middle␣form:␣" ^ entry)
  | Atma → if padam then emit_warning ("Unexpected␣active␣form:␣" ^ entry)
           else compute_middle_present9 sstem wstem short stem entry third
  | Ubha → let thirda = if pada then third else []
           and thirdm = if pada then [] else third in do
```

```
              { compute_active_present9 sstem wstem short vow stem entry thirda
              ; compute_middle_present9 sstem wstem short stem entry thirdm
              }
  ]
;
```

Benedictive/precative, formed from *conjug_optativea* with aorist stem
NB. Whitney§837 makes it an optative mode of the root aorist

```
value conjug_benedictivea conj weak entry =
  let conjugw person suff = (person, fix weak suff) in
  enter1 entry
  (Conju (fbenea conj)
   [ (Singular,
        [ conjugw First "yaasam"
        ; conjugw Second "yaas" (* ambig opt *)
        ; conjugw Third "yaat" (* ambig opt *)
        ])
   ; (Dual,
        [ conjugw First "yaasva"
        ; conjugw Second "yaastam"
        ; conjugw Third "yaastaam"
        ])
   ; (Plural,
        [ conjugw First "yaasma"
        ; conjugw Second "yaasta"
        ; conjugw Third "yaasur"
        ])
   ])
;
value conjug_benedictivem conj sibstem entry =
  let conjug person suff = (person, fix sibstem suff) in
  enter1 entry
  (Conju (fbenem conj)
   [ (Singular,
        [ conjug First "iiya"
        ; conjug Second "ii.s.thaas"
        ; conjug Third "ii.s.ta"
        ])
   ; (Dual,
        [ conjug First "iivahi"
```

```
        ; conjug Second "iiyaasthaam"
        ; conjug Third "iiyaastaam"
        ])
   ; (Plural,
        [ conjug First "iimahi"
        ; conjug Second "ii.dhvam"
        ; conjug Third "iiran"
        ])
   ])
;
value compute_benedictive rstem entry =
   (* Macdonell§150 Kale§960 Whitney§924 Henry§298 *)
  let bene_stem = let ps_stem = passive_stem entry rstem in
      match entry with (* Deshpande gram p328 *)
      [ "daa#1" | "paa#1" | "sthaa#1" | "haa#1" → (* not "j~naa#1" *)
            match ps_stem with
            [ [ 4 (* ii *) :: rest ] → [ 10 (* e *) :: rest ] (* ii -¿ e *)
            | _ → failwith "Anomaly␣bene_stem"
            ] (* NB Deshpande: also j naayaat *)
      | "puu#1" → revcode "punii" (* weak gana 9 puniiyaat Vi.s.nu sahasr. *)
      | _ → ps_stem
      ] in do
  { conjug_benedictivea Primary bene_stem entry (* productive, although rare *)
     (* middle very rare: viik.si.siiran et pratipatsiiran in Abhisamayaalafkaara (David Rei-
gle) and k.r.sii.s.ta in BhP and stotras (Harry Spier) *)
  ; match entry with
     [ "bhuu#1" → let sibstem = revcode "bhavi.s" in
          conjug_benedictivem Primary sibstem entry (* bhavi.sii.s.ta *)
     | "k.r#1" → let sibstem = revcode "k.r.s" in (* k.r.sii.s.ta *)
          conjug_benedictivem Primary sibstem entry (* Kanakadhaarastotra *)
     | "iik.s" → let sibstem = revcode "iik.si.s" in
          conjug_benedictivem Primary sibstem entry (* viik.si.siiran *)
     | "j~naa#1" → let sibstem = revcode "j~naas" in
          conjug_benedictivem Primary sibstem entry (* j naasi.s.ta Deshpande *)
     | "daa#1" → let sibstem = revcode "daas" in
          conjug_benedictivem Primary sibstem entry (* daasi.s.ta Deshpande *)
     | "pad#1" → let sibstem = revcode "pats" in
          conjug_benedictivem Primary sibstem entry (* pratipatsiiran *)
     | "m.r" → let sibstem = revcode "m.r.s" in
          conjug_benedictivem Primary sibstem entry (* m.r.sii.s.ta P{1,3,61} *)
```

```
    | "luu#1" → let sibstem = revcode "lavi.s" in
        conjug_benedictivem Primary sibstem entry (* lavi.sii.s.ta P{3,4,116} *)
    | _ → ()
    ]
  }
;
(****************)
(* Future system *)
(****************)
```

Similar to *compute_thematic_paradigm_act*

```
value compute_futurea conj stem entry =
  let conjug person suff = (person, fix stem suff) in do
  { enter1 entry (Conju (ffutura conj)
   [ (Singular,
        [ conjug First "aami"
        ; conjug Second "asi"
        ; conjug Third "ati"
        ])
   ; (Dual,
        [ conjug First "aavas"
        ; conjug Second "athas"
        ; conjug Third "atas"
        ])
   ; (Plural,
        [ conjug First "aamas"
        ; conjug Second "atha"
        ; conjug Third "anti"
        ])
   ])
  ; record_part (Pfuta_ conj stem entry)
  }
;
value compute_futurem conj stem entry =
  let conjug person suff = (person, fix stem suff) in do
  { enter1 entry (Conju (ffuturm conj)
   [ (Singular,
        [ conjug First "e"
        ; conjug Second "ase"
        ; conjug Third "ate"
```

```
        ])
  ; (Dual,
        [ conjug First "aavahe"
        ; conjug Second "ethe"
        ; conjug Third "ete"
        ])
  ; (Plural,
        [ conjug First "aamahe"
        ; conjug Second "adhve"
        ; conjug Third "ante"
        ])
  ])
 ; record_part_m_th pfutm stem entry
 }
;
(* Conditional - preterit of future, built from imperfect on future stem *)
(* where non-performance of the action is implied - pluperfect conditional *)
(* used in antecedent as well as in consequent clause - Apte§216 *)
(* "si␣vous␣étiez␣venu,␣vous␣l'auriez␣vue" *)
value compute_conda conj stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (fconda conj) (thematic_preterit_a conjug))
;
value compute_condm conj stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (fcondm conj) (thematic_preterit_m conjug))
;
value compute_future stem entry =
  match entry with
    [ "as#1" → () (* uses bhuu *)
    | "iiz#1" | "lii" | "knuu" | "baadh" → do (* Para allowed in future *)
          { compute_futurea Primary stem entry
          ; compute_futurem Primary stem entry
          }
    | _ → match voices_of entry with
        [ Para → do (* active only *)
          { compute_futurea Primary stem entry
          ; match entry with (* conditional or atma on demand *)
            [ "grah" | "jiiv" | "bhuu#1" | "zaas" | "stu" | "sm.r" | "haa#1"
                    → compute_conda Primary stem entry
```

```
        | "khaad" → compute_futurem Primary stem entry
        | _ → ()
        ]
      }
    | Atma → (* middle only *)
      compute_futurem Primary stem entry
    | (* both *) _ → do
      { compute_futurea Primary stem entry
      ; compute_futurem Primary stem entry
      ; match entry with (* rare conditional *)
        [ "i" | "k.r#1" | "gam" | "ji" | "j~naa#1" | "tap" | "daa#1"
        | "nii#1" | "bandh" | "budh#1" | "m.r" | "yaj#1" | "sthaa#1" → do
            { compute_conda Primary stem entry
            ; compute_condm Primary stem entry
            }
        | _ → ()
        ]
      }
    ]
  ]
;
value compute_future_ca stem entry = do
  { compute_futurea Causative stem entry
  ; compute_futurem Causative stem entry
  ; match entry with (* rare conditional *)
    [ "j~naa#1" → do
        { compute_conda Causative stem entry
        ; compute_condm Causative stem entry
        }
    | _ → ()
    ]
  ; record_part_m_th pcausfm stem entry
  }
;
(* Possible intercalating vowel i for se.t and ve.t roots Whitney§935 *)
(* intercalates returns a set of possible intercalations. *)
(* 3 indicates metathesis: ar becomes ra by ar_ra below *)
(* 4 is specific to naz nasalisation *)
(* This information should be lexicalised with a generative lexicon. *)
value intercalates root =
```

```
let anit = [ 0 ] (* no intercalation *)
and set = [ 1 ] (* intercalate i *)
and vet = [ 0; 1 ] (* intercalate i optionally *)
    (* NB for likh and vij 0 means intercalate i on weak stem *)
and setl = [ 2 ] (* intercalate ii *)
and serb = [ 1; 2 ] (* intercalate i or ii *) in fun (* rstem *)
 [ [] → error_empty 10
 | [ 7; 45 (* v.r *) ] → serb (* v.r#1 and v.r#2 *)
 | [ 7 (* -.r *) :: _ ] → set
 | [ 8 (* -.rr *) :: _ ] → serb
 | [ 6; 48 (* suu#1 *) ] → vet
 | [ 6 (* -uu *) :: _ ] → set (* Kale p. 186 *)
 | [ c :: r ] →
     if vowel c then
        if all_consonants r then
           match root with
           [ "k.sii" | "ji" | "nii#1" | "vaa#3" | "zii#1" | "su#2"
           | "stu" | "sru" | "haa#1" → vet
           | ".dii" | "nu#1" | "yu#1" | "yu#2" | "ru" | "zri"
           | "k.su" | "k.s.nu" | "snu" (* Kale *) | "zuu"
               → set
           | _ → anit
           ]
        else set
     else if semivowel c then set
     else match root with
          [ "k.rt#1" | "c.rt" | "ch.rd" | "t.rd" | "n.rt" (* Pan7,2,57 *)
          | "ak.s" | "a~nj" | "k.rp" | "k.lp" | "kram" | "k.sam"
          | "klid" | "kliz" | "gup" | "guh" | "ghu.s" | "jan" | "ta~nc"
          | "tap" | "tyaj#1" | "dah#1" | "d.rp" | "nam" | "naz"
          | "bandh" | "budh#1" | "bhaj" | "majj" | "man" | "m.rj"
          | "yam" | "ruh" | "labh" | "likh" | "vap#2" | "vas#1" | "vah#1"
          | "vij" | "vid#1" | "v.rj" | "v.rt#1" | "vrazc" | "sad#1" | "sah#1"
          | "sidh#2" | "svap" | "han#1" | "syand" (* WR: set atma, anit para *)
              → vet
          | "grah" → setl
          | "s.rj#1" → [ 3 ] (* sra.s.taa *)
          | "k.r.s" → [ 3 :: vet ] (* ar -¿ ra optionally *)
          | "bh.rjj" | "sp.rz#1" → [ 3 :: anit ] (* idem *)
          | "ad#1" | "aap" | "krudh#1" | "kruz" | "k.sip" | "k.sud"
```

```
            | "k.sudh#1" | "khid" | "chid#1" | "tud#1" | "tu.s" | "t.rp#1"
            | "tvi.s#1" | "diz#1" | "dih" | "du.s" | "duh#1" | "d.rz#1"
            | "dvi.s#1" | "nah" | "nij" | "nud" | "pac" | "pad#1" | "pi.s"
            | "pu.s#1" | "praz" | "bha~nj" | "bha.s" | "bhid#1"
            | "bhuj#1" | "bhuj#2" | "mih" | "muc#1" | "m.rz" | "yaj#1" | "yabh"
            | "yuj#1" | "yudh#1" | "ra~nj" | "rabh" | "ram" | "raadh" | "ric"
            | "ruj#1" | "rudh#1" | "rudh#2" | "ruh#1" | "lip" | "liz" | "lih#1"
            | "lup" | "vac" | "vap#1" | "vic" | "vid#2" | "viz#1" | "vi.s#1"
            | "vyadh" | "zak" | "zad" | "zap" | "zi.s" | "zudh" | "zu.s"
            | "zli.s" | "sa~nj" | "sic" | "sidh#1" | "s.rp" | "skand"
            | "sva~nj" | "svid#2" | "had"
              → anit
            | _ → set (* default all multisyllabic, gana 10, nominal verbs plus: "afg" |
"a~nc" | "an#2" | "arh" | "av" | "az#1" | "az#2" | "as#2" | "aas#2" | "indh" |
"inv" | "i.s#1" | "i.s#2" | "iik.s" | "iifkh" | "ii.d" | "iiz#1" | "uc" | "u~nch" |
"umbh" | "uuh" | ".rc#1" | ".rj" | ".rdh" | "edh" | "kafk" | "kam" | "kamp" | "ka.s" |
"kaafk.s" | "ku.n.th" | "ku.n.d" | "kup" | "krand" | "krii.d" | "khan" | "khaad" |
"gu~nj" | "gam" | "ghu.s" | "ghaat" | "ghuur.n" | "cand" | "cit#1" | "cumb" |
"chand" | "jak.s" | "jap" | "jalp" | "jinv" | "j.rmbh" | "tak" | "tan#1" | "tan#2" |
"tark" | "tvar" | "dagh" | "dabh" | "dham" | "dhva.ms" | "dhvan" | "nand" | "nind" |
"pa.th" | "pat#1" | "pi~nj" | "piz" | "ba.mh" | "bhand" | "bhaa.s" | "bhraaj" |
"ma.mh" | "ma.n.d" | "mad#1" | "mand#1" | "mlecch" | "yat#1" | "yaac" | "ra.mh" |
"rak.s" | "raaj#1" | "ruc#1" | "rud#1" | "lag" | "lafg" | "lafgh" | "lap" |
"lamb" | "laa~nch" | "la.s" | "lu.n.th" | "lok" | "loc" | "vad" | "vand" | "vam" |
"vaz" | "vas#2" | "vaa~nch" | "vaaz" | "vip" | "ven" | "vyath" | "vraj" | "vrii.d" |
"za.ms" | "zafk" | "zas" | "zaas" | "zuc#1" | "san#1" | "skhal" | "stambh" |
"spand" | "spardh" | "sp.rh" | "sphu.t" | "svan" | "has" | "hi.ms" *)
            ]
  ]
;
(* Whitney§631-§640 Bandharkar II p44 augment ii in present system 2nd class *)
value augment_ii = fun (* *)
  [ "an#2" | "rud#1" | "zvas#1" | "svap" | "jak.s" → True
    (* and thus "praa.n#1" too gives praa.niit *)
  | _ → False
  ]
;
```

Perfect passive participle

```
value intercalate_pp root rstem =
```

```
(* some redundancy with intercalates but really different, specially since the default is anit
for verbs ending with single consonant *)
  let anit = [ 0 ] (* no intercalation *)
  and set = [ 1 ] (* intercalate i *)
  and vet = [ 0; 1 ] (* intercalate i optionally *) in
  match rstem with
  [ [ c :: r ] →
      if vowel c then
          match root with
          [ "jaag.r" | "zii#1" → set
          | _ → anit
          ]
      else match r with
         [ [ v :: _ ] when vowel v →
               match root with
                 (* TODO utiliser intercalates sauf exceptions *)
               [ "radh" | "naz#1" | "trap#1" | "d.rp" | "druh#1" | "muh" | "jap"
               | "snih#1" | "snuh#1" (* P{7,2,45} *)
               | "i.s#1" | "sah#1" | "lubh" | "ru.s#1" | "ri.s" (* P{7,2,48} *)
               | "uuh" | "k.subh" | "tap" | "yat#1" | "ruup" | "vas#1" | "vas#4"
               | "zap" | "zas" | "zaas" | "h.r.s" (* P{7,2,...} *)
               | "zak" (* zakita P{7,2,17} (Kaazikaa) *)
               | "gaah" (* gaahita *)
               | "yas" (* aayasita *)
               | "kliz" | "puu#1" | "a~nc" (* P{7,2,51,53,50} *) → vet
               | "ghu.s" (* P{7,2,23} *) | "ka.s" (* P{7,2,22} *)
               | "dh.r.s" (* P{7,2,19} *)
               | "am" | "tvar" (* P{7,2,28} *) → vet (* but only set for -tvaa *)
               | "kas" | "k.sam" | "gup" | "dyut#1" | "dham" | "nud" | "m.rj" → vet
                 (* NB zaas vet for stem zaas but admits also zi.s only anit *)
               | "aj" | "a.t" | "at" | "an#2" | "az#2" | "aas#2" | "i.s#2"
               | "ii.d" | "iir" | "iiz#1" | "ii.s" | "iih" | "uc" | ".rc#1" | ".rj"
               | "ej" | "edh" | "kath" | "kal" | "kaaz" | "kiil" | "kuc" | "kup"
               | "ku.s" | "kuuj" | "k.rz" | "krii.d" | "klav" | "kvath"
               | "k.sar" | "k.sudh#1" | "k.svi.d" | "khaad" | "ga.n" | "gad" | "gal"
               | "granth" | "gha.t" | "ghaat" | "cak" | "ca.t" | "car" | "cal"
               | "cud" | "cur" | "chal" | "jiiv" | "jval" | "ta.d" | "tam" | "tul"
               | "t.r.s#1" | "tru.t" | "tvi.s#1" | "day" | "dal" | "dol" | "dhaav#1"
               | "dhiir" | "dhvan" | "na.t" | "nad" | "pa.th" | "pa.n" | "pat#1"
               | "piz" | "pii.d" | "pulak" | "puuj" | "prath" | "pru.s#1" | "phal"
```

```
            | "baadh" | "bha.n" | "bhas" | "bhaa.s" | "bhaas#1" | "bhuu.s"
            | "bhraaj" | "ma.mh" | "manth" | "mah" | "likh" | "mil" | "mi.s"
            | "miil" | "mud#1" | "mu.s#1" | "m.rg" | "yaac" | "rac" | "ra.n"
            | "ras" | "rah" | "raaj#1" | "ruc#1" | "rud#1" | "lag" | "lap" | "lal"
            | "la.s" | "las" | "lu.th" | "lul" | "lok" | "loc" | "vad" | "val"
            | "vas#2" | "vaaz"| "vaas#3" | "vid#1" | "vip"| "ven" | "vyath"
            | "vraj" | "vra.n" | "vrii.d" | "zubh#1" | "zcut#1" | "zrath"
            | "zlath" | "zlaagh" | "zvas#1" | ".s.thiiv" | "suuc"| "suud" | "sev"
            | "skhal" | "stan" | "stim" | "sthag" | "sphu.t" | "sphur" | "svad"
            | "svan" | "svar#1" | "has" | "hras" | "hraad" | "hlaad" | "hval"
                → set
            | "palaay" → set (* very special item *)
            | "grah" → set (* but will get ii *)
            | _ → anit
            ]
        | _ → match root with
            [ "umbh" | "muurch" | "mlecch" | "zrambh" (* vizrambhita *)
            | "skambh" (* vi.skabdha *) | "stambh" (* stabdha stabhita *)
            | "zvas" (* samaazvasta *) → vet
            | "cak.s" | "jak.s" | "bh.rjj" (* ca.s.ta bh.r.s.ta *)
            | "ra~nj" | "sa~nj" | "bandh" (* rakta sakta baddha *) → anit
            | _ → if aa_it root ∨ ii_it root ∨ u_it root ∨ uu_it root
                      then anit
                   else set
            ]
        ]
  | [] → error_empty 11
  ]
;
value intercalate_tvaa root rstem =
  let set = [ 1 ] (* intercalate i *)
  and anit = [ 0 ] (* no intercalation *)
  and vet = [ 0; 1 ] (* intercalate i optionally *) in
  match root with
  [ "zam#2" (* unused without preverb *)
  | "av" → [] (* WR no absol *)
  | "ka.s" | "dh.r.s" | "am" | "tvar" | ".r.s" → set
  | "nud" → anit
  | _ → if uu_it root ∨ u_it root then vet
        else intercalate_pp root rstem
```

```
  ]
;
value is_set_pp root rstem = List.mem 1 (intercalate_pp root rstem)
and is_anit_pp root rstem = List.mem 0 (intercalate_pp root rstem)
and is_set_tvaa root rstem = List.mem 1 (intercalate_tvaa root rstem)
and is_anit_tvaa root rstem = List.mem 0 (intercalate_tvaa root rstem)
;
type ppp_suffix =
  [ Na of Word.word
  | Tia of Word.word (* allowing i intercalation *)
  | Ta of Word.word (* not allowing intercalation *)
  | Va of Word.word
  | Ka of Word.word
  ]
;
(* The ppp constructors as postfix operators applied to a stem given as string *)
value sNa s = Na (revstem s)
and sTa s = Ta (revstem s)
and sTia s = Tia (revstem s)
and sVa s = Va (revstem s)
;
(* Computes the Primary ppp stems of roots *)
value compute_ppp_stems entry rstem =
  match entry with
      (* First participles in -na *)
    [ "vrazc" → [ sNa "v.rk" ] (* exception - v.rk root stem of vrazc *)
    (* Most roots starting with 2 consonants take -na P{8,2,43} *)
    (* but not "k.svi.d" "zrath" *)
    | "iir" | "und" | "k.rr" | "klid" | "k.saa" | "k.sii" | "k.sud" | "k.svid"
    | "khid" | "g.rr#1" | "glai" | "chad#1" | "chid#1" | "ch.rd" | "j.rr"
    | ".dii" | "tud#1" | "t.rd" | "t.rr" | "dagh" | "d.rr" | "dev" | "draa#1"
    | "draa#2" | "nud" | "pad#1" | "pii" | "p.rr" | "pyaa" | "bha~nj"
    | "bhid#1" | "bhuj#1" | "majj" | "man" | "mid" | "mlaa" | "ri" | "ruj#1"
    | "lii" | "luu#1" | "vij" | "vid#2" | "vrii" | "vlii" | "zad" | "zuu"
    | "z.rr" | "sad#1" | "skand" | "st.rr" | "styaa" | "syand" | "svid#2" | "had"
    | "haa#2" →
      (* except lag which is "nipaatana" (exception) P{7,2,18} *)
      let ppna w = [ Na w ] in
      match rstem with
      [ [ 2 :: _ ] | [ 4 :: _ ] | [ 6 :: _ ] (* stems in aa ii uu *)
```

```
    → ppna rstem
  | [ 3 :: r ] → ppna [ 4 :: r ] (* piina rii.na vrii.na *)
  | [ 8 :: r ] (* .rr -¿ r+vow *) →
    let vow =
      match entry with
      [ "p.rr" → 6 (* uu *)
      | _ → 4 (* ii *)
          (* "k.rr" — "g.rr#1" — "j.rr" — "t.rr" — "d.rr" — "st.rr" *)
      ] in
    let stem = [ 43 (* r *) :: [ vow :: r ] ] in
    match entry with
    [ "p.rr" → [ Ta stem :: ppna stem ] (* alternate form puurta *)
    | "st.rr" → [ Ta [ 7 :: r ] :: ppna stem ] (* alternate form st.rta *)
    | _ → ppna stem
    ]
  | [ 11 :: r ] (* ai *) → ppna [ 2 :: r ] (* glaana *)
  | [ 19 :: _ ] | [ 20 :: _ ] (* g gh *) → ppna rstem (* daghna *)
  | [ 24 :: r ] (* j *) →
    let stem = match r with
               [ [ 26 :: s ] (*  n *) (* bhagna *)
               | [ 24 :: s ] (* j *) → [ 19 :: s ] (* magna *)
               | _ → [ 19 :: r ] (* revert to guttural g *)
               ] in
    ppna stem
  | [ 34 (* d *) :: ([ 36 (* n *) :: _ ] as r) ] →
    (* d is dropped eg und skand *)
    let ppn = ppna r in
    match entry with
    [ "und" → [ sTa "ud" :: ppn ] (* for utta and abs -udya *)
    | _ → ppn
    ]
  | [ 34 (* d *) :: r ] →
    (* assimilation of d to n - special sandhi Macdonnel§60 foot 1 *)
    let ppn = ppna [ 36 (* n *) :: r ] in (* en fait il faudrait d'+n-¿nn *)
    match entry with
    [ "vid#2" → [ Ta rstem :: ppn ] (* 2 forms *)
    | "nud" → [ Ta rstem :: [ Tia rstem :: ppn ] ] (* 3 forms *)
    | _ → ppn
    ]
  | [ 36 :: ([ 1 :: r ] as w) ] (* -an *) →
```

```
            [ Ta w :: ppna [ 2 :: r ] ] (* mata+maana *)
    | [ 43 (* r *) :: r ] → ppna rstem (* iir.na *)
    | [ 45 (* v *) :: [ 10 (* e *) :: r ] ] → (* dev *)
            ppna [ 6 (* uu *) :: [ 42 (* y *) :: r ] ] (* dyuuna *)
    | _ → failwith ("Unexpected␣ppp␣in␣-na␣for␣" ^ entry)
    ] (* end participles in -na *)
  | "pac" → [ sVa "pak" ] (* exception P{8.2.51} *)
  | "zu.s" → [ Ka rstem ] (* exception P{8.2.52} *)
  | _ → (* otherwise participle in -ta (Panini kta) *)
        let ppstems =
    let ppstem = match entry with
        [ "dhaa#1" → revcode "hi" (* double weakening hi-ta P{7,4,42} *)
        | "bh.rjj" → [ 124; 7; 40 ] (* bh.rj' - mrijification of truncate *)
        | ".rc#1" → revcode "arc" (* strong *)
        | ".rj" → revcode "arj" (* strong *)
        | "k.svi.d" → revcode "k.sve.d"
        | "vip" → revcode "vep"
        | "m.rg" → revcode "marg" (* strong *)
        | "jak.s" → revcode "jagh" (* jagdha *)
        | "trai" → revcode "traa" (* glai given in -na section *)
        | "k.san" → revcode "k.sa" (* removal of final nasal *)
        | "gam" → revcode "ga" (* P{6,4,37} *)
        | "tan#1" → revcode "ta"
        | "nam" → revcode "na"
        | "yam" → revcode "ya"
        | "ram" → revcode "ra"
        | "van" → revcode "va"
        | "han#1" → revcode "ha" (* also "man" mata given with maana *)
        | "khan" → revcode "khaa" (* P{6,4,42} lengthening of vowel *)
        | "jan" → revcode "jaa" (* id *)
        | "san#1" → revcode "saa" (* id *)
        | "am" → revcode "aan" (* -am -¿ -aan P{6,4,15} Wh§955a *)
        | "kam" → revcode "kaan"
        | "kram" → revcode "kraan"
        | "cam" → revcode "caan"
        | "k.sam" → revcode "k.saan"
        | "dam#1" → revcode "daan"
        | "bhram" → revcode "bhraan"
        | "vam" → revcode "vaan"
        | "zram" → revcode "zraan"
```

```
| "zam#1" | "zam#2" → revcode "zaan"
| "dhvan" → revcode "dhvaan" (* id. for final n *) (* Wh§955a *)
| "daa#2" → revcode "di" (* aa -¿ i P{7,4,40} *)
| "maa#1" → revcode "mi"
| "zaa" → revcode "zi"
| "saa#1" → revcode "si"
| "sthaa#1" → revcode "sthi"
| "diiv#1" → revcode "dyuu" (* iiv -¿ yuu *)
| "siiv" → revcode "syuu"
| "daa#1" → revcode "dad" (* ad hoc P{7,4,46} *)
| "dham" → revcode "dhmaa" (* P{7,3,78} *)
| "dhaav#2" → revcode "dhau"
| "dhv.r" → revcode "dhuur"
| "puuy" → revcode "puu"
| "bhi.saj#2" → revcode "bhi.sajy"
| "skambh" → revcode "skabh" (* skambh -¿ skabh *)
| "stambh" → revcode "stabh" (* stambh -¿ stabh *)
| "zrath" → revcode "zranth"
| "muurch" → revcode "muur" (* muurta *)
| "av" → revcode "uu" (* uuta *)
| "i" | ".r" | "k.r#1" | "kyaa" | "khyaa" | "gu~nj" | "gh.r"
| "ghraa" | "ci" | "cyu" | "ji" | "daa#3" | "du" | "dru#1" | "dh.r"
| "dhyaa" | "dhru" | "nu#1" | "praa#1" | "bh.r" | "mi" | "m.r"
| "yaa#1" | "yu#1" | "yu#2" | "raa#1" | "ru" | "va~nc" | "vaa#2"
| "v.r#1" | "v.r#2" | "zaas" | "zri" | "zru" | "si" | "su#2"
| "s.r" | "stu" | "snaa" | "snu" | "smi" | "sm.r" | "haa#1" | "hi#2"
| "hu" | "h.r#1" → rstem
 (* roots ending in a vowel do not take passive_stem in general ? *)
 (* vérifier forme passive pour racines ci-dessus *)
| _ → passive_stem entry rstem (* possibly duhified and mirjified *)
] in [ Ta ppstem :: match entry with
         [ ".rc#1" | ".rj" | "k.svi.d" | "ba.mh" | "ma.mh" | "manth"
         | "m.rg" | "yaj#1" | "vyadh" | "grah" | "vrazc" | "praz"
         | "zrath" | "svap" | "stambh" →
               [ Tia ppstem ] (* avoids *ma.mhita *)
         | "vaz" | "vac" | "vap" | "vap#1" | "vap#2" | "vad"
         | "vas#1" | "vas#4" →
               [ Tia rstem; Tia ppstem ]
         | "guh" → [ Tia (revstem "guuh") ] (* P{6,4,89} *)
         | _ → [ Tia rstem ] (* standard Paninian way *)
```

```
                    ]
                ] in
            let extra_forms =
            match entry with (* supplementary forms *)
            [ "a~nc" → [ sNa "ak" :: [ sTia "a~nc" ] ] (* "akna", "a~ncita" *)
            | "kuc" → [ sTia "ku~nc" ] (* "ku~ncita" *)
            | "grah" → [ sTa "g.rbh" :: [ sTia "g.rbh" ] ] (* "g.rbhiita" *)
            | "car" → [ sNa "ciir" ] (* irreg. na ppp "ciir.na" *)
            | "tvar" → [ sNa "tuur" ] (* irreg. na ppp "tuur.na" *)
            | "du" → [ sNa "duu" ] (* "duuna" *)
            | "lag" → [ sNa "lag" ] (* irreg. na ppp "lagna" P{7,2,18} *)
            | "druh#1" → [ sTa "druh" ] (* opt. duhify "druu.dha" *)
            | "dhuu#1" → [ sTa "dhu" ]
            | "muh" → [ sTa "muh" ] (* opt. duhify "muu.dha" *)
            | "mlecch" → [ sTa "mlich" ] (* "mli.s.ta" *)
            | "vaa#3" → [ sTa "u" ]
            | "sah#1" → [ sTa "soh" ]
            | "suu#1" → [ sTa "su" :: [ sNa "suu" ] ] (* suta suuna *)
            | "snih#1" → [ sTa "snih" ] (* opt. duhify "snii.dha" *)
            | "snuh#1" → [ sTa "snuh" ] (* opt. duhify "snuu.dha" *)
            | "haa#1" → [ sNa "hii" :: [ sNa "haa" ] ] (* irreg. na ppp *)
            | "hrii#1" → [ sNa "hrii" ] (* "hrii.na" *)
            | _ → []
            ] in extra_forms @ ppstems
    ]
;
```

Metathesis -arx -¿ -rax (x=.s.t ou jy)
similaire order/ordre meter/mètre master/maître manner/manière

```
value ar_ra = fun
  [ [ c :: [ 43 :: [ 1 :: r ] ] ] → [ c :: [ 1 :: [ 43 :: r ] ] ]
  | [ c :: [ 43 :: [ 2 :: r ] ] ] → [ c :: [ 2 :: [ 43 :: r ] ] ]
  | w → failwith ("metathesis failure " ^ Canon.rdecode w)
  ]
;
(* Stems used for periphrastic futur, infinitive, and gerundive in -tavya *)
(* Redundancy and discrepancies with intercalates ought to be addressed. *)
value perstems rstem entry =
  let sstem = strong_stem entry rstem in
  let inter = match rstem with
```

```
[ [ 7; 45 (* v.r *) ] → [ 1; 2 ] (* i/ii* v.r#1 and v.r#2 *)
| [ 7 (*.r *) :: _ ] → [ 0 ]
| _ → match entry with
        [ "gam" | "dham" | "praz" | "vaa#3" | "za.ms" | "han#1" | "huu"
          → [ 0 ]
        | "v.rj" → [ 1 ]
        | "zuc#1" → [ 0; 1 ] (* zoktum *)
        | "d.rz#1" | "sp.rz#1" → [ 3 ] (* ar -¿ ra dra.s.tum *)
        | "k.r.s" | "bh.rjj" → [ 0; 3 ] (* berk *)
        | "naz#1" → [ 0; 1; 4 ] (* berk - (1 not in WR) *)
        | "radh" | "trap#1" | "d.rp" | "druh#1" | "muh" | "rudh#2"
        | "snih#1" | "snuh#1" (* P{7,2,45} *)
        | "i.s#1" | "sah#1" | "lubh" | "ru.s#1" | "ri.s" (* P{7,2,48} *)
            → [ 0; 1 ]
        (* TODO: also optionally all uu − it roots - P{7,2,44} *)
        | _ → intercalates entry rstem
        ]
] in
map insert_sfx inter
  where insert_sfx = fun
    [ 0 → match entry with
           [ "majj" → code "mafk" (* Whitney§936a *)
           | "jan" → code "jaa"
           | "dham" → code "dhmaa"
           | "nij" → code "nej" (* for gana 3 *)
           | "vah#1" → code "voh" (* vo.dhaa P{6,3,112} *)
           | "sah" → code "soh" (* so.dhum P{6,3,112} *)
           | "likh" | "vij" → rev [ 3 :: rstem ] (* i with weak stem *)
           | "vrazc" → code "vraz" (* ought to be truncated by int sandhi *)
           | "za.ms" → code "zas"
           | "huu" → code "hvaa"
           | _ → rev (match rstem with
                   [ [ c :: r ] → match c with
                        [ 10 | 11 | 12 | 13 → [ 2 :: r ] (* eg gai -¿ gaa *)
                        | _ → sstem
                        ]
                   | [] → error_empty 12
                   ])
           ]
    | 1 → let w = match entry with
```

```
              [ "uc" | "mil" | "sphu.t" | "sphur" → rstem (* PB for Inf ? *)
              | "guh" → revcode "guuh" (* P{6,4,89} *)
              | "sad#1" → revcode "siid"
              | "sp.rh" → revcode "sp.rhay"
              | "haa#1" → revcode "jah"
              | _ → sstem
              ] in
            sandhi w (code "i") (* sandhi sanitizes a possible j' or h' *)
      | 2 → sandhi sstem (code "ii") (* grah *)
      | 3 → rev (ar_ra sstem) (* metathesis: kra.s.taa bhra.s.taa dra.s.taa *)
      | 4 → code "na.mz" (* exception naz *)
      | _ → failwith "perstems:␣weird␣intercalate␣code"
      ]
;
value compute_future_gen rstem entry =
  let sstem = strong_stem entry rstem in
  let stems = map insert_sfx (intercalates entry rstem)
    where insert_sfx = fun
      [ 0 → let w = match entry with
            [ "naz" → revcode "nafk" (* Whitney§936a *)
            | "majj" → revcode "mafk" (* Whitney§936a *)
            | "d.rz#1" → revcode "drak" (* drak.sya *)
            | "gai" → revcode "gaa"
            | "jan" → revcode "jaa"
            | "nij" → revcode "nej" (* consistent with gana 3 *)
            | "bharts" → revcode "bhart"
            | "likh" | "vij" → [ 3 :: rstem ] (* i with weak stem (hack) *)
            | "vas#1" → revcode "vat" (* vatsyati Whitney§167 P{7,4,49} *)
            | "vrazc" → revcode "vrak" (* vrak.sya *)
            | "saa#1" → rstem (* saa si *)
            | _ → sstem (* for nij gana 3 *)
            ] in sandhi w (code "sya") (* eg dah -¿ dhak.sya *)
      | 1 → let w = match entry with
            [ "uc" | "␣mil" | "sphu.t" | "sphur" → rstem
            | "guh" → revcode "guuh" (* P{6,4,89} *)
            | "dabh" → revcode "dambh"
            | "nij" → revcode "ni~nj" (* consistent with gana 2 *)
            | "sad#1" → revcode "siid"
            | "vaa#3" → revcode "ve"
            | "haa#1" → revcode "jah"
```

```
            | "huu" → revcode "hve"
            | _ → sstem
            ] in sandhi w (code "i.sya")
      | 2 → sandhi sstem (code "ii.sya") (* grah *)
      | 3 → sandhi (ar_ra sstem) (code "sya") (* metathesis k.r.s bh.rjj s.rj *)
      | _ → failwith "Weird intercalate code"
      ] in
  iter mk_future stems
      where mk_future stem = match Word.mirror stem with
        [ [ 1 :: st ] → compute_future st entry
        | _ → error_empty 13
        ] (* Note that sandhi with sy would fail with finalize *)
;
value compute_future_10 rstem entry =
  let fsuf = revcode "i.sy" in
  match entry with
    [ "tul" → do (* 2 forms *)
        { compute_future (fsuf @ (revcode "tulay")) entry
        ; compute_future (fsuf @ (revcode "tolay")) entry
        }
    | _ → let stem = strengthen_10 rstem entry in
          let aystem = Word.mirror (sandhi stem [ 1; 42 ] (* ay *)) in
          let fstem = fsuf @ aystem in
          compute_future fstem entry
    ]
;
```

Passive system

```
value admits_passive = fun
  [ (* We filter out roots with no attested passive forms *)
    "an#2" | "av" | "as#1" | "ah" | "iiz#1" | "uc" | "kan" | "kuu"
  | "knuu" | "k.sar" | "k.si" | "kha.n.d" | "daa#2" | "dyut#1" | "dru#1"
  | "pat#2" | "paz" | "paa#2" | "pii" | "praa#1" | "bruu" | "ruc#1" | "vas#4"
  | "vidh#1" | "vip" | "vyac" | "zam#1" | "zi~nj" | "zrambh" | "zvit#1" | "sap#1"
  | "siiv" | "spaz#1" | "spardh" | "h.r#2" | "hrii#1"
  | "ma.mh" (* supplied by "mah" *) (* — "arh" — "k.lp" no ps but pfp *)
      → False
(* But "iiz#1" "uc" "kuu" "k.sar" "dru#1" "pii" "ruc#1" "vip" "zam#1" "zi~nj"
"zrambh" "siiv" "spardh" "hrii#1" admit ppp. *)
  | _ → True
```

```
  ]
;
value admits_ppp_abs = fun
  [ "ak.s" (* vedic a.s.ta overgenerates with a.s.tan *)
  | "ad#1" (* jak.s jagdha P{2,4,36} *)
  | "bruu" (* vac *)
  | "paz" (* d.rz *)
  | "as#1" | "kan" | "k.si" | "gaa#1" | "paa#2" | "praa#1" (* omit ved. praata *)
  | "bal" | "ma.mh" | "vaz" | "vyac" | "zaz" | "zam#2" | "zvit#1" | "sac"
  | "sap#1" | "h.r#2" (* — "spaz#1" *) → False
  | _ → True
  ]
;
```

Similar to *compute_thematic_middle*

```
value compute_passive_present verbal stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju verbal
   [ (Singular, let l =
         [ conjug First "e"
         ; conjug Second "ase"
         ; conjug Third "ate"
         ] in if entry = "tap" then [ conjug Third "ati" :: l ] else l
              (* Bergaigne exception tapyati *))
   ; (Dual,
         [ conjug First "aavahe"
         ; conjug Second "ethe"
         ; conjug Third "ete"
         ])
   ; (Plural,
         [ conjug First "aamahe"
         ; conjug Second "adhve"
         ; conjug Third "ante"
         ])
   ])
;
value compute_passive_imperfect verbal stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju verbal
   [ (Singular,
```

```
        [ conjug First "e"
        ; conjug Second "athaas"
        ; conjug Third "ata"
        ])
  ; (Dual,
        [ conjug First "aavahi"
        ; conjug Second "ethaam"
        ; conjug Third "etaam"
        ])
  ; (Plural,
        [ conjug First "aamahi"
        ; conjug Second "adhvam"
        ; conjug Third "anta"
        ])
  ])
;
value compute_passive_optative verbal stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju verbal
  [ (Singular,
        [ conjug First "eya"
        ; conjug Second "ethaas"
        ; conjug Third "eta"
        ])
  ; (Dual,
        [ conjug First "evahi"
        ; conjug Second "eyaathaam"
        ; conjug Third "eyaataam"
        ])
  ; (Plural,
        [ conjug First "emahi"
        ; conjug Second "edhvam"
        ; conjug Third "eran"
        ])
  ])
;
value compute_passive_imperative verbal stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju verbal
  [ (Singular,
```

```
        [ conjug First "ai"
        ; conjug Second "asva"
        ; conjug Third "ataam"
        ])
    ; (Dual,
        [ conjug First "aavahai"
        ; conjug Second "ethaam"
        ; conjug Third "etaam"
        ])
    ; (Plural,
        [ conjug First "aamahai"
        ; conjug Second "adhvam"
        ; conjug Third "antaam"
        ])
    ])
;
(* Same as (reversed) internal sandhi of (reversed) stem and "y" *)
value affix_y stem =
  [ 42 (* y *) :: Int_sandhi.restore_stem stem ]
;
value compute_passive_system conj root pastem = do
  { compute_passive_present (fpresp conj) pastem root
  ; compute_passive_imperfect (fimpftp conj) pastem root
  ; compute_passive_optative (foptp conj) pastem root
  ; compute_passive_imperative (fimperp conj) pastem root
  ; record_part_m_th (vpprp conj) pastem root
  }
;
(* NB. For gana 4 verbs passive differs from middle mostly by accent but distinction necessary
since different regime *)
value compute_passive conj root stem =
  let ps_stem = affix_y stem (* "y" marks passive *) in
  compute_passive_system conj root ps_stem
;
value compute_passive_raw root =
  let ps_stem = passive_stem root (revstem root) in
  compute_passive Primary root ps_stem
;
value compute_passive_10 root ps_stem =
  match root with
```

```
  [ "tul" → ((* no passive*))
  | _ → compute_passive Primary root ps_stem
  ]
;
value compute_passive_11 root ps_stem =
  match root with
  [ "adhvara" | "asuuya" | "iras" | "ka.n.du" | "karu.na" | "tapas"
  | "namas" → ((* no passive *))
  | _ → compute_passive Primary root ps_stem
  ]
;
```

Perfect system
Reduplication for perfect. *redup_perf* takes a string, and returns (s, w, o, e, b) where s is the (reversed) strong stem word, w is the (reversed) weak stem word, o is an optional lengthened stem word, e is a boolean flag (True if 2nd sg weak) b is a boolean flag (True if optional union-vowel i)
NB b=iopt not sufficient. See Whitney§797
Warning: baroque code ahead

```
value redup_perf root =
  let (revw, revs, revl) = match root with
      [ "ji" → stems "gi" (* palatal -¿ velar *)
      | "ci" → stems "ki" (* idem *)
      | "cit#1" → stems "kit" (* idem *)
      | "umbh" → stems "ubh" (* remove penultimate nasal *)
      | "sva~nj" → stems "svaj" (* idem *)
      | "han#1" → stems "ghan" (* velar h -¿ gh *)
      | "hi#2" → stems "ghi" (* idem *)
      | "guh" → stems "guuh" (* P{6,4,89} *)
      | "diiv#1" → stems "dev"
      | "dham" → stems "dhmaa"
      | "praz" → let w = revcode "pracch" in (w, w, w) (* Whitney§794c *)
      | "zaas" → let w = revcode root in (w, w, w) (* redup voy a, not i *)
      | _ → stems root (* NB: keep penultimate nasal "ta~nc" *)
      ] in
  match Word.mirror revw with (* ugly double reversal to get the stem *)
  [ [] → error_empty 14
  | [ 3 ] (* "i" *) → let wk = [ 4; 42 ] (* iiy P{7,4,69} *)
                      and st = [ 3; 42; 10 ] (* iye *) (* iyaya *)
                      and lg = [ 3; 42; 11 ] (* iyai *) (* iyaaya *) in
```

```
                    (rev st, rev wk, Some (rev lg), False, True)
  | [ c1 :: r ] →
      if vowel c1 then let (s, w) = match c1 with
         [ 1 (* a *) → let w = match r with
           [ [ c2 ] → if root = "az#1" then (revw @ [ 36; 2 ]) (* aan- az1 *)
                       else ([ c2; 2 (* aa *)])
           | [ 17; _ ] | [ 26; _ ] | [ 43; 22 ] | [ 43; 49 ]
               → (revw @ [ 36; 2 ])
               (* aan- for ak.s, a~nc, a~nj, arc (en fait .rc), arh *)
           | _ → (revw @ [ 36; 1 ] (* an- *))
           ] in (strong w, w)
         | 3 (* i *) → let wk = [ 4 (* ii *) :: if r = [ 47 ] (* i.s *) then r
                                                    else [ 42 (* y *) :: r ] ]
                       and st = [ 3; 42; 10 ] (* iye *) @ r in
                       (rev st, rev wk)
         | 5 (* u *) → let wk = [ 6 (* uu *) :: r ]
                       and redup = match root with
                           [ "vaz" → 2 | _ → 12 ] in
                       let st = [ 5; 45; redup ] (* uvo/uvaa *) @ r in
                       (rev st, rev wk)
         | 7 (* .r *) → let w = match r with
                     [ [ 22 ] | [ 35 ] | [ 47 ] → (* Whitney§788a *)
                      (revw @ [ 36; 2 ]) (* aan- for .rc1, .rdh, .r.s *)
                     | [] → [ 43; 1 ] (* ar for .r *)
                     | _ → revw
                     ] in (strong w, w)

         | _ (* aa ii uu *) → (revs, revw)
         ] in (s, w, None, False, False)
      else
      let (v, p, a) = lookvoy r (* p = prosodically long, a = vriddhi augment *)
         (* lookvoy computes the vowel v, and the two booleans p and a *)
         where rec lookvoy = fun
           [ [] → error_vowel 1
           | [ c2 ] → if vowel c2 then (c2, False, True)
                      else error_vowel 2
           | [ c2 :: r2 ] →
                      if vowel c2 then
                          let l = length (contract r2) in
                          let p = long_vowel c2 ∨ l > 1
                          and a = c2 = 1 (* a *) ∧ l = 1 in
```

```
                          (c2, p, a)
                        else lookvoy r2
      ] in (* c is reduplicating consonant candidate *)
let c = if sibilant c1 then match r with
            [ [] → error_vowel 3
            | [ c2 :: _ ] → if vowel c2 ∨ nasal c2 then c1
                              else if stop c2 then c2
                              else (* semivowel c2 *) c1
            ]
          else c1 in
let rv = (* rv is reduplicating vowel *)
  if v > 6 (* .r .rr .l dipht *) then match root with
    [ "ce.s.t" | "diiv#1" | "dev" | "sev" | "mlecch" | "vye"
        → 3 (* i *) (* vye for vyaa *)
    | _ → 1 (* a *) (* also bhuu elsewhere *)
    (* but Vedic k.lp etc have long aa Whitney§786a *)
    ]
  else match root with
    [ "maa#3" → 3 (* i *) (* analogy with present *)
    | "vyath" | "vyadh" | "vyaa" | "jyaa#1" | "pyaa" | "syand" | "dyut#1"
    | "myak.s" → 3
      (* Whitney§785 also "vyac" and ved. "tyaj#1"; "vyaa" treated other *)
    | "kan" | "mah" → 2 (* ved lengthened redup vowel Whitney§786a *)
    | _ → short v (* reduplicated vowel is short *)
    ]
and rc = (* reduplicating consonant *) match c with
  [ 17 | 18 (* k kh *) → 22 (* c *)
  | 19 | 20 | 49 (* g gh h *) → 24 (* j *)
  | 23 | 25 | 28 | 30 | 33 | 35 | 38 | 40 → c − 1 (* xh -¿ x *)
  | _ → c (* c by default *)
  ] in
let (affix, sampra) = match root with (* ya -¿ ii va -¿ uu *)
    [ "yaj#1" → ([ 3 (* i *)], Some (mrijify (revcode "iij")))
    | "vac" → ([ 5 (* u *)], Some (revcode "uuc"))
    | "vad" → ([ 5 (* u *)], Some (revcode "uud"))
    | "vap" | "vap#1" | "vap#2" → ([ 5 (* u *) ], Some (revcode "uup"))
    | "vaz" → ([ 5 (* u *)], Some (revcode "uuz"))
    | "vas#1" | "vas#4" → ([ 5 (* u *)], Some (revcode "uus"))
    | "vah#1" → ([ 5 (* u *)], Some (revcode "uuh"))
    | "vaa#3" → ([ 5 (* u *)], Some (revcode "uuv"))
```

```
          | _ → ([ rv; rc ], None)
          ]
     and vriddhi = match root with
          [ "vyadh" | "svap" | "grah" → True
            (* since special weak stem returned by stems *)
          | _ → a
          ] in
     let glue = match root with
          [ "sphur" | "sphu.t" → revaffix False affix (* no retroflexion *)
          | _ → revaffix True affix
          ] in
     let (weak, eweak, iopt) = match sampra with (* iopt = optional i *)
          [ Some weak → (weak, False, True)
          | None → if rc = c ∨ root ="bhaj" then match r with
            [ [ 1 :: w ] → match root with
              [ "jan" → (glue (revcode "j~n"), True, True)
              | "val" | "mah" → (glue revw, False, False)
              | _ → match w with
                [ [ c' ] when consonant c' →
                      (revaffix True [ 10 (* e *); c ] w, True, True)
                      (* roots of form c.a.c' with c,c' consonant or .m Scharf *)
                      (* cf. P{6,4,119-126} – ZZ may lead to hiatus *)
                | _ → (glue revw, False, False)
                ]
              ]
            | _ → (glue revw, False, False)
            ] else
              let (short, iopt) = match root with
                [ "gam" → (revcode "gm",True) (* actually i forbidden *)
                | "ghas" → (revcode "k.s",False)
                | "han#1" → (revcode "ghn",True)
                | "khan" → (revcode "khn",False)
                | _ → (revw, False)
                ] in (glue short, False, iopt)
          ]
      and strong = glue (if p then revw else revs)
      and longifvr = if vriddhi then revl else revs in
      let olong = if p then None else Some (glue longifvr) in
      (strong, weak, olong, eweak, iopt)
  ]
```

```
;
value compute_perfecta conj strong weak olengthened eweak iopt entry =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = (person, fix weak suff) in do
  { enter1 entry (Conju (fperfa conj)
   [ (Singular, let l = match olengthened with
      [ Some lengthened →
          let conjugl person suff = (person, fix lengthened suff) in
          [ conjugs First "a"
          ; conjugl First "a"
          ; let conjug = if eweak then conjugw else conjugs in
            conjug Second "itha"
          ; conjugl Third "a"
          ]
     | None →
          [ conjugs First "a" (* ex: aap -¿ aapa *)
          ; conjugs Second "itha"
          ; conjugs Third "a"
          ] @ if entry ="az#1" then
            let optstrong = revcode "aana.mz" in
            let conjugs person suff = (person, fix optstrong suff) in
          [ conjugs First "a"
          ; conjugs Second "itha"
          ; conjugs Third "a" (* actually also regular aaza Whitney§788a *)
          ] else [ ] (* Whitney§788a *)
     ] in if iopt then [ conjugs Second "tha" :: l ] else l)
   ; (Dual,
          [ conjugw First "iva"
          ; conjugw Second "athur"
          ; conjugw Third "atur"
          ])
   ; (Plural,
          [ conjugw First "ima"
          ; conjugw Second "a"
          ; if entry ="raaj#1" then (Third, code "rejur")
            else conjugw Third "ur" (* Henry: paptur véd. pat1 *)
          ])
   ])
  ; let pstem = if entry ="raaj#1" then (revcode "rej") else weak in
    record_part (Ppfta_ conj pstem entry)
```

```
  }
;
value compute_perfectm conj stem entry =
  let conjugw person suff = (person, fix stem suff) in do
  { enter1 entry (Conju (fperfm conj)
   [ (Singular, let l =
         [ conjugw First "e"
         ; conjugw Second "i.se"
         ; conjugw Third "e"
         ] in if entry = "guh" then
                   let juguhe = code "juguhe" in (* Whitney§793i *)
                   l @ [ (First, juguhe); (Third, juguhe) ]
              else l)
   ; (Dual,
         [ conjugw First "ivahe"
         ; conjugw Second "aathe"
         ; conjugw Third "aate"
         ])
   ; (Plural,
         [ conjugw First "imahe"
         ; conjugw Second "idhve"
         ; conjugw Third "ire"
         ])
   ])
  ; record_part_m_ath (vppftm conj) stem entry (* -aana *)
  }
;
value compute_perfect_c strong weak olengthened eweak iopt entry =
  match voices_of entry with
  [ Para → do
       { compute_perfecta Primary strong weak olengthened eweak iopt entry
       ; match entry with
         [ "cit#1" → do
             { compute_perfectm Primary weak entry
             ; compute_perfectm Primary (revcode "cikitr") entry (* WR *)
             }
         | "vac" → compute_perfectm Primary weak entry
(* record_part_m_ath ppftm weak entry (* anuucaana *) *)
         | _ → ()
         ]
```

```
      }
  | Atma → let stem = match entry with
                      [ "cak.s" | "ba.mh" → strong
                      | _ → weak
                      ] in
           compute_perfectm Primary stem entry
  | _ → do { compute_perfecta Primary strong weak olengthened eweak iopt entry
           ; let stem = match entry with
                        [ "kan" → revcode "cak" (* kan -¿ kaa *)
                        | _ → weak
                        ] in
             compute_perfectm Primary stem entry
           }
  ]
;
value compute_perfecta_aa stem entry =
  let conjug person suff = (person, fix stem suff) in do
  { enter1 entry (Conju perfa
   [ (Singular,
        [ conjug First "au"
        ; conjug Second "itha"
        ; conjug Second "aatha"
        ; conjug Third "au"
        ])
   ; (Dual,
        [ conjug First "iva"
        ; conjug Second "athur"
        ; conjug Third "atur"
        ])
   ; (Plural,
        [ conjug First "ima"
        ; conjug Second "a"
        ; conjug Third "ur"
        ])
   ])
  ; record_part (Ppfta_ Primary stem entry)
  }
;
value compute_perfectm_aa stem entry =
  let conjug person suff = (person, fix stem suff) in do
```

```
  { enter1 entry (Conju perfm
   [ (Singular,
         [ conjug First "e"
         ; conjug Second "i.se"
         ; conjug Third "e"
         ])
   ; (Dual,
         [ conjug First "ivahe"
         ; conjug Second "aathe"
         ; conjug Third "aate"
         ])
   ; (Plural,
         [ conjug First "imahe"
         ; conjug Second "idhve"
         ; conjug Third "ire"
         ])
   ])
  ; record_part_m_ath ppftm stem entry (* stem-aana *)
     (* middle part rare - eg cakraa.na pecaana anuucaana zepaana *)
  }
;
value compute_perfect_aa stem entry =
  match voices_of entry with
  [ Para → compute_perfecta_aa stem entry
  | Atma → compute_perfectm_aa stem entry
  | _ → do { if entry = "traa" then () (* to avoid parasitic tatra *)
                 else compute_perfecta_aa stem entry
             ; compute_perfectm_aa stem entry (* eg tatre WR *)
             }
  ]
;
(* dissymetric in i and u - problematic *)
value fix_dup weak suff mc = (* Gonda §18.I §6 *)
  let s = code suff in match s with
  [ [ c :: _ ] → match weak with
      [ [ 5 (* u *) :: l ] | [ 6 (* uu *) :: l ] (* eg stu *) →
         let sf = if vowel c then [ 45 (* v *) :: s ] else s in
         sandhi [ 5 :: l ] sf
      | [ 3 (* i *) :: l ] | [ 4 (* ii *) :: l ] (* eg nii *) →
         let sf = [ 42 (* y *) :: if vowel c then s
```

```
                                        else [ 3 (* i *) :: s ] ] in
        let isf = if mc (* multiconsonant roots eg krii *)
                       then [ 3 (* i *) :: sf ]
                    else sf in
        sandhi l isf
      | _ → sandhi weak s
      ]
  | _ → error_suffix 12
  ]
;
value multi_consonant root = match revcode root with
  [ [ v :: r ] → vowel v ∧ length r > 1
  | [] → error_empty 15
  ]
;
value compute_perfecta_v strong weak entry =
  let lengthened = lengthened weak
  and iforb = List.mem entry (* option intercalating i forbidden Whitney§797c *)
                [ "k.r#1"; "bh.r"; "v.r#2"; "s.r"; "dru#1"; "zru"; "stu"; "sru" ]
  and mc = multi_consonant entry in
  let conjugw person suff = (person, fix_dup weak suff mc)
  and conjugs person suff = (person, fix strong suff)
  and conjugl person suff = (person, fix lengthened suff) in do
  { enter1 entry (Conju perfa
   [ (Singular, let l =
        [ conjugs First "a"
        ; conjugl First "a"
        ; conjugs Second "tha"
        ; conjugl Third "a"
        ] in if iforb then l else [ conjugs Second "itha" :: l ])
   ; (Dual,
        [ conjugw First "va"
        ; conjugw Second "athur"
        ; conjugw Third "atur"
        ])
   ; (Plural,
        [ conjugw First "ma"
        ; conjugw Second "a"
        ; conjugw Third "ur"
        ])
```

```
  ])
  ; record_part (Ppfta_ Primary weak entry)
  }
;
value compute_perfectar conj stem entry =
  let conjugs person suff = (person, fix stem suff)
  and conjugl person suff = (person, fix (lengthened stem) suff) in do
  { enter1 entry (Conju (fperfa conj)
   [ (Singular,
        [ conjugs First "a"
        ; conjugl First "a"
        ; conjugs Second "itha"
        ; conjugl Third "a"
        ])
   ; (Dual,
        [ conjugs First "iva"
        ; conjugs Second "athur"
        ; conjugs Third "atur"
        ])
   ; (Plural,
        [ conjugs First "ima"
        ; conjugs Second "a"
        ; conjugs Third "ur"
        ])
   ])
  ; record_part (Ppfta_ conj stem entry)
  }
;
value compute_perfect_ril stem entry = (* -.rr or multiconsonant -.r *)
  match voices_of entry with
        [ Para → compute_perfectar Primary stem entry
        | Atma → compute_perfectm Primary stem entry
        | _ → do { compute_perfectar Primary stem entry
                 ; compute_perfectm Primary stem entry
                 }
        ]
;
value compute_perfectm_v weak mc entry =
  let conjugw person suff = (person, fix_dup weak suff mc) in do
  { enter1 entry (Conju perfm
```

```
  [ (Singular,
        [ conjugw First "e"
        ; conjugw Second "se"
        ; if entry = "m.r" then (Third, code "mamre")
          else conjugw Third "e"
        ])
  ; (Dual,
        [ conjugw First "vahe"
        ; conjugw Second "aathe"
        ; conjugw Third "aate"
        ])
  ; (Plural,
        [ conjugw First "mahe"
        ; conjugw Second "dhve"
        ; conjugw Third "ire"
        ])
  ])
 ; record_part_m_ath ppftm weak entry (* weak-aana *)
   (* middle part rare - eg cakraa.na pecaana anuucaana zepaana *)
 }
;
value compute_perfect_bhuu root =
 let conjug person suff = (person, fix (revcode "babhuu") suff) in
 enter1 root (Conju perfa
  [ (Singular,
        [ conjug First "va"
        ; conjug Second "tha"
        ; conjug Second "vitha"
        ; conjug Third "va"
        ])
  ; (Dual,
        [ conjug First "viva"
        ; conjug Second "vathur"
        ; conjug Third "vatur"
        ])
  ; (Plural,
        [ conjug First "vima"
        ; conjug Second "va"
        ; conjug Third "vur"
        ])
```

```
    ])
;
value compute_perfect_vid root = (* perfect in the sense of present *)
  let conjugw person suff = (person, fix (revcode "vid") suff)
  and conjugs person suff = (person, fix (revcode "ved") suff) in
  enter1 root (Conju perfa
    [ (Singular,
          [ conjugs First "a"
          ; conjugs Second "tha"
          ; conjugs Third "a"
          ])
    ; (Dual,
          [ conjugw First "va"
          ; conjugw Second "thur"
          ; conjugw Third "tur"
          ])
    ; (Plural,
          [ conjugw First "ma"
          ; conjugw Second "a"
          ; conjugw Third "ur"
          ])
    ])
;
value compute_perfect_ah root =
  enter1 root (Conju perfa
    [ (Singular,
          [ (Second, code "aattha")
          ; (Third, code "aaha")
          ])
    ; (Dual,
          [ (Second, code "aahathur")
          ; (Third, code "aahatur")
          ])
    ; (Plural,
          [ (Third, code "aahur")
          ])
    ])
;
value compute_perfect_vyaa root =
  (* This code is consistent with Dhaaturuupaprapa nca, except for middle 1st sg where it
```

```
lists "vivyaye" rather than "vivye" *)
  let weak = revcode "vivii" (* redup de vii Whitney§801c *)
  and strong = revcode "vivye" (* P{6,1,46} *)
  and long = revcode "vivyai" in
  let conjugw person suff = (person, fix_dup weak suff False)
  and conjugs person suff = (person, fix strong suff)
  and conjugl person suff = (person, fix long suff) in do
  { enter1 root (Conju perfa
   [ (Singular,
         [ conjugl First "a"
         ; conjugs First "a"
         ; conjugs Second "itha" (* P{7,2,66} *)
         ; conjugl Third "a"
         ])
   ; (Dual,
         [ conjugw First "va"
         ; conjugw Second "athur"
         ; conjugw Third "atur"
         ])
   ; (Plural,
         [ conjugw First "ma"
         ; conjugw Second "a"
         ; conjugw Third "ur"
         ])
   ])
  ; record_part (Ppfta_ Primary weak root)
  ; compute_perfectm_v weak False root (* mc=False! *)
  }
;
value compute_perfect_v strong weak entry =
  let mc = multi_consonant entry in
  match voices_of entry with
  [ Para → compute_perfecta_v strong weak entry
  | Atma → compute_perfectm_v weak mc entry
  | Ubha → do
      { compute_perfecta_v strong weak entry
      ; compute_perfectm_v weak mc entry
      }
  ]
;
```

```
value compute_perfect entry =
  match entry with
    [ "bhuu#1" → do
        { compute_perfect_bhuu entry (* No middle forms Whitney§800d *)
        ; record_part (Ppfta_ Primary (revcode "babhuu") entry)
        ; record_part_m_ath ppftm (revcode "babhuuv") entry
        }
    | "vid#1" → do
        { compute_perfect_vid entry (* middle forms ? *)
        ; record_part (Ppfta_ Primary (revcode "vid") entry)
        }
    | "ah" → compute_perfect_ah entry
    | "vyaa" → compute_perfect_vyaa entry (* does not fit standard aa scheme *)
    | "zvaa" → let (strong, weak,_,_,_) = redup_perf "zuu" in (* P{6,1,30} *)
                compute_perfect_v strong weak entry (* Whitney§794b zizvaaya *)
(* Whitney§794b also jyaa pyaa vyaa hvaa; we treat vyaa above, and hvaa is huu. Thus
pyaa is covered by pii. jyaa1 as jii gives jijyau same WR *)
    | "indh" → compute_perfectm Primary (revcode "iidh") entry
    | "mah" → let (strong, weak, _, _, _) = redup_perf entry in
               compute_perfectm Primary strong entry (* ZZ Atma for Para root *)
    | _ → let (strong, weak, olong, eweak, iopt) = redup_perf entry in
          match weak with
          [ [ c :: rest ] →
             if c = 2 (* aa *) ∨ (c > 9 ∧ c < 14) (* e ai o au *)
             then compute_perfect_aa rest entry (* shortened weak stem *)
             else if c > 2 ∧ c < 7 (* i ii u uu *)
                  then compute_perfect_v strong weak entry
             else if c = 7 (* .r *) ∧ multi_consonant entry ∨ c = 8 (* .rr *)
                  then compute_perfect_ril strong entry
             else if c = 7 (* .r *) then compute_perfect_v strong weak entry
             else compute_perfect_c strong weak olong eweak iopt entry
          | [] → error_empty 16
          ]
    ]
;
value compute_perfect_desida st entry =
(* entry :string is the root, st is the desiderative (reverse word) stem. *)
(* We create a fake root from st to reuse redup_perf which uses a string.*)
  let (strong, weak, olong, eweak, iopt) = redup_perf (Canon.rdecode st) in
  compute_perfecta Desiderative strong weak olong eweak iopt entry
```

```
and compute_perfect_desidm st entry =
  let (_, weak, _, _, _) = redup_perf (Canon.rdecode st) in
  compute_perfectm Desiderative weak entry
;
(****************************)
(* Periphrastic perfect li.t *)
(****************************)
(* Construction of the periphrastic perfect, used for perfect of secondary conjugations, de-
nominative verbs and a few roots. It builds a form in -aam suffixed by a perfect form of the
auxiliairies k.r bhuu et as P{3,1,35-40} *)
value peri_perf_stem entry =
  let stem = match entry with
  [ "iik.s" | "ii.d" | "iir" | "iih" | "uk.s" | "uc" | "ujjh" | "uuh" | "edh"
    (* Macdonell§140a1 Whitney§1071c Filliozat§66 edhaa.mcakre *)
  | "ind" | "indh" | "inv" | "ii.s" | "umbh" | "cakaas" → entry
  | "aas#2" → "aas" (* trim *)
  | "u.s" → "o.s" (* guna WR *)
  | "jaag.r" → "jaagar" (* Macdonell§140a2 *)
  | "bh.r" → "bibhar"
  | "nii#1" → "nay"
  | "i" → "ay" (* Whitney roots *)
  | "vyaa" → "vye" (* Whitney roots *)
  | "huu" → "hve" (* Macdonell§140a3 *)
  | "hrii#1" → "jihre" (* Whitney roots *)
  | _ → raise Not_attested (* no known periphrastic perfect *)
  ] in revcode stem
;
value build_perpft c abstem root =
  enter1 root (Invar (c, Perpft) (fix abstem "aam"))
;
```

Aorist system
augment True for aorist, False for injunctive

```
value sigma augment stem suff =
  let sfx = code suff in
  let ssfx = match sfx with
    [ [ 32 (* t *) :: _ ]
    | [ 33 (* th *) :: _ ] → match stem with
      [ [ c :: _ ] →
          if vowel c ∨ nasal c ∨ c = 43 (* r *) then [ 48 (* s *) :: sfx ]
```

```
          else sfx
        | _ → error_empty 17
        ]
      | [ c :: _ ] → [ 48 (* s *) :: sfx ]
      | [ ] → [ ]
      ] in
  let form = sandhi stem ssfx in
  if augment then aug form else form
;
value sigma_paradigm conjug =
   [ (Singular,
          [ conjug First "am"
          ; conjug Second "iis"
          ; conjug Third "iit"
          ])
   ; (Dual,
          [ conjug First "va"
          ; conjug Second "tam"
          ; conjug Third "taam"
          ])
   ; (Plural,
          [ conjug First "ma"
          ; conjug Second "ta"
          ; conjug Third "ur"
          ])
   ]
;
value compute_ath_s_aorista long entry =
  let conjug person suff = (person, sigma True long suff) in
  enter1 entry (Conju (aora 4) (sigma_paradigm conjug))
;
value compute_ath_s_injuncta long entry =
  let conjug person suff = (person, sigma False long suff) in
  enter1 entry (Conju (inja 4) (sigma_paradigm conjug))
;
value compute_ath_s_aoristm stem entry =
  let conjug person suff = (person, sigma True stem suff)
  and conjugroot person suff = (person, fix_augment stem suff)
  and conjugdhvam person =
        let suff = match stem with
```

```
          [ [ 1 (* a *) :: _ ] | [ 2 (* aa *) :: _ ] → "dhvam"
          | [ 43 (* r *) :: _ ] → ".dhvam"
          | [ c :: _ ] → if vowel c then ".dhvam" else "dhvam"
          | _ → error_empty 18
          ] in
      (person, fix_augment stem suff) in
  let conjugc = if entry = "k.r#1" (* Whitney§882a *)
                ∨ entry = "daa#1" (* Whitney§884 *) then conjugroot
                else match stem with
                     [ [ 43 :: _ ] | [ 36 :: _ ] | [ 41 :: _ ] → conjug
                       (* r n m Whitney§881*)
                     | [ c :: _ ] when consonant c → conjugroot
                 (* | [ c :: _ ] when short_vowel c → conjugroot ? *)
                     | _ → conjug
                     ] in
  enter1 entry (Conju (aorm 4)
   [ (Singular,
        [ conjug First "i"
        ; conjugc Second "thaas"
        ; conjugc Third "ta"
        ])
   ; (Dual,
        [ conjug First "vahi"
        ; conjug Second "aathaam"
        ; conjug Third "aataam"
        ])
   ; (Plural,
        [ conjug First "mahi"
        ; conjugdhvam Second
        ; conjug Third "ata"
        ])
   ])
;
value compute_ath_s_injunctm stem entry =
  let conjug person suff = (person, sigma False stem suff)
  and conjugroot person suff = (person, fix stem suff)
  and conjugdhvam person =
      let suff = match stem with
          [ [ 1 (* a *) :: _ ] | [ 2 (* aa *) :: _ ] → "dhvam"
          | [ 43 (* r *) :: _ ] → ".dhvam"
```

```
          | [ c :: _ ] → if vowel c then ".dhvam" else "dhvam"
          | _ → error_empty 19
          ] in
      (person, fix stem suff) in
  let conjugc = if entry = "k.r#1" then conjugroot else conjug in
  enter1 entry (Conju (injm 4)
   [ (Singular,
         [ conjug First "i"
         ; conjugc Second "thaas"
         ; conjugc Third "ta"
         ])
   ; (Dual,
         [ conjug First "vahi"
         ; conjug Second "aathaam"
         ; conjug Third "aataam"
         ])
   ; (Plural,
         [ conjug First "mahi"
         ; conjugdhvam Second
         ; conjug Third "ata"
         ])
   ])
;
value isigma augm stem suff long_i =
  let sfx = code suff in
  let sfx' = match sfx with
    [ [ 4 (* ii *) :: _ ] → sfx
    | _ → let ivoy = if long_i then 4 (* ii *) else 3 (* i *) in
            (* long i for root grah - Whitney§900b *)
            Int_sandhi.int_sandhi [ 47; ivoy ] (* i.s *) sfx
    ] in
  let form = sandhi stem sfx' in
  if augm then aug form else form
;
value compute_ath_is_aorista stem entry =
  let long_i = (entry = "grah") in
  let conjug person suff = (person, isigma True stem suff long_i) in
  enter1 entry (Conju (aora 5) (sigma_paradigm conjug))
;
value compute_ath_is_injuncta stem entry =
```

```
  let long_i = (entry = "grah") in
  let conjug person suff = (person, isigma False stem suff long_i) in
  enter1 entry (Conju (inja 5) (sigma_paradigm conjug))
;
value isigma_m_paradigm conjug conjugdhvam =
   [ (Singular,
        [ conjug First "i"
        ; conjug Second "thaas"
        ; conjug Third "ta"
        ])
   ; (Dual,
        [ conjug First "vahi"
        ; conjug Second "aathaam"
        ; conjug Third "aataam"
        ])
   ; (Plural,
        [ conjug First "mahi"
        ; conjugdhvam Second
        ; conjug Third "ata"
        ])
   ]
;
value compute_ath_is_aoristm stem entry =
  let long_i = (entry = "grah") in
  let conjug person suff = (person, isigma True stem suff long_i)
  and conjugdhvam person = (person, fix_augment stem suff)
      where suff = (if long_i then "ii" else "i") ^ "dhvam" in
  enter1 entry (Conju (aorm 5) (isigma_m_paradigm conjug conjugdhvam))
;
value compute_ath_is_injunctm stem entry =
  let long_i = (entry = "grah") in
  let conjug person suff = (person, isigma False stem suff long_i)
  and conjugdhvam person = (person, fix stem suff)
      where suff = (if long_i then "ii" else "i") ^ "dhvam" in
  enter1 entry (Conju (injm 5) (isigma_m_paradigm conjug conjugdhvam))
;
value sisigma augm stem suff =
  let sfx = code suff in
  let ssfx = match sfx with
    [ [ 4 :: _ ] -> [ 48 (* s *) :: sfx ]
```

```
    | _ → Int_sandhi.int_sandhi [ 47; 3; 48 ] (* si.s *) sfx
    ] in
  let form = sandhi stem ssfx in
  if augm then aug form else form
;
value compute_ath_sis_aorista stem entry =
  let conjug person suff = (person, sisigma True stem suff) in
  enter1 entry (Conju (aora 6) (sigma_paradigm conjug))
;
value compute_ath_sis_injuncta stem entry =
  let conjug person suff = (person, sisigma False stem suff) in
  enter1 entry (Conju (inja 6) (sigma_paradigm conjug))
;
value sasigma augm stem suff =
  let sfx = fix [ 48 ] (* s *) suff in
  let form = sandhi stem sfx in
  if augm then aug form else form
;
value sa_aorist_a conjug =
   [ (Singular,
         [ conjug First "am"
         ; conjug Second "as"
         ; conjug Third "at" (* secondary (shorter) ending Whitney§542 *)
         ])
   ; (Dual,
         [ conjug First "aava"
         ; conjug Second "atam"
         ; conjug Third "ataam"
         ])
   ; (Plural,
         [ conjug First "aama"
         ; conjug Second "ata"
         ; conjug Third "an"
         ])
   ]
;
value compute_ath_sa_aorista stem entry =
  let conjug person suff = (person, sasigma True stem suff) in
  enter1 entry (Conju (aora 7) (sa_aorist_a conjug))
;
```

```
value compute_ath_sa_injuncta stem entry =
  let conjug person suff = (person, sasigma False stem suff) in
  enter1 entry (Conju (inja 7) (sa_aorist_a conjug))
;
value sa_aorist_m conjug =
   [ (Singular,
         [ conjug First "i"
         ; conjug Second "athaas"
         ; conjug Third "ata"
         ])
   ; (Dual,
         [ conjug First "aavahi"
         ; conjug Second "aathaam"
         ; conjug Third "aataam"
         ])
   ; (Plural,
         [ conjug First "aamahi"
         ; conjug Second "adhvam"
         ; conjug Third "anta"
         ])
   ]
;
value compute_ath_sa_aoristm stem entry =
  let conjug person suff = (person, sasigma True stem suff) in
  enter1 entry (Conju (aorm 7) (sa_aorist_m conjug))
;
value compute_ath_sa_injunctm stem entry =
  let conjug person suff = (person, sasigma False stem suff) in
  enter1 entry (Conju (injm 7) (sa_aorist_m conjug))
;
value compute_root_aorista weak strong entry =
  let conjugw person suff = (person, fix_augment weak suff)
  and conjugs person suff = (person, fix_augment strong suff) in
  enter1 entry (Conju (aora 1)
   [ (Singular, if entry = "bhuu#1" then (* Whitney§830 *)
         [ (First, code "abhuuvam") (* RV abhuvam *)
         ; conjugw Second "s"
         ; conjugw Third "t"
         ] else (* Whitney§831 *)
         [ conjugs First "am"
```

```
          ; conjugs Second "s"
          ; conjugs Third "t"
          ])
    ; (Dual,
          [ conjugw First "va"
          ; conjugw Second "tam"
          ; conjugw Third "taam"
          ])
    ; (Plural,
          [ conjugw First "ma"
          ; conjugw Second "ta"
          ; (Third, match weak with
               [ [ 2 (* aa *) :: r ]
                    → fix_augment r "ur"
               | [ 41; 1; 43; 17 ] (* kram *) (* Whitney§833a *)
                    → fix_augment weak "ur" (* also yam dabh n.rt mand *)
               | [ 6; 40 ] (* bhuu *) → code "abhuuvan"
               | [ 41; 1; 19 ] (* gam *) → code "agman"
               | _ → fix_augment weak "an"
               ])
          ])
    ])
;
value compute_root_injuncta weak strong entry =
  let conjugw person suff = (person, fix weak suff)
  and conjugs person suff = (person, fix strong suff) in
  enter1 entry (Conju (inja 1)
   [ (Singular, if entry = "bhuu#1" then
          [ (First, code "bhuuvam")
          ; conjugw Second "s"
          ; conjugw Third "t"
          ] else
          [ conjugs First "am"
          ; conjugs Second "s"
          ; conjugs Third "t"
          ])
    ; (Dual,
          [ conjugw First "va"
          ; conjugw Second "tam"
          ; conjugw Third "taam"
```

```
        ])
  ; (Plural,
        [ conjugw First "ma"
        ; conjugw Second "ta"
        ; (Third, match weak with
             [ [ 2 (* aa *) :: r ] → fix r "ur"
             | [ 6; 40 ] (* bhuu *) → code "bhuuvan"
             | [ 41; 1; 19 ] (* gam *) → code "gman"
             | _ → fix weak "an"
             ])
        ])
  ])
;
value compute_root_aoristm stem entry = (* rare *)
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (aorm 1) (conjugs_past_m conjug))
;
value compute_root_injunctm stem entry = (* rare *)
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju (injm 1) (conjugs_past_m conjug))
;
value compute_root_aoristp stem entry = (* passive aorist Whitney§843 *)
  (* P{3,1,60-66} suffix ci.n usage réflexif-passif agent/objet karmakart.r *)
  (* TODO use Kümmel 1996 for Vedic plural 3rd forms *)
  let conjug person suff = (person, fix_augment stem suff) in
  let conju3 = Conju aorp1 [ (Singular, [ conjug Third "i" ]) ] in
  enter1 entry conju3
;
value compute_root_injunctp stem entry = (* passive injunctive ? *)
  let conjug person suff = (person, fix stem suff) in
  let conju3 = Conju injp1 [ (Singular, [ conjug Third "i" ]) ] in
  enter1 entry conju3
;
(* identical to compute_thematic_impfta *)
value compute_thematic_aorista stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (aora 2) (thematic_preterit_a conjug))
;
value compute_thematic_injuncta stem entry =
  let conjug person suff = (person, fix stem suff) in
```

```
  enter1 entry (Conju (inja 2) (thematic_preterit_a conjug))
;
(* identical to compute_thematic_impftm *)
value compute_thematic_aoristm stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (aorm 2) (thematic_preterit_m conjug))
;
value compute_thematic_injunctm stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju (injm 2) (thematic_preterit_m conjug))
;
(* identical to compute_thematic_impfta *)
(* de Saussure (Memoire sur le systeme primitif des voyelles dans les langues IE) says:
reduplicated aorists represent imperfects of a verbal class. *)
value compute_redup_aorista stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (aora 3) (thematic_preterit_a conjug))
  (* NB Macdonnel dixit – Gonda says "ur" for Third Plural *)
;
value compute_redup_injuncta stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju (inja 3) (thematic_preterit_a conjug))
;
(* identical to compute_thematic_impftm *)
value compute_redup_aoristm stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (aorm 3) (thematic_preterit_m conjug))
;
value compute_redup_injunctm stem entry =
  let conjug person suff = (person, fix stem suff) in
  enter1 entry (Conju (injm 3) (thematic_preterit_m conjug))
;
value amui = fun (* root with a amui - used in redup_aor *)
  [ "kath" → True (* P{7,4,93} *)
  | _ → False
  ]
;
(* Reduplication for aorist/injunctive *)
value redup_aor weak root =
  let mess = "Redup_aor␣" ^ root in
```

```
match rev weak with (* ugly double reversal *)
  [ [] → error_empty 20
  | [ c1 :: r ] →
    if vowel c1 then match c1 with (* very rare - Whitney§862 *)
       [ 1 (* a *) → match r with
           [ [ c2 ] → weak @ [ c2; 1 (* a *)] (* am aorist aamamat *)
           | _ → failwith mess
           ]
       | 4 (* ii *) → match r with
           [ [ 17; 47 ] (* iik.s *) → revcode "iicik.s"
           | _ → failwith mess
           ]
       | 7 (* .r *) → match r with
           [ [ 22 ] (* .rc1 *) → revcode ".rcic"
           | _ → failwith mess
           ]
       | _ → failwith mess
       ]
    else
    let (v, heavy) = lookvoy r
         (* heavy syllable = long vowel, or short before two consonants (long by position)
*)
       where rec lookvoy = fun
         [ [] → failwith mess
         | [ c2 ] → if vowel c2 then (c2, ¬ (short_vowel c2))
                    else failwith mess
         | [ c2 :: r2 ] → if vowel c2 then
                              let h = if short_vowel c2 then mult r2
                                      else True in
                              (c2, h)
                          else lookvoy r2
         ]
    and c = if sibilant c1 then match r with
         [ [] → failwith mess
         | [ c2 :: _ ] → if vowel c2 then c1
                         else if nasal c2 then c1
                         else if stop c2 then c2
                         else (* semivowel c2 *) c1
         ] else c1 in
    let rv = (* rv is reduplicating vowel *)
```

```
        if v = 5 then match root with
          [ "dru#1" | "zru" | "stu" | "sru" → 5
          | "dyut#1" → 3 (* also "zru" azizravat (WR) *)
          | _ → 6 (* u -¿ uu *)
          ]
        else if v = 6 then 5 (* uu →u *)
        else match root with
            [ "klid" | "tvar" | "tvi.s#1" | "zri" | "grah" | "vrazc" → 3
            | "j~naa#1" | "sthaa#1" | "hlaad" (* hidden heavy since stem in i *)
                → 3
            | "gaah" (* heavy exception *) → 4
            | _ → if heavy ∨ amui root then
                        if v = 1 ∨ v = 2 ∨ v = 7 then 1 (* Whitney§860 *)
                        else 3 (* short →ii, long →i *) (* P{7,4,93} *)
                    else 4
            ]
      and rc = match c with (* c is reduplicating consonant *)
        [ 17 | 18 (* k kh *) → 22 (* c *)
        | 19 | 20 | 49 (* g gh h *) → 24 (* j *)
        | 23 | 25 | 28 | 30 | 33 | 35 | 38 | 40 → c − 1 (* xh →x *)
        | _ → c
        ]
      and strengthened = match root with
        [ "ji" → revcode "jay"
        | _ → match weak with
              [ [ c :: r ] →
                if vowel c then match c with
                                [ 3 | 4 (* i ii *) → [ 42 (* y *) :: weak ]
                                | 5 | 6 (* u uu *) → [ 45 (* v *) :: weak ]
          (* or 45 :: [ 1 :: r ] (stu) 'atu.s.tavam tu.s.t'avat RV (WR) *)
                                | 7 | 8 (* .r .rr *) → [ 43 :: [ 1 (* ar *) :: r ] ]
                                | _ → weak (* Whitney§866-868 *)
                                ]
                else weak
              | _ → error_empty 21
              ]
        ] in
      revaffix True [ rv; rc ] strengthened
    ]
;
```

```
value compute_aorist entry =
  let (weak, strong, long) = stems entry in do (* 7 formations *)
  { match entry with (* 1. root aorist - Panini sic-luk *)
    [ "k.r#1" | "kram" | "gam" | "gaa#1" | "jan" | "j~naa#1"
    | "daa#1" | "daa#2" | "dhaa#1" | "dhaa#2" | "paa#1" | "bhuu#1" | "muc#1"
    | "zaa" | "saa#1" | "sthaa#1" | "has" | "haa#1" → do
      { compute_root_aorista weak strong entry
      ; match entry with
        [ "k.r#1" | "gam" | "jan" → compute_root_aoristm weak entry (* rare *)
        | "sthaa#1" (* Whitney§834a. *) →
                     compute_root_aoristm (revstem "sthi") entry (* asthita *)
        | "dhaa#1" → compute_root_aoristm (revstem "dhi") entry
        | _ → ()
        ]
      ; let stem = if entry = "muc#1" then strong else match long with
            [ [ 2 (* aa *) :: _ ] → [ 42 (* y *) :: long ]
            | _ → long
            ] in
        compute_root_aoristp stem entry (* passive *)
      (* For root aorist participles, see Whitney§840 and Burrow p178 *)
      (* For optative mode Whitney§837 see benedictive/precative. *)
      }
    | "prii" → let st = revcode "priiyaa" in compute_root_aorista st st entry
    | "svid#2" → let st = revcode "svidyaa" in compute_root_aorista st st entry
    | "iik.s" | "m.r" → compute_root_aoristm weak entry
    (* Now other passive/impersonal aorist in -i *)
    | "vac" → do (* passive aorist *)
      { compute_root_aoristp long entry
      ; compute_root_aoristp (revcode "voc") entry
      }
    | "p.rr" → compute_root_aoristp (revcode "puur") entry
    | "kaaz" | "k.sip" | "diip" | "duh#1" | "d.rz#1" | "dvi.s#1" | "budh#1"
    | "yuj#1" | "vid#1" | "s.rj#1"
        → compute_root_aoristp strong entry
    | "rabh" → compute_root_aoristp (revcode "rambh") entry
    | "ci" | "jaag.r" | "t.rr" | "pac" | "pad#1" | "zru" | "stu" | "hu"
        → compute_root_aoristp long entry
           (* NB "zru" -¿ azraavi WR while Whitney§844a *azraayi typo *)
    | _ → () (* "i" -¿ iiyaat hard *)
    ]
```

```
; match entry with (* 2. thematic aorist af *)
  [ "aap" | "krudh" | "gam" | "g.rdh" | "ghas" | "das" | "dyut#1" | "muc#1"
  | "yuj#1" | "ric" | "ruc#1" | "rudh#2" | "ruh" | "vid#2" | "v.rt#1"
  | "zuc#1" | "zudh" | "sic" | "stan" | "huu"
    → do
      { compute_thematic_aorista weak entry
      ; compute_thematic_aoristm weak entry (* middle very rare *)
      }
  | "vac" → let stem = revcode "voc" in do
      { compute_thematic_aorista stem entry
      ; compute_thematic_aoristm stem entry
      }
  | "vyaa" → let stem = revcode "vi" in do
      { compute_thematic_aorista stem entry
      ; compute_thematic_aoristm stem entry
      }
  | "zak" | "zuu" | "zcut#1" | "zram" → compute_thematic_aorista weak entry
  | "zru" → compute_thematic_aorista (revcode "zrav") entry
  | "khyaa" → compute_thematic_aorista (revcode "khy") entry
  | "as#2" → compute_thematic_aorista (revcode "asth") entry
  | "pat#1" → compute_thematic_aorista (revcode "papt") entry
  | (* roots in .r or .rr take strong stem *)
    ".r" | "d.rz#1" → compute_thematic_aorista strong entry
  | _ → ()
  ]
; match entry with (* 3. reduplicated aorist caf *)
  [ "am" | ".rc#1" | "kath" | "k.r#1" | "k.r.s" | "k.lp" | "ga.n" | "gam"
  | "gaah" | "car" | "ce.s.t" | "jan" | "ji" | "tvar" | "tvi.s#1" | "dah#1"
  | "diz#1" | "dih" | "diip" | "dru#1" | "dh.r" | "naz" | "pac" | "pa.th"
  | "miil" | "muc#1" | "yaj#1" | "rak.s" | "ric" | "viz#1" | "v.r#1"
  | "v.rt#1" | "vyadh" | "zri" | "zru" | "stu" (* — "dhaa#1" *) →
    let stem = redup_aor weak entry in do
    { compute_redup_aorista stem entry (* but atu.s.tavam RV (WR) *)
    ; compute_redup_aoristm stem entry
    }
  | "iik.s" | "kamp" | "klid" | "gup" | "cur" | "m.r" | "d.rz#1" | "dyut#1"
  | "vrazc" | "siiv" | "sru" → (* active only *)
    let stem = redup_aor weak entry in
    compute_redup_aorista stem entry
  | "grah" → do
```

```
    { let stem = redup_aor (revcode "grah") entry in do
      { compute_redup_aorista stem entry
      ; compute_redup_aoristm stem entry
      }
    ; let stem = redup_aor (revcode "grabh") entry in do (* ved Whitney§223g *)
      { compute_redup_aorista stem entry
      ; compute_redup_aoristm stem entry
      }
    }
  | "daa#1" → let stem = (revcode "diidad") (* ad hoc *) in do
      { compute_redup_aorista stem entry
      ; compute_redup_aoristm stem entry
      }
    (* then exceptions to treatment of aa with intercalaring ii *)
  | "raadh" → let stem = redup_aor (revcode "radh") (* riiradh *) entry in
              compute_redup_aorista stem entry (* Macdonnel p 126 *)
  | "haa#1" → let stem = revcode "jiijah" in
              compute_redup_aorista stem entry
  | _ → ()
  ]
; match entry with (* reduplicated aorist - extra forms, secondary conjs *)
  [ "naz" → compute_redup_aorista (revcode "nez") entry
  | _ → ()
  ]
; match entry with (* 4. sigma aorist sic *)
  [ "aap" | "k.r#1" | "khan" | "gup" | "chid#1" | "ji" | "tud" | "t.rr"
  | "tyaj#1" | "dah#1" | "daa#1" | "d.rz#1" | "draa#2" | "dhaa#1" | "dhyaa"
  | "dhyai" | "dhv.r" | "nak.s" | "nii#1" | "pac" | "praz" | "prii"
  | "budh#1" | "bhaa#1" | "bhii#1" | "muc#1" | "yaj#1" | "yuj#1" | "ram"
  | "labh" | "v.r#2" | "vyadh" | "zru" | "sidh#1" | "s.rj#1" | "stu"
  | "sp.rz#1" | "hu" → do
    { let stem = match entry with
            [ "d.rz#1" | "s.rj#1" | "sp.rz#1" → long_metathesis weak
            | "ram" → weak
            | _ → long
            ] in
      compute_ath_s_aorista stem entry
    ; match entry with (* Whitney§890 *)
            [ "khan" (* akhaan *)
            | "dah#1" (* adhaak *)
```

```
          (* — "d.rz1" adraak wrong *adaar.t below TODO use ar_ra *)
          | "yaj#1" (* ayaa.t *)
          (* — "s.rj1" asraak wrong *asaar.t below *)
            → let lopa = sigma True long "" in
              enter1 entry (Conju (aora 4) [ (Singular, [ (Third, lopa) ]) ])
          | _ → ()
          ]
    ; if entry = "yuj#1" ∨ entry = "chid#1"
        then compute_ath_s_aorista strong entry else ()
      (* ayok.siit and acchetsiit besides ayauk.siit and acchaitsiit *)
    ; match entry with
      [ "gup" | "d.rz#1" | "s.rj#1" → () (* active only *)
      | _ → let stemm = match weak with
          [ [ c :: r ] → match c with
              [ 3 | 4 | 5 | 6 (* i ii u uu *) → strong
              | 2 (* aa *) → [ 3 :: r ]
              | 7 (* .r *) → if entry = "dhv.r" then revcode "dhuur" else weak
              | _ → weak
              ]
          | _ → error_empty 22
          ] in compute_ath_s_aoristm stemm entry
      ]
    }
  | "k.r.s" → do
      { let stem = long in compute_ath_s_aorista stem entry (* akaark.siit *)
      ; let stem = ar_ra long in
        compute_ath_s_aorista stem entry (* metathesis akraak.siit *)
      }
  | "vrazc" → let stem = revcode "vraak" in (* as for future *)
              compute_ath_s_aorista stem entry
  | "spaz#1" | "smi" | "haa#2" → compute_ath_s_aoristm weak entry (* middle only
*)
  | _ → ()
  ]
; match entry with (* 5. i.s aorist se.t-sic *)
  [ "ak.s" | "aj" | "aas#2" | "i.s#1" | "iik.s" | "uk.s" | "uc" | "u.s"
  | "uuh" | ".rc#1" | "k.rt#1" | "krand" | "kram" | "k.san" | "khan"
  | "car" | "ce.s.t" | "jap" | "jalp" | "jaag.r" | "t.rr" | "diip"
  | "pa.th" | "puu#1" | "p.rc"| "pru.s#1" | "baadh" | "budh#1" | "mad#1"
  | "mud#1" | "muurch" | "mlecch" | "yaac" | "ruc#1" | "lu~nc" | "luu#1"
```

```
| "vad" | "vaz" | "vadh" | "vid#1" | "v.r#1" | "vraj" | "z.rr" | "sidh#2"
| "skhal" | "stan" | "stu" | "hi.ms" → do
  { let stem = match weak with
        [ [ 7 (* .r *) :: _ ] → (* complex Paninian see Müller Gram xii *)
          if entry = "jaag.r" then strong (* jaagari.sam RF IC 2 p 88 *)
          else long (* avaariit *)
        | [ 8 (* .rr *) :: _ ] →
          if entry = "z.rr" then strong (* azariit *)
          else long
        | [ c :: _ ] →
          if vowel c then long
          else match entry with
               [ "kan" | "khan" | "car" | "mad#1" | "vad" | "skhal" → long
               | _ → strong
               ]
        | [] → error_empty 23
        ] in
    compute_ath_is_aorista stem entry
  ; compute_ath_is_aoristm strong entry
  }
| "ku.s" | "gup" | "vrazc" | "zcut#1" | "sphu.t" → (* active only *)
  compute_ath_is_aorista strong entry
| "zuu" →
  compute_ath_is_aorista (revcode "zve") entry
| "kan" | "k.r#2"| "p.rr" | "zaz" → (* active only *)
  compute_ath_is_aorista long entry
| "kamp" | "jan" | "zii#1" | "spand" → (* middle only *)
  compute_ath_is_aoristm strong entry
| "grah" → do
  { let stem = revcode "grah" in do (* same as group above *)
    { compute_ath_is_aorista stem entry
    ; compute_ath_is_aoristm stem entry
    }
  ; let stem = revcode "grabh" in do (* supplement (ved) – Whitney§900b *)
    { compute_ath_is_aorista stem entry
    ; compute_ath_is_aoristm stem entry
    }
  }
| _ → ()
]
```

```
  ; match entry with (* 6. si.s aorist se.t-sic *)
    [ "j~naa#1" | "dhyaa" | "dhyai" | "nam" | "paa#2" | "mnaa" | "yaa#1" | "laa"
    | "zaa" → do (* dhyai for dhyaa *)
      { compute_ath_sis_aorista strong entry
      ; compute_ath_is_aoristm strong entry (* is aorist (5) used in middle *)
      }
    | _ → ()
    ]
; match entry with (* 7. sa aorist ksa *)
      [ "k.r.s" | "kruz" | "kliz" | "guh" | "diz#1" | "dih" | "duh#1" | "lih#1"
      | "viz#1" | "v.rj" | "sp.rz#1" → do (* P{7,3,72-73} *)
      { compute_ath_sa_aorista weak entry
      ; if entry = "kruz" ∨ entry = "kliz" then ((* Para *))
        else compute_ath_sa_aoristm weak entry
      }
    | "pac" → do (* Kiparsky apaak.sam *)
      { compute_ath_sa_aorista long entry
      ; compute_ath_sa_aoristm long entry
      }
    | _ → ()
    ]
  }
;
(* First approximation: we compute same forms as corresponding aorists. *)
(* Then restriction to attested usage *)
value compute_injunctive entry =
  let (weak, strong, long) = stems entry in do (* 7 families *)
  { match entry with (* 1. root injunct *)
    [ "gam" | "gaa#1" | "bhuu#1" → do
      { compute_root_injuncta weak strong entry
      ; if entry = "gam" then compute_root_injunctm weak entry (* rare *) else ()
      ; let stem = match long with
            [ [ 2 (* aa *) :: _ ] → [ 42 (* y *) :: long ]
            | _ → long
            ] in
        compute_root_injunctp stem entry (* passive *)
      }
    | "k.r#1" → compute_root_injunctm weak entry
    | _ → ()
    ]
```

```
; match entry with (* 2. thematic injunct *)
  [ "gam" | "g.rdh" | "zuc#1" → do
    { compute_thematic_injuncta weak entry
    ; compute_thematic_injunctm weak entry (* middle is very rare *)
    }
  | "zram" → compute_thematic_injuncta weak entry (* zramat *)
  | "vac" → let weak = revcode "voc" in do
            { compute_thematic_injuncta weak entry (* vocat *)
            ; compute_thematic_injunctm weak entry (* vocantq *)
            }
  | "zru" → compute_thematic_injuncta (revcode "zrav") entry (* zravat *)
  | _ → ()
  ]
; match entry with (* 3. reduplicated injunct *)
  [ "k.r#1" | "gam" →
    let stem = redup_aor weak entry in do
    { compute_redup_injuncta stem entry
    ; compute_redup_injunctm stem entry
    }
  | _ → ()
  ]
; match entry with (* 4. sigma injunct *)
  [ "k.r#1" | "chid#1" | "tyaj#1" | "pac" | "praz" | "bhii#1" | "sidh#1" → do
    { let stema = long in
      compute_ath_s_injuncta stema entry
    ; if entry = "chid#1" then compute_ath_s_injuncta strong entry else ()
      (* cchetsiit besides cchaitsiit *)
    ; let stemm = match weak with
           [ [ c :: r ] → match c with
              [ 3 | 4 | 5 | 6 (* i ii u uu *) → strong
              | 2 (* aa *) → [ 3 :: r ] (* turn aa to i *)
              | _ → weak
              ]
           | _ → error_empty 24
           ] in
      compute_ath_s_injunctm stemm entry
    }
  | _ → ()
  ]
; match entry with (* 5. i.s injunct *)
```

```
[ "ak.s" | "aj" | "aas#2" | "i.s#1" | "iik.s" | "uk.s" | "uc" | "u.s"
| "uuh" | ".rc#1" | "k.rt#1" | "krand" | "kram" | "k.san" | "khan" | "car"
| "ce.s.t" | "jalp" | "jaag.r" | "t.rr" | "diip" | "pa.th"
| "puu#1" | "p.rc" | "baadh" | "budh#1" | "mad#1" | "mud#1" | "muurch"
| "mlecch" | "yaac" | "ruc#1" | "lu~nc" | "luu#1" | "vad" | "vadh" | "vaz"
| "vid#1" | "v.r#1" | "vraj" | "z.rr" | "sidh#2" | "skhal" | "stan"
| "stu" | "hi.ms" → do
  { let stem = match weak with
        [ [ 7 (* .r *) :: _ ] →
          if entry = "jaag.r" then strong (* jaagari.sam RF IC 2 p 88 *)
          else long (* avaariit *)
        | [ 8 (* .rr *) :: _ ] →
          if entry = "z.rr" then strong (* azariit *)
          else long
        | [ c :: _ ] →
          if vowel c then long
          else match entry with
               [ "kan" | "khan" | "car" | "mad#1" | "vad" | "skhal" → long
               | _ → strong
               ]
        | [] → error_empty 25
        ] in
    compute_ath_is_injuncta stem entry
  ; compute_ath_is_injunctm strong entry
  }
| "gup" | "vrazc" | "zcut#1" | "sphu.t" → (* active only *)
  compute_ath_is_injuncta strong entry
| "zuu" →
  compute_ath_is_injuncta (revcode "zve") entry
| "kan" | "k.r#2"| "p.rr" → (* active only *)
  compute_ath_is_injuncta long entry
| "kamp" | "jan" | "zii#1" | "spand" → (* middle only *)
  compute_ath_is_injunctm strong entry
| "grah" → do
  { let stem = revcode "grah" in do (* same as group above *)
    { compute_ath_is_injuncta stem entry
    ; compute_ath_is_injunctm stem entry
    }
  ; let stem = revcode "grabh" in do (* supplement (ved) – Whitney§900b *)
    { compute_ath_is_injuncta stem entry
```

```
          ; compute_ath_is_injunctm stem entry
          }
        }
     | _ → ()
     ]
  } (* injunctives of kinds 6. and 7. missing *)
 ;
(* Aorist of causative *)
value compute_redup_aorista_ca stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (caaora 3) (thematic_preterit_a conjug))
  (* NB Macdonnel dixit – Gonda says "ur" for Third Plural *)
;
value compute_redup_aoristm_ca stem entry =
  let conjug person suff = (person, fix_augment stem suff) in
  enter1 entry (Conju (caaorm 3) (thematic_preterit_m conjug))
;
value compute_aor_ca cpstem entry =
  match entry with
  [ (* Whitney§861b *) "j~naa#1" | "daa#1" | "sthaa#1"
    (* Henry§339: *)
  | "diip" (* adidiipat *)
  | "du.s" (* aduudu.sat *)
  | "ri.s" (* ariiri.sat *)
  | "p.r#1" (* apiiparat *)
  | "t.rr" (* atiitarat *)
  | "vah#1" (* aviivahat *)
  | "hlaad" (* ajihladat *)
(*— "jan" (* wrong *ajijiinat for ajiijanat *) — "sp.rz#1" (* wrong *apii.spazat for apisp.rzat *) TODO *) →
      match cpstem with (* cpstem-ayati is the ca stem *)
     [ [ 37 :: [ 2 :: w ] ] → (* w-aapayati *)
          let voy = if entry = "daa#1" then 1 (* a *)
                    else 3 (* i *) (* aap -¿ ip Whitney§861b *) in
          let istem = [ 37 :: [ voy :: w ] ] in
          let stem = redup_aor istem entry in do
          { compute_redup_aorista_ca stem entry (* ati.s.thipat adiidapat *)
          ; compute_redup_aoristm_ca stem entry
          }
     | [ 37 :: [ 1 :: _ ] ] →
```

```
          let stem = redup_aor cpstem entry in do
          { compute_redup_aorista_ca stem entry (* ajij napat *)
          ; compute_redup_aoristm_ca stem entry
          }
     | [ c :: w ] →
          let (v, light, r) = look_rec True w
               where rec look_rec b = fun
               [ [ ] → error_empty 26
               | [ x :: w' ] → if vowel x then (x, b ∧ short_vowel x, w')
                                 else look_rec False w'
               ] in
          let voy = match v with
                 [ 5 (* u *) → 6
                 | 6 (* uu *) → 5
                 | 1 | 2 → if light then 4 (* ii *)
                           else 1 (* a *)
                 | _ → if light then 4 (* ii *)
                           else 3 (* i *)
                 ] in
          let istem = [ c :: [ voy :: r ] ] in
          let stem = redup_aor istem entry in do
          { compute_redup_aorista_ca stem entry (* adidiipat *)
          ; compute_redup_aoristm_ca stem entry
          }
     | _ → error_empty 27
     ]
  | _ → ()
  ]
;
```

Periphrastic future, Infinitive, Passive future participle in -tavya

```
value compute_peri_fut conj perstem entry =
  let conjug person suff = (person, sandhi perstem (code suff)) in
  enter1 entry (Conju (conj, Perfut Active)
   [ (Singular,
        [ conjug First "taasmi"
        ; conjug Second "taasi"
        ; conjug Third "taa"
        ] )
   ; (Dual,
```

```
        [ conjug First "taasvas"
        ; conjug Second "taasthas"
        ; conjug Third "taarau"
        ])
   ; (Plural,
        [ conjug First "taasmas"
        ; conjug Second "taastha"
        ; conjug Third "taaras"
        ])
   ])
;
value record_pfp_tavya conj perstem entry =
  let pfp_stem = fix perstem "tavya" in
  record_part (Pfutp_ conj (rev pfp_stem) entry) (* rev compat entry by Pfpart *)
;
value build_infinitive c inf_stem root = do
(* By default, for Causative, we get eg bhaavayitum, and later forms such as bhaavitum
have to be entered as supplements; see Witney§1051c. *)
  { enter1 root (Invar (c, Infi) (fix inf_stem "tum"))
  ; enter1 root (Inftu c (fix inf_stem "tu")) (* Xtu-kaama compounds *)
(* NB. bahuv cpds in -kaama and -manas constructed with infinitives in -tu See Renou HLS
p72 from Patanjali; Renou grammaire §107 dagdhukaama also Assimil p194 eg tyaktukaama
anu.s.thaatukaama "desirious␣to␣proceed" vaktukaama "who␣wants␣to␣speak" dra.s.tumanas
"inclined␣to␣see" dra.s.tuzakya "able␣to␣see" *)
  }
;
value perif conj perstem entry = do
  { match entry with
    [ "cint" → () (* no future *)
    | _ → compute_peri_fut conj perstem entry
    ]
  ; let inf_stem = match conj with
        [ Primary → (* Difference infinitive/tavya forms and peri-future *)
              match entry with (* should rather appear in perstems *)
              [ "g.rr#1" → revcode "giri" (* giritum, not gariitum *)
              | "jak.s" → revcode "jagh" (* jagdhum *)
              | "p.rr" → revcode "puuri" (* puuritum *)
              | "sva~nj" → revcode "svaj" (* svaktum *)
              | "sa~nj" → revcode "saj" (* saktum *)
              | "s.rp" → revcode "sarpi" (* sarpitum *)
```

```
                | ".dii" → revcode ".dii" (* .diitum *)
                | _ → perstem
                ]
            | _ → perstem
            ] in
        build_infinitive conj inf_stem entry (* pb saa1 setum WR -situm *)
    ; if admits_passive entry then record_pfp_tavya conj perstem entry else ()
    (* other pfps generated from pfp_ya et pfp_aniiya below *)
    }
;
(* Computes periphrastic future, infinitive and pfp_tavya Primary forms *)
value compute_perif rstem entry =
    let pstems = perstems rstem entry in
    iter (fun st → perif Primary (rev st) entry) pstems
;
```

Passive future participle in -ya and -aniiya in all conjugations

```
value palatal_exception root = List.mem root
    [ "aj"; "vraj" (* P{7,3,60} *)
    ; "zuc#1" (* P{7,3,59} zocya *)
    ; "yaj#1"; "yaac"; "ruc#1"; ".rc#1" (* P{7,3,66} *)
    ; "tyaj#1" (* tyaajya Vaartika on P{7,3,66} *)
    ; "s.rj#1"; "v.rj"; "p.rc" (* because of -kyap P{3,1,110} *)
    ; "raaj#1" (* in order not to get raagya - unjustified by Panini ? *)
    ]
;
value velarification rstem = (* P{7,3,52} *)
    match Word.mirror rstem with (* double rev *)
    [ [ c :: _ ] when velar c → rfix rstem "ya" (* P{7,3,59} *)
(* Actually the following velarification should be registered as an optional form, since
P{7,3,65} says that it does not apply in the sense of necessity *)
    | _ → let st = match rstem with (* Int_sandhi.restore_stem not needed *)
                [ [ 22 (* c *) :: [ 26 (* n *) :: r ] ] →
                  [ 17 (* k *) :: [ 21 (* f *) :: r ] ] (* vafkya *)
                | [ 22 (* c *) :: r ] → [ 17 (* k *) :: r ] (* paakya vaakya *)
                | [ 24 (* j *) :: [ 24 (* j *) :: r ] ] →
                  [ 19 (* g *) :: [ 19 (* g *) :: r ] ] (* bh.rggya *)
                | [ 24 (* j *) :: [ 26 (* n *) :: r ] ] →
                  [ 19 (* g *) :: [ 21 (* f *) :: r ] ] (* safgya *)
                | [ 24 (* j *) :: r ] → [ 19 (* g *) :: r ] (* maargya *)
```

```
            | _ → rstem
            ] in
        rfix st "ya"
  ]
;
value record_pfp_ya conj ya_stem root =
  let pfp_stem =
      if conj = Primary then
         if palatal_exception root then rfix ya_stem "ya"
         else match root with
               [ "hi.ms" → revcode "hi.msya" (* no retroflex s Whitney§183a *)
               | _ → velarification ya_stem (* .nyat *)
               ]
      else rfix ya_stem "ya" (* yat *) in
  record_part (Pfutp_ conj pfp_stem root)
 ;
value record_pfp_aniiya conj iya_stem root =
  let pfp_stem = rfix iya_stem "aniiya" in
  record_part (Pfutp_ conj pfp_stem root)
 ;
(* Primary conjugation pfp in -ya except for ganas 10 and 11 *)
value pfp_ya rstem entry =
  let (_, strong, long) = stems entry in
  (* NB we do not use weak_stem and thus rstem is not mrijified/duhified *)
  let ya_stem = match rstem with
    [ [ 1 :: _ ] → rstem
    | [ 2 :: r ]
    | [ 11 (* ai *) :: r ]
    | [ 12 (* o *) :: r ]
    | [ 13 (* au *) :: r ] → match entry with
        [ "mnaa" | "zaa" | "saa#1" → rstem (* mnaaya zaaya avasaaya *)
        | _ → [ 10 :: r ] (* deya *)
        ]
    | [ 3 :: _ ] | [ 4 :: _ ] → strong
    | [ 5 (* u *) :: r ] → match entry with
        [ "stu" → [ 45 :: [ 2 :: r ] ] (* u -¿ aav *)
        | "yu#1" → [ 6 :: r ] (* u -¿ uu *)
        | "yu#2" → raise Not_attested
        | _ → strong
        ]
```

```
| [ 6 (* uu *) :: _ ] → match entry with
    [ "huu" → revcode "hav" (* havya WR (?) *)
    | _ → strong
    ]
| [ 7 (* .r *) :: _ ] → match entry with
    [ "dhv.r" → strong (* dhvarya *)
    | "d.r#1" | "v.r#2" → [ 32 :: rstem ] (* d.rtya v.rtya P{3,1,109} *)
      (* others as supplementary forms with interc t in record_pfp below *)
    | _ → long (* kaarya (k.rt .nyat) P{3,1,124} *)
    ]
| [ 8 (* .rr *) :: _ ] → match entry with
    [ "st.rr" → strong (* starya *)
    | _ → long
    ]
  (* now consonant rules - order of patterns important *)
| [ 22; 7 ] (* .rc *)
| [ 24; 7 ] (* .rj *) → strong (* arc arj *)
| [ 24; 7; 41 ] (* m.rj *) → long (* maarj P{7,2,114} *)
| [ 47; 7 ] (* .r.sya autonomous *)
| [ 32; 7; 17 ] (* k.rt *) → raise Not_attested (* k.rtya comes from k.r1 *)
| [ 48; 1 ] (* as1 *) →
        if entry = "as#1" then raise Not_attested (* use bhuu *)
                          else rstem (* asya - may overgenerate *)
| [ 48; 1; 46 ] (* zas *) → rstem
| [ 48; 2; 46 ] (* zaas *) → revcode "zaa.s" (* zaa.sya + zi.sya extra *)
| [ 33; 36; 1; 43; 19 ] (* granth *) → revcode "grath"
| [ 35; 1; 45 ] (* vadh/han *) → rstem (* vadhya *)
| [ 36; 1; 49 ] (* han *) → revcode "ghaat" (* (h=h') P{7,3,32+54} *)
| [ 35; 1; 42; 45 ] (* vyadh *) → revcode "vedh"
| [ 46; 1; 43; 37 ] (* praz *) → revcode "p.rcch"
| [ 46; 1; 37 ] (* paz *) → raise Not_attested (* pazya WR -Panini *)
| [ 46; 1; 45 ] (* vaz *) → rstem (* vazya (?) *)
| [ 49; 43; 1 ] (* arh *) → revcode "argh" (* arghya (h=h') *)
| [ 17; 1; 46 ] (* zak *)
| [ 49; 1; 48 ] (* sah *) → rstem (* zakya sahya P{3,1,99} -yat *)
| [ 43; 1; 22 ] (* car *) → rstem (* carya P{3,1,100} -yat *)
                                  (* but caarya obtainable as ca -.nyat *)
(* NB car gad mad yam also take -yat P{3,1,100}: record_extra_pfp_ya *)
| [ 24; 1 ] (* aj *) → rstem (* ajya *)
| [ c :: [ 1 :: _ ] ] when labial c → rstem (* P{3,1,98} -yat *)
```

```
    | [ c :: [ 1 :: r ] ] → [ c :: [ 2 :: r ] ]
                          (* a lengthened if last non labial *)
                          (* above often optional, see record_extra_pfp_ya below *)
    | [ c :: [ 7 :: _ ] ] → rstem (* d.rz1 v.r.s but NOT m.rj *)
    | [ c :: [ v :: _ ] ] when short_vowel v (* gunify *) → strong
    | _ → rstem
    ] in
  record_pfp_ya Primary ya_stem entry
 ;
(* Primary conjugation pfp in -ya for gana 10 *)
value pfp_ya_10 rstem entry =
  let pfp_stem = rfix rstem "ya" in
  record_part (Pfutp_ Primary pfp_stem entry)
 ;
(* Primary conjugation pfp in -aniiya *)
value pfp_aniiya rstem entry =
  let iya_stem =
     match entry with
     [ "uk.s" | "cint" → rstem | "yu#1" | "yu#2" → raise Not_attested
     | "dham" → revcode "dhmaa" (* P{7,3,78} *)
     | "vyadh" → revcode "vedh"
     | _ → match Word.mirror rstem with
            [ [ 4 :: _ ] | [ 6 :: _ ] → rstem (* ii- uu- no guna *)
            | _ → strong_stem entry rstem
            ]
     ] in
  record_pfp_aniiya Primary iya_stem entry
 ;
value record_pfp_10 entry rstem = do
  { try pfp_ya_10 rstem entry with [ Not_attested → () ]
  ; try pfp_aniiya rstem entry with [ Not_attested → () ]
  }
;
```

Absolutive and Past Participle

```
value record_part_ppp ppstem entry = do
  { record_part (Ppp_ Primary ppstem entry)
  ; record_part (Pppa_ Primary ppstem entry) (* pp-vat (krit tavat) *)
  }
;
```

```
value record_abso_ya form entry = enter1 entry (Invar (Primary, Absoya) form)
  and record_abso_tvaa form entry = enter1 entry (Absotvaa Primary form)
;
(* First absolutives in -ya *)
value record_abs_ya entry rstem w = do
  (* intercalate t for light roots Kiparsky159 Macdonell§165 *)
  { let absya =
        if light rstem then fix w "tya" (* check test light *)
        else let rst = match entry with
              [ (* roots in -m and -n in gana 8 P{6,4,37} *)
                  "van" | "man" | "tan#1" (* man also in gana 4 *)
              | "gam" | "nam" | "yam" | "han#1" (* anudatta ? *)
              | "kram" | "klam" | "zam#2" | "zram" (* P{6,4,15} *)
              | "daa#2" | "saa#1" | "sthaa#1" | "maa#1" (* P{7,4,40} *)
              | "daa#1" (* P{7,4,46} *)
              | "dhaa#1" (* P{7,4,42} *)
                    → rstem
              | "zii#1" → revcode "zay" (* P{7,4,22} *)
              | "arh" → revcode "argh" (* arghya (h=h') *)
              | _ → w
              ] in match entry with
                  [ "hi.ms" → code "hi.msya" (* no retroflex s Whitney§183 *)
                  | _ → fix rst "ya"
                  ] in
    record_abso_ya absya entry
  ; match entry with (* alternate forms in -ya and -tvaa *)
    [ "gam" | "tan#1" | "nam" | "man" | "van" | "han#1" →
      (* a+nasal optional assimilation to light roots *)
        record_abso_ya (fix w "tya") entry
    | "dhaa#1" → record_abso_tvaa (code "dhitvaa") entry
    | "plu" → record_abso_ya (code "pluuya") entry
    | "b.rh#1" → record_part_ppp (revcode "b.r.mhita") entry (* MW -WR *)
    | "v.r#2" → do { record_abso_tvaa (code "varitvaa") entry
                   ; record_abso_tvaa (code "variitvaa") entry
                   }
    | "kram" → record_abso_tvaa (code "krantvaa") entry (* P{6,4,18} *)
    | "zaas" → (* passive stem zi.s *)
        let w = revcode "zi.s" in do (* as if ipad=0 *)
        { record_part_ppp (rfix w "ta") entry
        ; record_abso_tvaa (fix w "tvaa") entry
```

```
        ; record_abso_ya (fix w "ya") entry
        }
    | _ → ()
    ]
  }
;
value alternate_pp = fun
  [ "m.r.s" | "svid#2" | "dh.r.s" | "puu#1" (* next roots of gu.na 1 have penultimate
"u" *)
  | "kul" | "k.sud" | "guh" | "jyut" | "dyut#1" | "mud#1" | "rud#1" | "ruh#1"
  | "lul" | "zuc#1" | "zubh#1" | "zu.s" → True
  | _ → False
  ]
;
(* Condition for extra abs in -tvaa with guna: root starts with consonant and ends in any
consonant but y or v and has i or u as penultimate. Given by P{1,2,26}. Example: sidh1 *)
value alternate_tvaa entry rstem =
  match Word.mirror rstem with (* double rev *)
  [ [ c :: _ ] → consonant c ∧ match rstem with
      [ [ 42 (* y *) :: _ ] | [ 45 (* v *) :: _ ] → False
      | [ c' :: rest ] → consonant c' ∧ match rest with
          [ [ 3 (* i *) :: _ ] | [ 5 (* u *) :: _ ] → True | _ → False ]
      | _ → False
      ]
  | _ → match entry with
          [ "t.r.s#1" | "m.r.s" | "k.rz" (* P{1,2,25} *)
          | "puu#1" (* P{1,2,22} *) → True
          | _ → False
          ]
  ]
;
(* Records the (reversed) ppp stem (computed by compute_ppp_stems) and builds absolu-
tives in -tvaa and -ya ( should be separated some day). *)
value record_ppp_abs_stems entry rstem ppstems =
  let process_ppstem = fun
      [ Na w → do
          { record_part_ppp (rfix w "na") entry
          ; let stem = match entry with (* roots in -d *)
              [ "k.sud" | "chad#1" | "chid#1" | "ch.rd" | "tud#1" | "t.rd" | "nud"
              | "pad#1" | "bhid#1" | "mid" | "vid#2" | "zad" | "sad#1" | "had"
```

```
        | "svid#2" → match w with
                      [ [ 36 (* n *) :: r ] → [ 34 (* d *) :: r ]
                      | _ → failwith "Anomaly␣Verbs"
                      ]
        | "vrazc" → revcode "v.rz" (* not v.rk *)
        | "und" | "skand" | "syand" → [ 34 (* d *) :: w ]
        | _ → w
        ] in match entry with
        [ "mid" →
               let abs_mid st = record_abso_tvaa (fix st "itvaa") entry in
               do { abs_mid stem; abs_mid (revcode "med") (* guna *)}
        | _ → do { record_abso_tvaa (fix stem "tvaa") entry
                 ; record_abso_ya (fix stem "ya") entry
                 }
        ]
    }
  | Ka w → do
      { record_part_ppp (rfix w "ka") entry (* zu.ska P{8,2,51} *)
      ; record_abso_ya (fix w "ya") entry
      }
  | Va w → do
      { record_part_ppp (rfix w "va") entry
      ; record_abso_tvaa (fix w "tvaa") entry
      ; record_abso_ya (fix w "ya") entry
      }
  | Ta w → do
      { if is_anit_pp entry rstem then record_part_ppp (rfix w "ta") entry
        else ((* taken care of as Tia *))
      ; if is_anit_tvaa entry rstem then record_abso_tvaa (fix w "tvaa") entry
        else ((* taken care of as Tia *))
      ; (* abs -ya computed whether set or anit *)
        match entry with
        [ "av" → record_abs_ya entry rstem (revcode "aav") (* -aavya *)
        | _ → record_abs_ya entry rstem w
        ]
      }
  | Tia w → let (ita, itvaa) = if entry = "grah" then ("iita","iitvaa")
                                else ("ita","itvaa") in do
      { if is_set_pp entry rstem then
            match entry with
```

```
              [ "dh.r.s" | "zii#1" (* "svid#2" "k.svid" "mid" P{1,2,19} *)
                 → record_part_ppp (rfix (strong w) ita) entry
              | _ → do
                { record_part_ppp (rfix w ita) entry
                ; if alternate_pp entry then
                      record_part_ppp (rfix (strong w) ita) entry
                  else ()
                }
              ]
           else ()
        ; if is_set_tvaa entry rstem then do
              { let tstem = match entry with
                     [ "m.rj" → lengthened rstem (* maarj *)
                     | "yaj#1" | "vyadh" | "grah" | "vrazc" | "praz" | "svap"
                     | "vaz" | "vac" | "vap" | "vap#1" | "vap#2" | "vad"
                     | "vas#1" | "vas#4" → w
                     | "siiv" → revcode "sev" (* gu.na *)
                     | "stambh" → rstem (* stabhita but stambhitvaa! *)
                     | _ → strong w
                     ] in
                record_abso_tvaa (fix tstem itvaa) entry
              ; if alternate_tvaa entry rstem then
                    record_abso_tvaa (fix w "itvaa") entry
                else ()
              }
           else ()
        }
     ] in
  iter process_ppstem ppstems
;
(* Simple version for denominatives - tentative *)
value record_ppp_abs_den ystem entry =
 let ppstem = trunc (revstem entry) in do
  { record_part_ppp (rfix ppstem "ita") entry
  ; match entry with
     [ "aakar.na" → record_abso_tvaa (fix ppstem "ya") entry (* fake abso-ya! *)
     | _ → record_abso_tvaa (fix ystem "itvaa") entry
     ]
  (* no general record_abso_ya since usually no preverb to denominatives *)
  }
```

```
;
(* Absolutive in -am - Macdonell§166 Stenzler§288 P{3,4,22} .namul *)
(* Registered both in Invar and in Absotvaa, since may be used with preverbs. *)
(* Used specially for verbs that may be iterated - having done again and again *)
value record_abso_am root =
  let record form = let word = code form in do
      { record_abso_tvaa word root (* no preverb *)
      ; record_abso_ya word root (* some preverb *)
      } in
  match root with
  [ "as#2" → record "aasam" (* may overgenerate *)
  | "ka.s" → record "kaa.sam" (* P{3,4,34} *)
  | "kram" → record "kraamam"
  | "k.r#1" → record "kaaram" (* P{3,4,26-28} *)
  | "khan" → record "khaanam"
  | "grah" → record "graaham"
  | "c.rt" → record "c.rtam"
  | "jiiv" → record "jiivam" (* P{3,4,30} *)
  | "j~naa#1" → record "j~naayam"
  | "t.r.s#1" → record "tar.sam"
  | "daa#1" → record "daayam"
  | "naz#1" → record "naazam"
  | "paa#1" → record "paayam"
  | "pi.s" → record "pe.sam" (* P{3,4,35+38} *)
  | "pu.s#1" → record "po.sam" (* P{3,4,40} *)
  | "puur#1" → record "puuram" (* P{3,4,31} *)
  | "praz" → record "p.rccham"
  | "bandh" → record "bandham"
  | "bhuj#1" → record "bhojam"
  | "bhuu#1" → record "bhaavam"
  | "vad" → record "vaadam"
  | "v.rt#1" → record "vartam" (* P{3,4,39} *)
  | "zru" → record "zraavam"
  | "sa~nj" → record "sa~ngam"
  | "s.r" → record "saaram"
  | "s.rp" → record "sarpam"
  | "skand" → record "skandam"
  | "stambh" → record "stambham"
  | "han" → record "ghaatam" (* P{3,4,36+37} *)
  | "knuu" → record "knopam" (* from causative *)
```

```
  | _ → ()
  ]
(* NB Bandharkar: colloquial expressions iic+V.namul suivi de forme finie de V *)
(* eg "hastagraaha.m␣g.r.naati" il tient par la main *)
(* Should be also definable for causative, eg knopam caknuu P{3,4,33} *)
;
(* absolutive of secondary conjugations *)
value record_absolutive c abs_stem_tvaa abs_stem_ya intercal entry =
  let record_abso_ya form = enter1 entry (Invar (c, Absoya) form)
  and record_abso_tvaa form = enter1 entry (Absotvaa c form) in do
  { let sfx = if intercal then "itvaa" else "tvaa" in
    record_abso_tvaa (fix abs_stem_tvaa sfx)
  ; record_abso_ya (fix abs_stem_ya "ya")
  }
;
value record_pppca cpstem cstem entry =
  let ppstem = [ 1 :: [ 32 :: [ 3 :: cpstem ] ] ] (* cp-ita *) in do
  { record_part (Ppp_ Causative ppstem entry)
  ; record_part (Pppa_ Causative ppstem entry) (* pp-vat *)
  ; let abs_stem_ya = match entry with (* Whitney§1051d *)
        [ "aap" | ".r" | ".rc#1" | ".rdh" | "kal" | "k.lp" | "kram" | "gam"
        | "jan" | "jval" | "dh.r" | "rac" | "zam#1" | "p.rr" | "bhak.s" | "v.rj"
            → cstem (* retains ay: -gamayya to distinguish from -gamya *)
        | _ → cpstem (* eg -vaadya -vezya *)
        ]
    and abs_stem_tvaa = cstem (* retains ay: gamayitvaa *) in
    record_absolutive Causative abs_stem_tvaa abs_stem_ya True entry
       (* cp-ita -¿ cp-ayitvaa, -cp-ayya ou -cp-ya *)
  }
;
value record_pppdes stem entry =
  let ppstem = [ 1 :: [ 32 :: [ 3 :: stem ] ] ] in (* s-ita *) do
  { record_part (Ppp_ Desiderative ppstem entry)
  ; record_part (Pppa_ Desiderative ppstem entry) (* pp-vat *)
  ; let abs_stem_tvaa = [ 3 :: stem ] (* s-i *)
    and abs_stem_ya = stem in
    record_absolutive Desiderative abs_stem_tvaa abs_stem_ya False entry
       (* s-ita -¿ s-itvaa, -s-iya *)
  }
;
```

Intensive or frequentative

```
value compute_intensive_presenta strong weak iiflag entry =
(* info not used for check because of ambiguity of third sg - we want no error message in the
conjugation engine display *)
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = (person, fix3w weak iiflag False suff) in do
  { enter1 entry (Conju intensa
   [ (Singular,
         [ conjugs First "mi"
         ; conjugw First "iimi"
         ; conjugs Second "si"
         ; conjugw Second "iisi"
         ; conjugs Third "ti"
         ; conjugw Third "iiti" ])
   ; (Dual,
         [ conjugw First "vas"
         ; conjugw Second "thas"
         ; conjugw Third "tas"
         ])
   ; (Plural,
         [ conjugw First "mas"
         ; conjugw Second "tha"
         ; conjugw Third "ati"
         ])
   ])
  ; let wstem = if iiflag then match weak with
          [ [ 4 :: w ] → w (* ii disappears before vowels in special roots *)
          | _ → failwith "Wrong weak stem of special intensive"
          ]
                    else weak in (* 3rd pl weak stem *)
    record_part (Pprared_ Intensive wstem entry)
  ; if entry = "draa#1" then let ppstem = revcode "daridrita" in
                                        record_part (Ppp_ Intensive ppstem entry)
    else ((* TODO *))
  }
;
value compute_intensive_impfta strong weak iiflag entry =
  let conjugs person suff = (person, fix_augment strong suff)
  and conjugw person suff = (person, fix3w_augment weak iiflag False suff) in
  enter1 entry (Conju intimpfta
```

```
  [ (Singular,
        [ conjugs First "am"
        ; conjugs Second "s"
        ; conjugw Second "iis"
        ; conjugs Third "t"
        ; conjugw Second "iit"
        ])
  ; (Dual,
        [ conjugw First "va"
        ; conjugw Second "tam"
        ; conjugw Third "taam"
        ])
  ; (Plural,
        [ conjugw First "ma"
        ; conjugw Second "ta"
        ; conjugw Third "ur"
        ])
  ])
;
value compute_intensive_optativea weak iiflag entry =
  let conjugw person suff = (person, fix3w weak iiflag False suff) in
  enter1 entry (conjug_optativea int_gana Intensive conjugw)
;
value compute_intensive_imperativea strong weak iiflag entry =
  let conjugs person suff = (person, fix strong suff)
  and conjugw person suff = (person, fix3w weak iiflag False suff) in
  enter1 entry (Conju intimpera
  [ (Singular,
        [ conjugs First "aani"
        ; (Second, match weak with
            [ [ c :: _ ] → fix3w weak iiflag False suff
              where suff = if vowel c then "hi" (* "dhi" or "hi" after vowel *)
                           else "dhi"
            | _ → error_empty 28
            ] )
        ; conjugs Third "tu"
        ; conjugs Third "iitu"
        ])
  ; (Dual,
        [ conjugs First "aava"
```

```
        ; conjugw Second "tam"
        ; conjugw Third "taam"
        ])
  ; (Plural,
        [ conjugs First "aama"
        ; conjugw Second "ta"
        ; conjugw Third "atu"
        ])
  ])
;
```

(* Reduplication for the intensive conjugation - TODO Macdonell§173 *value redup_int entry* = ... For the moment, the reduplicated stem is read from the lexicon. It is not clear whether there are enough intensive forms to warrant a paradigm rather than a table. *)

Similar to *compute_active_present3* with Intensive, plus optional ii forms

```
value compute_intensivea wstem sstem entry third =
  let iiflag = False in (* let (sstem, wstem) = redup_int entry in *) do
  { compute_intensive_presenta sstem wstem iiflag entry (* no third *)
  ; compute_intensive_impfta sstem wstem iiflag entry
  ; compute_intensive_optativea wstem iiflag entry
  ; compute_intensive_imperativea sstem wstem iiflag entry
  ; if entry ="bhuu#1" (* bobhoti *) then
       let stem = revcode "bobhav" in build_perpft Intensive stem entry
    else () (* EXPERIMENTAL *)
  }
;
(* Takes reduplicated stem from lexicon. A generative version would use redup3 and add
"ya" like passive *)
value compute_intensivem = compute_thematic_middle int_gana Intensive
and compute_intensivem2 st =
  compute_athematic_present3m Intensive int_gana st False
;
```

Present system

```
value compute_present_system entry rstem gana pada third =
  try
   (* pada=True for active (parasmaipade), False for middle (aatmanepade) *)
   let padam = if third = [] then False else pada in (* artifact for fake below *)
   match gana with
   [ 1 | 4 | 6 | 10 (* thematic conjugation *) →
     let compute_thematic_present stem =
```

```
      match voices_of_gana gana entry with
      [ Para → (* active only *) if pada then
           compute_thematic_active gana Primary stem entry third
           else emit_warning ("Unexpected middle form: " ^ entry)
      | Atma → (* middle only *)
           if padam then emit_warning ("Unexpected active form: " ^ entry)
           else compute_thematic_middle gana Primary stem entry third
      | Ubha →
           let thirda = if pada then third else []
           and thirdm = if pada then [] else third in do
           { compute_thematic_active gana Primary stem entry thirda
           ; compute_thematic_middle gana Primary stem entry thirdm
           }
      ] in
   match gana with
   [ 1 → match entry with
         [ "kram" → do (* 2 forms Whitney§745d *)
             { compute_thematic_present rstem
             ; compute_thematic_present (revcode "kraam") (* lengthen *)
             }
         | "cam" → do (* 2 forms Whitney§745d *)
             { compute_thematic_present rstem
             ; compute_thematic_present (revcode "caam") (* lengthen *)
             }
         | "t.rr" → do (* 2 forms *)
             { compute_thematic_present (revcode "tir")
             ; compute_thematic_present (revcode "tar")
             }
         | "manth" → do (* 2 forms *)
             { compute_thematic_present rstem
             ; compute_thematic_present (revcode "math") (* suppr nasal *)
             }
         | "a~nc" → do (* 2 forms *)
             { compute_thematic_present rstem
             ; compute_thematic_present (revcode "ac") (* suppr nasal *)
             }
         | "uuh" → do (* 2 forms *)
             { compute_thematic_present rstem
             (* compute_thematic_middle 1 Primary (strong rstem) entry (if pada then [] else third
(* ohate ved *) *)
```

```
        }
      | "huu" → do (* 2 forms *) (* hvayati, havate *)
        { compute_thematic_present (revcode "hve")
        ; compute_thematic_middle 1 Primary (revcode "hav") entry
              (if pada then [] else third) (* havate *)
        }
      | _ → let stem = match entry with
        [ ".r" → revcode ".rcch" (* P{7,3,78} Whitney§747 *)
        | "gam" → revcode "gacch" (* P{7,3,77} Whitney§747 *)
        | "yam" → revcode "yacch" (* P{7,3,77} *)
        | "yu#2" → revcode "yucch"
        | "kuc" → revcode "ku~nc" (* add nasal *)
        | "da.mz" → revcode "daz" (* suppr penult nasal P{6,4,25} *)
        | "ra~nj" → revcode "raj" (* id *)
        | "sa~nj" → revcode "saj" (* id *)
        | "sva~nj" → revcode "svaj" (* id *)
        | "daa#1" → revcode "dad" (* dupl Whitney§672 ved *)
            (* P{7,3,78}: yacch for prayacch in meaning of giving *)
            (* also "s.r" -¿ "dhau" (corresponds to dhaav1) "dmaa" -¿ "dham" (cf
ppstem) *)
        | "dhaa#1" → revcode "dadh" (* id *)
        | "paa#1" → revcode "pib" (* fake 3rd gana P{7,3,78} *)
        | "ghraa" → revcode "jighr" (* id P{7,3,78} *)
        | "sthaa#1" → revcode "ti.s.th" (* id P{7,3,78} *)
        | "d.rh" → revcode "d.r.mh" (* .rh -¿ .r.mh *)
        | "b.rh#1" → revcode "b.r.mh" (* WR; Bucknell adds barhati *)
        | "iir.s" | "gaa#2" (* = gai *)
        | "daa#3" | "dyaa" | "dhyaa" | "pyaa" (* = pyai *)
        | "zu.s" | "zyaa" | "sphaa" → [ 42 (* y *) :: rstem ] (* add y *)
        | "maa#4" → revcode "may" (* shorten add y *)
        | "vyaa" → revcode "vyay"
        | "zuu" → revcode "zve" (* zvayati - similar to huu/hve *)
        | "knuu" → revcode "knuuy"
        | "guh" → revcode "guuh" (* lengthen P{6,4,89} *)
        | "grah" → revcode "g.rh.n" (* WR *)
        | "das" → revcode "daas"
        | "mnaa" → revcode "man" (* P{7,3,78} *)
        | "zad" → revcode "ziiy" (* P{7,3,78} *)
        | "sad#1" → revcode "siid" (* P{7,3,78} *)
        | ".sad" → revcode ".siid" (* fictive retro-root of sad1 *)
```

```
          | "m.rj" → mrijify (revcode "maarj") (* vriddhi *)
          | "yaj#1" | "vraj" | "raaj#1" | "bhraaj" → mrijify rstem
          | "kliiba" | "puula" → (* kliibate etc *) (* denominative verbs *)
              Phonetics.trunc_a rstem (* since thematic a added *)
          | "k.rp" → rstem
          | _ → strong rstem (* default *)
          ] in compute_thematic_present stem
        ]
  | 4 → let weak = match entry with
        [ "bhra.mz" → revcode "bhraz" (* suppr penult nasal *)
        | "ra~nj" → revcode "raj" (* id *)
        | "i" → revcode "ii"
        | "jan" → revcode "jaa"
        | "kan" → revcode "kaa"
        | "klam" → revcode "klaam"
        | "j.rr" → revcode "jiir"
        | "jyaa#1" → revcode "jii"
        | "tam" → revcode "taam"
        | "dam#1" → revcode "daam"
        | "daa#2" → revcode "d"
        | "d.rz#1" → raise Not_attested (* replaced by paz P{7,3,78} *)
        | "dhaa#2" → revcode "dha"
        | "bhram" → revcode "bhraam"
        | "mad#1" → revcode "maad"
        | "mid" → revcode "med"
        | "ri" → revcode "rii"
        | "vaa#3" → revcode "va" (* bizarre - should be ve class 1 *)
        | "vyadh" → revcode "vidh"
        | "zam#1" → revcode "zaam"
        | "zaa" → revcode "z"
        | "zram" → revcode "zraam"
        | "saa#1" → revcode "s"
        | _ → rstem
        ] in
    let ystem = [ 42 :: weak ] (* root-y *) in
    compute_thematic_present ystem
  | 6 → let stem = match rstem with
        [ [ 3 :: rest ] | [ 4 :: rest ] → [ 42 :: [ 3 :: rest ] ]
          (* -.i -¿ -iy eg k.si pii *)
        | [ 5 :: rest ] | [ 6 :: rest ] → [ 45 :: [ 5 :: rest ] ]
```

```
        (* -.u -¿ -uv eg dhru also kuu -¿ kuv *)
      | [ 7 :: rest ] → [ 42 :: [ 3 :: [ 43 :: rest ] ] ] (* mriyate *)
      | [ 8 :: rest ] → match entry with
             [ "p.rr" → revcode "p.r.n" (* ugly duckling *)
             | _ → [ 43 :: [ 3 :: rest ] ] (* .rr/ir *)
             ]
        (* -.rr -¿ -ir eg t.rr *)
      | _ → match entry with
             [ "i.s#1" → revcode "icch" (* P{7,3,78} *)
             | "vas#4" → revcode "ucch"
             | ".rj" → revcode ".r~nj"
             | "k.rt#1" → revcode "k.rnt"
             | "piz#1" → revcode "pi.mz"
             | "muc#1" → revcode "mu~nc"
             | "rudh#2" → revcode "rundh"
             | "sic" → revcode "si~nc"
             | "lip" → revcode "limp"
             | "lup" → revcode "lump"
             | "vid#2" → revcode "vind"
             | "praz" → revcode "p.rcch" (* ra/.r *)
             | "vrazc" → revcode "v.rzc" (* id déploiement vocalique *)
             | "s.rj" → mrijify rstem
             | _ → rstem (* root stem *)
             ]
      ] in compute_thematic_present stem
| 10 → let process10 y_stem = do
             { compute_thematic_present y_stem
             ; build_perpft Primary y_stem entry
             ; let perstem = [ 3 :: y_stem ] (* -ayi *) in
               perif Primary perstem entry
             } in
    match entry with
    [ "tul" → do (* 2 forms *)
        { process10 (revcode "tulay")
        ; process10 (revcode "tolay") (* guna *)
        }
    | "gup" → process10 (revcode "gopay") (* guna *)
    | "mid" → process10 (revcode "minday") (* nasal *)
    | _ → let base_stem = strengthen_10 rstem entry in
          let ystem = rev (sandhi base_stem [ 1; 42 ] (* ay *)) in
```

```
                process10 ystem
          ]
      | _ → failwith "Anomaly␣Verbs"
      ] (∗ end of thematic conjugation ∗)
  | 2 → (∗ athematic conjugation: 2nd class (root class) ∗)
      let set = augment_ii entry
      and sstem = strong_stem entry rstem
      and wstem = if entry ="as#1" then [ 48 ] else weak_stem entry rstem in do
      { match voices_of_gana 2 entry with
        [ Para → (∗ active only ∗) if pada then
            compute_active_present2 sstem wstem set entry third
            else emit_warning ("Unexpected␣middle␣form:␣" ^ entry)
        | Atma (∗ middle only ∗) →
            if padam then emit_warning ("Unexpected␣active␣form:␣" ^ entry)
            else compute_middle_present2 sstem wstem set entry third
        | Ubha →
            let thirda = if pada then third else [ ]
            and thirdm = if pada then [ ] else third in do
            { compute_active_present2 sstem wstem set entry thirda
            ; compute_middle_present2 sstem wstem set entry thirdm
            }
        ]
      ; match entry with (∗ special cases ∗)
        [ "as#1" → (∗ rare middle forms of as ∗)
          compute_athematic_present2m sstem [ 48 ] set entry (code "ste")
(∗ | "vac" → let weak = revcode "vaz" (× douteux − WR ×) in compute_athematic_present2m sstem
        | _ → ()
        ]
      }
  | 3 → let (sstem, wstem, iiflag) = redup3 entry rstem in
            match voices_of_gana 3 entry with
        [ Para → if pada then
            compute_active_present3 sstem wstem iiflag entry third
            else emit_warning ("Unexpected␣middle␣form:␣" ^ entry)
        | Atma →
            if padam then emit_warning ("Unexpected␣active␣form:␣" ^ entry)
            else compute_middle_present3 sstem wstem iiflag entry third
        | Ubha →
            let thirda = if pada then third else [ ]
            and thirdm = if pada then [ ] else third in do
```

```
          { compute_active_present3 sstem wstem iiflag entry thirda
          ; compute_middle_present3 sstem wstem iiflag entry thirdm
          }
        ]
  | 5 → (* athematic conjugation: 5th class *)
    let (stem, vow) = match rstem with
        [ [ 36; 3 ] (* in *) → ([ 3 ] (* i *), True) (* Whitney§716a *)
        | [ 5; 43; 46 ] (* zru *) → ([ 7; 46 ] (* z.r *), True)
        | [ 40 :: [ 41 :: r ] ] → ([ 40 :: r ], False) (* skambh stambh *)
          (* possibly other penultimate nasal lopa ? *)
        | [ c :: rest ] → if vowel c then ([ short c :: rest ], True)
                           else (rstem, False)
        | [] → error_empty 29
        ] in
    let wstem = rev (sandhi stem [ 36; 5 ]) (* stem-nu *)
    and sstem = rev (sandhi stem [ 36; 12 ]) (* stem-no *) in do
    { compute_present5 5 sstem wstem vow entry third pada padam
    ; if entry = "v.r#1" then (* extra derivation *)
          let wstem = revcode "uur.nu" and sstem = revcode "uur.no" in
          compute_present5 5 sstem wstem True entry third pada padam
      else ()
    }
  | 7 → (* athematic conjugation: 7th class *)
    match rstem with
    [ [ c :: rest ] when consonant c →
      let stem = match rest with
          [ [ hd :: tl ] → if nasal hd then tl else rest (* hi.ms *)
          | [] → error_empty 30
          ]
      and nasal = homonasal c in
      let wstem =
        if entry = "t.rh" then revcode "t.rfh"
        else [ c :: rev (sandhi stem [ nasal ]) ] (* stem-n *)
      and sstem =
        if entry = "t.rh" then [ c :: rev (sandhi stem [ 36; 10 (* -ne *)]) ]
        else [ c :: rev (sandhi stem [ 36; 1 ]) ] (* stem-na *) in
      compute_present7 sstem wstem entry third pada padam
    | _ → warning (entry ^ "␣atypic␣7\n")
    ]
  | 8 → (* k.r1 k.san tan1 man san1 *)
```

```
    match rstem with
    [ [ 36 (* n *) :: rest ] →
       let wstem = rev (sandhi rest [ 36; 5 ]) (* stem-nu *)
       and sstem = rev (sandhi rest [ 36; 12 ]) (* stem-no *) in
       compute_present5 8 sstem wstem True entry third pada padam
    | [ 7; 17 ] (* k.r *) →
       let wstem = revcode "kuru"
       and short = revcode "kur" (* before suffix -m -y -v Macdonell§134E *)
       and sstem = revcode "karo" in
       compute_presentk sstem wstem short entry third
    | _ → warning (entry ^ "␣atypic␣8\n")
    ]
| 9 → let (stem, vow) = match entry with (* vow = vowel ending root *)
        [ "j~naa#1" → (revcode "jaa" ,True) (* P{7,3,79} *)
        | "jyaa#1" → (revcode "ji" ,True)
        | "zraa" → (revcode "zrii" ,True)
        | "umbh" → (revcode "ubh" ,False) (* elision penul nasal *)
        | "granth" → (revcode "grath",False) (* id *)
        | "bandh" → (revcode "badh" ,False) (* id *)
        | "skambh" → (revcode "skabh",False) (* id *)
        | "stambh" → (revcode "stabh",False) (* id *)
        | "grah" → (revcode "g.rh" ,False) (* plus "g.rbh" added below *)
        | "k.sii" → (revcode "k.si" ,True)
        | _ → match rstem with
            [ [ c :: w ] → (st, vowel c)
              where st = if c = 6 (* uu *) then [ 5 :: w ] (* Whitney§728a *)
                         else if c = 8 (* .rr *) then [ 7 :: w ] else rstem
            | [] → error_empty 31
            ]
        ] in (* Macdonell§127.6 *)
    (* NB Retroflexion prevented in k.subh: k.subhnaati P{8,4,39} *)
    let retn = if Int_sandhi.retron stem then
        if entry = "k.subh" then 36 (* n *) else 31 (* .n *) else 36 (* n *)
    and glue = if entry = "k.subh" then List2.unstack else sandhi in
    let sstem = rev (glue stem [ 36; 2 ]) (* stem-naa *) (* naa accented *)
    and wstem = rev (glue stem [ 36; 4 ]) (* stem-nii *) (* nii unaccented *)
    and short = [ retn :: stem ] (* stem-n *) in do
    { compute_present9 sstem wstem short vow stem entry third pada padam
    ; if entry = "grah" then (* ved alternative form "g.rbh" Vt1 P{8,2,35} *)
        let stem = revcode "g.rbh" in
```

```
        let sstem = rev (sandhi stem [ 36; 2 ]) (* stem-naa *)
        and wstem = rev (sandhi stem [ 36; 4 ]) (* stem-nii *)
        and short = [ 31 :: stem ] (* stem-.n *) in
        compute_present9 sstem wstem short vow stem entry [] pada padam
      else ()
    }
  | 0 → () (* secondary conjugations - unused in this version *)
  | _ → failwith "Illegal␣present␣class"
  ]
 with [ Not_attested → () ]
(* end Present system *)
;
```

Passive system
Passive future participle (gerundive) in -ya and -aniiya

```
value record_pfp entry rstem = do
 { try pfp_ya rstem entry with [ Not_attested → () ]
 ; try pfp_aniiya rstem entry with [ Not_attested → () ]
 ; (* Supplements *)
   let record_extra_pfp_ya form =
       record_part (Pfutp_ Primary (revcode form) entry) in
   match entry with
   [ "k.r#1" (* P{3,1,120} .duk.r n + kyap *)
   | "stu" | "bh.r" | "i" | "m.r" → (* P{3,1,109} Renou§155e *)
     (* intercalate t after roots ending in short vowel Renou§146 *)
     let pfp_tya = rfix rstem "tya" in (* k.rtya stutya bh.rtya itya m.rtya *)
     record_part (Pfutp_ Primary pfp_tya entry)
   | "ju.s" → record_extra_pfp_ya "ju.sya" (* jo.sya P{3,1,109} *)
   | "khan" → do
     { record_extra_pfp_ya "khaanya" (* add to khanya P{3,1,123} *)
     ; record_extra_pfp_ya "kheya" (* further P{3,1,111} *)
     }
   | "ji" → do
     { record_extra_pfp_ya "jitya" (* Renou§155e P{3,1,117} *)
     ; record_extra_pfp_ya "jayya" (* (jeya) P{6,1,81} *)
     }
   | "k.sii" → record_extra_pfp_ya "k.sayya" (* (k.seya) P{6,1,81} *)
   | "grah" → record_extra_pfp_ya "g.rhya" (* P{3,1,119} *)
   | "cuu.s" → record_extra_pfp_ya "co.sya"
   | "ci" → do
```

```
      { record_extra_pfp_ya "caayya"
        (* P{3,1,131} fire only with pari- upa- sam- *)
      ; record_extra_pfp_ya "citya" (* P{3,1,131} in sense of fire *)
      }
    | "vad" → do
      { record_extra_pfp_ya "udya" (* P{3,1,106} for brahmodya *)
      ; record_extra_pfp_ya "vadya" (* id for brahmavadya sn *)
      }
    | "bhuu#1" → record_extra_pfp_ya "bhaavya" (* (bhavya) P{3,1,123} *)
      (* NB "bhuuya" is lexicalized as noun - P{3,1,107} *)
    | "m.rj" → record_extra_pfp_ya "m.rjya" (* (maargya) P{3,1,113} *)
    | "yuj#1" → record_extra_pfp_ya "yugya" (* (yogya) P{3,1,121} *)
    | "v.r#2" → record_extra_pfp_ya "vare.nya" (* vara.niiya (-aniiya) *)
    | "guh" → record_extra_pfp_ya "guhya" (* Vart P{3,1,109} *)
    | "duh#1" → record_extra_pfp_ya "duhya" (* idem *)
    | "za.ms" → record_extra_pfp_ya "zasya" (* idem *)
    | "zaas" → record_extra_pfp_ya "zi.sya" (* P{3,1,109} *)
      (* Following examples show that gunification is often optional. *)
      (* Some of the following forms seem actually preferable. *)
    | ".r" → record_extra_pfp_ya "arya" (* (aarya) P{3,1,103} (owner) *)
    | "kup" → record_extra_pfp_ya "kupya" (* (kopya) P{3,1,114} *)
    | "gad" → record_extra_pfp_ya "gadya" (* gaadya for pv- P{3,1,100} *)
    | "mad" → record_extra_pfp_ya "madya" (* maadya for pv- P{3,1,100} *)
    | "tyaj#1" → record_extra_pfp_ya "tyajya" (* for sa.mtyajya (tyaajya) *)
    | "bhid#1" → record_extra_pfp_ya "bhidya" (* P{3,1,115} for river *)
    | "d.rz#1" → record_extra_pfp_ya "darzya" (* WR only RV *)
    | "yaj#1" → record_extra_pfp_ya "yajya" (* devayajya P{3,1,123} *)
    | "yat" → record_extra_pfp_ya "yatya" (* Vart P{3,1,97} -WR *)
    | "ruc#1" → record_extra_pfp_ya "rucya" (* (rocya) P{3,1,114} *)
    | "va~nc" → record_extra_pfp_ya "va~ncya" (* P{7,3,63} for motion *)
    | "vah#1" → record_extra_pfp_ya "vahya" (* (vaahya) P{3,1,102} instr *)
    | "v.r.s" → record_extra_pfp_ya "var.sya" (* P{3,1,120} (v.r.sya) *)
    | "sa~nj" → record_extra_pfp_ya "sajya" (* for prasajya (not Paninian?) *)
    (* ? takya catya hasya *)
    | _ → ()
    ]
  }
;
```

Gana 11. Denominatives

Denominative verbs are given ga.na 11, and their stems are computed by *den_stem_a* and

den_stem_m below (for Para and Atma respectively). They are derivative verbs from dative forms of substantives. Roots kept in ga.na 10 (debatable, this is subject to change), are: arth ka.n.d kath kal kiirt kuts ga.n garh gup gha.t.t cint cur .damb tandr tark tul bharts mantr m.r.d rac rah ruup var.n lok suud sp.rh
Also gave.s, because possible ga.na 1 and pp - should be added separately
Also lelaa, which has a strange status (marked as verb rather than root)
asu is bizarre, lexicalized under asuuya
The next two functions are used only by the Grammar interface, the forms memorized are computed from the lexicalized 3rd sg form
BEWARE. the entry forms given in the next two functions must be in normalized form - no non-genuine anusvaara This should be replaced by the recording of the 3rd sg form, like others.

```
value den_stem_a entry = (* in general transitive Whitney§1059c *)
  let rstem = revstem entry in
  match entry with
  [ "putrakaama" | "rathakaama" (* P{3,1,9} *)
  | "sukha" | "du.hkha" (* also "adhvara" "m.rga" below *)
  | "i.sudhi" | "gadgada" (* P{3,1,27} *)
  | "agada" (* Kale§660 *) | "iras" (* — "pu.spa" replaced by root pu.sp *)
     → trunc rstem (* -()yati *) (* lopa *)
  (* — "maarg" — "mok.s" — "lak.s" — "suuc" -¿ 1 :: rstem (* -ayati *) presently
roots class 10 *)
  | "kutsaa" | "maalaa" | "mudraa" | "medhaa"
     → [ 1 :: trunc_aa rstem ] (* -()ayati - shortening final aa *)
  | "udazru"
     → [ 1 :: trunc_u rstem ] (* -()ayati - final u becomes a *)
  | "agha" | "azana#2" | "azva" | "ka.n.du" | "khela" | "jihma" | "pramada"
  | "lohita" | "mantu" | "manda" | "valgu" | "sakhi" | "samudra#1"
    (* to become P{3,1,13} kya.s *)
  | "asu" (* lexicalized under "asuuya" *)
     → lengthen rstem (* lengthening -aayati *)
  | "asuuya" (* "asu" lengthened *) | "gomaya" | "vyaya" (* euphony *)
     → trunc (trunc rstem)
  | (* "artha" —*) "veda" | "satya" (* P{3,1,25} Vt. *)
     → [ 1 :: [ 37 :: [ 2 :: trunc rstem ] ] ] (* -aapayati - interc p *)
  (* — (* very rare Whitney§1059d e.g. "putra" *) -¿ 3 :: trunc_a rstem (* -()iyati *)
*)
  | "adhvara" | "tavi.sa" | "putra" | "praasaada" (* treat as P{3,1,10} *)
  | "udaka" | "kavi" | "dhana" | "maa.msa" | "vastra" (* desire Kale§643 *)
     → [ 4 :: trunc rstem ] (* -()iiyati *) (* P{3,1,8} kyac *)
```

```
  | "kart.r" → [ 4 :: [ 43 :: trunc rstem ] ] (* .r -¿ rii Kale§642 *)
  | "go" → [ 45 :: [ 1 :: trunc rstem ] ] (* o -¿ av Kale§642 *)
  | "nau#1" → [ 45 :: [ 2 :: trunc rstem ] ] (* au -¿ aav Kale§642 *)
  | "raajan" → [ 4 :: trunc (trunc rstem) ] (* nasal amui Kale§642 *)
    (* now the general case: keep the nominal stem - to cause (transitive) *)
  | "a.mza" | "afka" | "afkha" | "andha" | "aparok.sa" | "apahasta" | "amitra"
  | "aakar.na" | "aakula" | "aavila" | "i.sa" | "unmuula" | "upahasta"
  | "ka.thora" | "kadartha" | "kar.na" | "kalafka" | "kalu.sa" | "kavala"
  | "ku.t.ta" | "kusuma" | "kha.da" | "garva" | "gocara" | "gopaa" | "carca"
  | "cuur.na" | "chala" | "chidra" | "tantra" | "tapas" | "tarafga" | "taru.na"
  | "tuhina" | "da.n.da" | "deva" | "dola" | "dravat" | "dhiira#1"
  | "nirmuula" | "nuutana" | "pa.tapa.taa" | "pallava"
  | "pavitra" | "paaza" | "pi.n.da" | "pulaka" | "puula" | "pratikuula"
  | "prati.sedha" | "pradak.si.na" | "prasaada" | "bhi.saj" (* — "mantra" *)
  | "malina" | "mizra" | "mukula" | "mukhara" | "mu.n.da" | "muutra"
  | "m.rga" | "yantra" | "rasa" | "ruuk.sa" | "lagha" (* u -¿ a *)
  (*— "var.na"*) | "vaasa#3" | "vizada" | "vra.na" | "zaanta" | "zithila"
  | "zyena" | ".sa.n.dha" | "sapi.n.da" | "saphala" | "sabhaaja" | "saantva"
  | "saavadhaana" | "suutra" | "stena" (* practice P{3,1,15} *)
  | "u.sas" | "namas" | "varivas" (* do P{3,1,19} *)
  | "utpuccha" (* do P{3,1,20} *)
  | "zlak.s.na" (* make P{3,1,21} *)
  | "lava.na" (* desire Kale§645 P{3,1,21} *)
  | "udan" (* Kale§645 *)
  | "hala" (* take P{3,1,21} *)
  | "kelaa" | "rekhaa" | "tiras" | "uras" | "payas" (* Kale§660 *)
  | "vaac" (* consonant Kale§642 *)
  | "dantura" (* possess *)
  | "k.r.s.na" (* agir comme *)
  | "viira" | "zabda" | "tira" (* MW *) | "ma~njara" | "sraja" | "manas"
      → rstem (* -yati *) (* standard causative meaning *)
  | "madhu" | "v.r.sa" (* also madhvasyati v.r.siiyati *)
  | "k.siira" (* also putra *)
      → [ 48 :: rstem ] (* -syati *) (* Kale§643 *)
  | _ → failwith ("Unknown␣denominative␣" ^ entry)
  ]
;
value den_stem_m entry = (* in general intransitive or reflexive Whitney§1059c *)
  let rstem = revstem entry in
  match entry with
```

```
  [ "i.sa" | "utpuccha" | "kuha" | "carca" | "manas" | "muutra"
    (*— "artha" — "mantra" now ga.na 10 arth mantr *)
  | "m.rga" | "viira" | "safgraama" | "suutra" (* also zithila below *)
      → rstem (* (a)-yate *)
  | "asuuya" (* "asu" lengthened *) | "vyaya" (* euphony *)
      → trunc (trunc rstem)
  | "tavi.sa" | "citra" (* do P{3,1,19} *) | "sajja"
      → [ 4 :: trunc_a rstem ] (* -()iiyate *)
(* — "arth" -¿ 1 :: rstem - arthayate for lexicon access - now ga.na 10 *)
  | "apsaras" | "sumanas" (* act as, become P{3,1,11-12} *)
  | "unmanas"
  | "uu.sman" (* emit P{3,1,16} *)
  | "raajan" (* play the role of *)
      → lengthen (trunc rstem) (* final consonant dropped *)
    (* now the general case: lengthen the nominal vowel stem *)
  | "pa.tapa.taa" | "mahii#2" | "m.r.saa"
  | "laalaa" | "svalpazilaa" | "vimanaa"
  | "ajira" | "kalu.sa" | "k.rpa.na" | "kliiba" | "garva" | "jala" | "jihma"
  | "taru.na" | "nika.sa" | "parok.sa" | "piiyuu.savar.sa" | "pu.spa" | "priya"
  | "bh.rza" | "maalyagu.na" | "lohita" | "zalabha" | "zithila" | "ziighra"
  | "zyaama" | "zyena" | "safka.ta"
  | "ka.n.du" | "karu.na" | "sukha" | "du.hkha" (* feel P{3,1,18} *)
(* Ga.nasukhaadi take suffix kyaf in -aayate : sukha,du.hkha,t.rpta,k.rcchra,asra,aasra,aliika,pratiipa,ka
*)
  | "t.rpta" (* -MW *)
  | "abhra" | "ka.nva" | "kalaha" | "k.sepa" | "megha" | "vaira" | "zabda"
  | "z.rfga" (* do P{3,1,17} *)
  | "durdina" | "sudina" | "niihaara" (* id. vaartika *)
  | "ka.s.ta" | "k.rcchra" (* strive to P{3,1,14} *)
  | "romantha" (* practice P{3,1,15} *)
  | "dhuuma" | "baa.spa" | "phena" (* emit P{3,1,16} *)
  | "kurafga" | "pu.skara" | "yuga" | "vi.sa" | "zizu" | "samudra#1"
  | "gomaya" | "sa.mdhyaa" (* resemble *)
  | "puru.sa" (* imitate *)
  | "k.r.s.na" | "manda" | "bhuusvarga" (* to become *)
      → lengthen rstem (* reflexive causative middle to become P{3,1,13} *)
  | _ → failwith ("Unknown␣denominative␣" ^ entry)
  ]
;
value compute_denom stem ystem entry = do (* other than present system - rare *)
```

```
  { build_perpft Primary ystem entry
  ; let fsuf = revcode "i.sy" in (* rare - similar to compute_future_10 *)
    compute_future (fsuf @ ystem) entry
  ; let perstem = [ 3 :: ystem ] (* -yi *) in
    perif Primary perstem entry
  ; match stem with
    [ [ 1 :: rest ] →
        match entry with
        [ "asuuya" → () (* wrong asya *)
        | "m.rga" → () (* from m.rg *)
        | _ → do (* experimental - rare acc. to Whitney *)
              { compute_passive_11 entry rest
              ; record_pfp_10 entry rest
              }
        ]
    | _ → () (* specially wrong for consonant stems *)
    ]
  }
;
value compute_denominative_a entry third =
  match Word.mirror third with
    [ [ 3 :: [ 32 :: [ 1 :: ([ 42 :: s ] as ystem) ] ] ] (* -yati *) → do
          { compute_thematic_active 11 Primary ystem entry third
          ; compute_denom s ystem entry
          ; record_ppp_abs_den ystem entry
          }
    | _ → failwith ("Anomalous␣denominative␣" ^ Canon.decode third)
    ]
and compute_denominative_m entry third =
  match Word.mirror third with
    [ [ 10 :: [ 32 :: [ 1 :: ([ 42 :: s ] as ystem) ] ] ] (* -yate *) → do
          { compute_thematic_middle 11 Primary ystem entry third
          ; compute_denom s ystem entry
          ; record_ppp_abs_den ystem entry
          }
    | _ → failwith ("Anomalous␣denominative␣" ^ Canon.decode third)
    ]
;
(* We use the lexicalized third stem *)
value compute_denominative entry pada third =
```

```
  match third with
  [ [] (* fake *) → do (* pada not informative, we try both *)
      { try let stem = den_stem_a entry in
              let ystem = [ 42 :: stem ] in do
              { compute_thematic_active 11 Primary ystem entry third
              ; compute_denom stem ystem entry
              ; record_ppp_abs_den ystem entry
              }
          with [ Failure _ → () ]
      ; try let stem = den_stem_m entry in
              let ystem = [ 42 :: stem ] in do
              { compute_thematic_middle 11 Primary ystem entry third
              ; compute_denom stem ystem entry
              ; record_ppp_abs_den ystem entry
              }
          with [ Failure _ → () ]
      }
  | _ → if pada then (* Para *) compute_denominative_a entry third
                    else (* Atma *) compute_denominative_m entry third
  ]
;
(*************************)
(* Main conjugation engine *)
(*************************)
(* compute_conjugs_stems : string → Conj_infos.vmorph → unit *)
(* Called by compute_conjugs and fake_compute_conjugs below *)
(* and Conjugation.secondary_conjugs *)
value compute_conjugs_stems entry (vmorph, aa) = do (* main *)
  { admits_aa.val := aa (* sets the flag for phantom forms for aa- preverb *)
  ; match vmorph with
 [ Conj_infos.Prim 11 pada third →
        (* note: pada of denominative verbs is lexicalized *)
        compute_denominative entry pada third
 | Conj_infos.Prim 10 pada third →
    (* root in gana 10, pada is True for Para, False for Atma of third form *)
    let rstem = revstem entry in (* root stem reversed *)
    try do
    { (* Present system plus perif pft and future, infinitives and pfp-tavya *)
      compute_present_system entry rstem 10 pada third
      (* missing: imperative in -taat Whitney§570-1 (post-vedic rare) *)
```

```
      (* Future and Conditional *)
    ; compute_future_10 rstem entry
      (* Passive *)
    ; let ps_stem = passive_stem entry rstem in
      compute_passive_10 entry (strong ps_stem)
    ; record_pfp_10 entry rstem
      (* Ppp and Absolutives *)
    ; let ystem = rfix rstem "ay"
      and ppstem = rfix rstem "ita" in do
      { record_part_ppp ppstem entry
      ; record_abso_tvaa (fix ystem "itvaa") entry
      ; let ya_stem = if light_10 rstem then ystem else rstem in
        record_abso_ya (fix ya_stem "ya") entry
      }
      (* No Perfect – periphrastic perfect generated by process10 above *)
    }
    with [ Control.Warning s → output_string stdout (s ^ "\n") ]
  | Conj_infos.Prim gana pada third →
    (* gana is root class, pada is True for Para, False for Atma of third form *)
    (* Primary conjugation *)
    let rstem = revstem entry in (* root stem reversed *)
    try do
    { compute_present_system entry rstem gana pada third (* Present system *)
    ; (* Future and Conditional *)
      match entry with
      [ "ifg" | "paz" | "cint" (* d.rz cit *)
      | "bruu" (* vac *)
      | "k.saa" | "cud" | "dhii#1" | "pat#2" | "praa#1" | "vidh#1" | "zlath"
          → () (* no future *)
      | "tud#1" | "cakaas" → () (* only periphrastic *)
      | "bharts" → compute_future_gen rstem entry (* exception gana 10 *)
      | "umbh" → do { compute_future_gen (revcode "ubh") entry (* 2 forms *)
                    ; compute_future_gen rstem entry
                    }
      | "saa#1" → do { compute_future_gen (revcode "si") entry
                     ; compute_future_gen rstem entry
                     }
      | "vyadh" → compute_future_gen (revcode "vidh") entry
      | "zuu" → compute_future_gen (revcode "zve") entry
      | "knuu" → compute_future_gen (revcode "knuuy") entry
```

```
    | _ → compute_future_gen rstem entry
    ]
  ; (* Periphrastic future, Infinitive, Passive future part. in -tavya *)
    match entry with
    [ "ifg" | "paz" (* for d.rz *) | "bruu" (* for vac *)
    | "k.saa" | "cud" | "dhii#1" | "pat#2" | "praa#1" | "vidh#1"
    | "haa#2" → () (* no perif *)
    | "saa#1" → do { compute_perif (revcode "si") entry
                   ; compute_perif rstem entry
                   }
    | "vyadh" → compute_perif (revcode "vidh") entry
    | "zuu" → compute_perif (revcode "zve") entry
    | "knuu" → compute_perif (revcode "knuuy") entry
    | "stambh" → compute_perif (revcode "stabh") entry
    | _ → compute_perif rstem entry
    ]
  ; (* Precative/Benedictive active rare, middle very rare in classical skt *)
     match entry with
     [ "as#1" → () (* uses bhuu *) (* but Zriivara: staat *)
     | _ → compute_benedictive rstem entry
     ]
  ; (* Passive *)
    if admits_passive entry then
       let ps_stem = passive_stem entry rstem in do
       { if entry = "arh" ∨ entry = "k.lp" then () (* admits pfp but no ps *)
         else compute_passive Primary entry ps_stem
         (* Passive future participle (gerundive) in -ya and -aniiya *)
       ; record_pfp entry rstem
       }
    else ()
  ; (* Ppp computation and recording (together with absolutives) *)
    if admits_ppp_abs entry then do
       { let ppstems = compute_ppp_stems entry rstem in
         record_ppp_abs_stems entry rstem ppstems
       ; record_abso_am entry (* rare *)
       }
    else ()
  ; (* Perfect *)
    match entry with
    [ "paz" (* d.rz *) | "bruu" (* vac *) | "ma.mh" (* mah *) | "ind"
```

```
    | "indh" | "inv" | "k.saa" | "cakaas" | "dhii#1" | "vidh#1"
        → () (* no perfect *)
    | "uuh" → () (* periphrastic *)
    | _ → compute_perfect entry
    ] (* NB perfect forms may have a passive meaning *)
  ; (* Periphrastic Perfect *) (* .namul on demand - except gana 10 above *)
    try let stem = peri_perf_stem entry in
          build_perpft Primary stem entry
    with [ Not_attested → () ]
  ; (* Aorist *) compute_aorist entry
  ; (* Injunctive *) compute_injunctive entry
  }
  with [ Control.Warning s → output_string stdout (s ^ "\n") ]
  (* end of Primary conjugation (including passive) *)
| Conj_infos.Causa third →
    (* Here we extract the causative stem from the third given in Dico *)
    (* rather than implementing all special cases of Whitney§1042. *)
    (* Alternative: compute cstem instead of reading it from the lexicon. Voir Panini krit
".ni" P{7,3,36-43} *)
    let (cstem, active) = match Word.mirror third with
        [ [ 3 :: [ 32 :: [ 1 :: st ] ] ] (* remove -ati *)
            → (st, True)
        | [ 10 :: [ 32 :: [ 1 :: st ] ] ] (* remove -ate *)
            → (st, False)
          (* We lose some information, but generate both active and middle *)
        | _ → failwith ("Weird causative " ^ Canon.decode third)
        ] in
    let cpstem = match cstem with
        [ [ 42 :: [ 1 :: st ] ] (* -ay *) → match entry with
            [ "dhvan" → revcode "dhvaan"
            | _ → st
            (* doubt: ambiguity in ps when the ca stem is not lengthened *)
            (* eg gamyate. Whitney§1052a says "causatively strengthened stem"? *)
            ]
        (* Why no ca in -aayati while such forms exist for ga.na 10 and 11 ? *)
        | _ → failwith ("Anomalous causative " ^ Canon.decode third)
        ] in
    let icstem = [ 3 :: cstem ] (* -ayi *) in
    let compute_causative stem = do (* both active and middle are generated *)
        { compute_causativea stem entry (if active then third else [])
```

```
        ; compute_causativem stem entry (if active then [] else third)
        } in
    do (* active, middle, passive present; active middle future, aor *)
    { compute_causative cstem
    ; compute_passive Causative entry cpstem (* adapt compute_passive_10? *)
    ; let fsuf = revcode "i.sy" in
      let fustem = fsuf @ cstem in
      compute_future_ca fustem entry
    ; compute_aor_ca cpstem entry (* Whitney§861b Henry§339 *)
    ; (* Passive future participle in -ya *)
      match entry with
      [ "gad" | "yam" | "has" → () (* to avoid redundancy with Primary pfp *)
      (* zi.s : justified redundancy with Primary pfp *)
      (* car : redundancy with Primary pfp to be justified *)
      | _ → record_pfp_ya Causative cpstem entry
      ]
    ; (* Passive future participle in -aniiya *)
      record_pfp_aniiya Causative cpstem entry
      (* Passive past participle and absolutives *)
    ; record_pppca cpstem cstem entry
      (* Periphrastic future, Infinitive, Gerundive/pfp in -tavya *)
    ; perif Causative icstem entry
      (* Periphrastic perfect Whitney§1045 *)
    ; build_perpft Causative cstem entry (* gamayaa.mcakaara *)
    }
| Conj_infos.Inten third → (* TODO passive, perfect, future, aorist, parts *)
    match Word.mirror third with (* active or middle are generated on demand *)
    (* paras. in -ati, -iiti, -arti (k.r2), -aati (draa1, yaj1), -etti (vid1) *)
    [ [ 3 :: [ 32 :: [ 4 :: ([ 45 :: [ 1 :: w ] ] as wk) ] ] ] (* x-aviiti *) →
        let st = [ 12 :: w ] in
        (* x-o eg for hu johavitti -¿ joho -¿ johomi johavaani *)
        compute_intensivea wk st entry third
    | [ 3 :: [ 32 :: [ 4 :: wk ] ] ] (* other -iiti *) →
        let st = strong wk in
        compute_intensivea wk st entry third
    | [ 3 :: [ 32 :: st ] ] (* ti *)
    | [ 3 :: [ 27 :: st ] ] (* .ti eg veve.s.ti *) →
        let wk = st in (* TEMP - no easy way to get weak stem from strong one *)
                       (* eg vevid from vevetti=veved+ti nenij from nenekti *)
        compute_intensivea wk st entry third
```

```
      | [ 10 :: [ 32 :: [ 1 :: st ] ] ] → (* -ate *)
          compute_intensivem st entry third
      | [ 10 :: [ 32 :: st ] ] → (* -te : nenikte *)
          compute_intensivem2 st entry third
      | _ → failwith ("Weird␣intensive␣" ^ Canon.decode third)
      ]
  | Conj_infos.Desid third → (* TODO passive, future, aorist, more parts *)
      let compute_krid st = do (* ppp pfp inf *)
          { record_pppdes st entry
          ; record_pfp_aniiya Desiderative st entry
          ; record_pfp_ya Desiderative st entry
          ; perif Desiderative [ 3 :: st ] entry
          } in
      match Word.mirror third with (* active or middle are generated on demand *)
        [ [ 3 :: [ 32 :: [ 1 :: st ] ] ] → do
            { compute_desiderativea st entry third
            ; compute_passive Desiderative entry st
            ; compute_futurea Desiderative [ 42 :: st ] entry
            ; compute_perfect_desida st entry
            ; compute_krid st
            }
        | [ 10 :: [ 32 :: [ 1 :: st ] ] ] → do
            { compute_desiderativem st entry third
            ; compute_passive Desiderative entry st
            ; compute_futurem Desiderative [ 42 :: st ] entry
            ; compute_perfect_desidm st entry
            ; compute_krid st
            }
        | _ → failwith ("Weird␣desiderative␣" ^ Canon.decode third)
        ]
  ]
  } (* end main do *)
;
```

Vedic Subjunctive
Various Vedic subjunctives needed for citations Whitney§562
No attempt for full paradigms, only specific attested forms
TODO add paradigms for i a. and aas2 m. Whitney§614

```
value compute_subjunctives () =
  let enter_subjunctivea conj root tin =
```

```
      enter1 root (Conju (conj, Conjug Subjunctive Active) [ tin ])
  and enter_subjunctivem conj root tin =
      enter1 root (Conju (conj, Conjug Subjunctive Middle) [ tin ]) in
  let subj_sg root person form =
    let tin = (Singular, [ (person, code form) ]) in
    enter_subjunctivea Primary root tin
  and subj_pl root person form =
    let tin = (Plural, [ (person, code form) ]) in
    enter_subjunctivea Primary root tin
  and subjm_sg3 root form =
    let tin = (Singular, [ (Third, code form) ]) in
    enter_subjunctivem Primary root tin
  and subj_cau_sg root person form =
    let tin = (Singular, [ (person, code form) ]) in
    enter_subjunctivea Causative root tin
  and subj_int_sg root person form =
    let tin = (Singular, [ (person, code form) ]) in
    enter_subjunctivea Intensive root tin in do
  { subj_sg "zru" Third "zro.sat"
  ; subj_sg "i.s#1" Third "icchaat"
  ; subj_sg "vac" Third "vocati" (* primary endings *)
  ; subj_sg "vac" Third "vocat" (* secondary endings *)
  ; subj_sg "vac" Second "vocas" (* both forms also available as inj *)
  ; subj_sg "pat#1" Third "pataati"
  ; subj_pl "gam" Third "gman" (* for apigman *)
  ; subj_cau_sg "jan" Second "janayaas"
  ; subj_cau_sg "cud" Third "codayaat" (* Gaayatrii pracodayaat *)
  ; subj_int_sg "vi.s#1" Third "vevi.sati"
(*; subj_sg "k.r#1" First "karavaa.ni" (* became imp Whitney§578 *) *)
  ; subjm_sg3 "k.r#1" "k.r.nvate" (* aussi pr5 md *)
  }
;
value compute_auxi_kridantas () =
  let stems str = let st = revstem str in match st with
      [ [ 1 :: rst ] → (rst, Word.mirror st)
      | _ → failwith "auxi_kridantas"
      ] in do (* A few auxiliary action nouns are generative for cvi compounds *)
  { let (rst, st) = stems "kara.na" in
    build_part_a_n (Primary, Action_noun) rst st "k.r#1"
  ; let (rst, st) = stems "kaara" in (* actually, should be Agent_noun *)
```

```
    build_part_a_m (Primary, Action_noun) rst st "k.r#1" (* also fem in -ii? *)
  ; let (rst, st) = stems "bhaavana" in
    build_part_a_m (Primary, Action_noun) rst st "bhuu#1"
  ; let (rst, st) = stems "bhaava" in
    build_part_a_m (Primary, Action_noun) rst st "bhuu#1"
  }
;
(* Called by Make_roots.roots_to_conjugs *)
value compute_conjugs root (infos : Conj_infos.root_infos) =
  let root_entry = Canon.decode root in compute_conjugs_stems root_entry infos
;
(* Supplementary forms *)
value compute_extra_rc () = (* vedic - P{7,1,38} *)
  enter1 ".rc#1" (Absotvaa Primary (code "arcya")) (* abs -ya with no preverb *)
and compute_extra_khan () = (* WR MW *)
  let root = "khan"
  and conj = Primary
  and pstem = revcode "khaa" (* khaa substituted optionally in ps *) in
  compute_passive conj root pstem
and compute_extra_car () = do
  { enter1 "car" (Absotvaa Primary (code "cartvaa"))
  ; enter1 "car" (Absotvaa Primary (code "ciirtvaa"))
  ; enter1 "car" (Invar (Primary, Infi) (code "cartum")) (* epic *)
  }
and compute_extra_jnaa () =
  let entry = "j~naa#1" in (* j napta vet P{7,2,27} *)
  let cstem = revcode "j~nap" in
  let ppstem = [ 1 :: [ 32 :: cstem ] ] (* j napta *) in do
  { record_part (Ppp_ Causative ppstem entry)
  ; record_part (Pppa_ Causative ppstem entry) (* pp-vat *)
  ; perif Causative cstem entry
  }
and compute_extra_tri () = do
      { build_infinitive Primary (revcode "tarii") "t.rr" (* id. *)
      ; build_infinitive Primary (revcode "tar") "t.rr" (* Whitney roots *)
      }
and compute_extra_dhaa () = (* Gaayatrii dhiimahi precative m. Whitney§837b *)
    enter1 "dhaa#1" (Conju benem [ (Plural, [ (First, code "dhiimahi") ]) ])
(* also "vidmahi" on yantra ? *)
and compute_extra_nind () = (* WR: RV *)
```

```
  enter1 "nand" (Conju perfa [ (Plural, [ (Third, code "ninidus") ])
                              ; (Plural, [ (First, code "nindimas") ]) ])
and compute_extra_prr () = (* paaryate as well as puuryate above *)
    let stem = revcode "paar" in compute_passive Primary "p.rr" stem
and compute_extra_bhaas () =
    enter1 "bhaa.s" (Invar (Primary, Infi) (code "bhaa.s.tum")) (* WR epic *)
and compute_extra_bhuj2 () =
    enter1 "hhuj#2" (Conju (Primary, voa 7) (* epics Wh688a *)
                             [ (Singular, [ (First, code "bhu~njiiyaam") ])
                             ; (Singular, [ (Second, code "bhu~njiiyaas") ])
                             ; (Singular, [ (Third, code "bhu~njiiyaat") ])
                             ])
and compute_extra_bhr () = (* Epics sa.mbhriyantu Oberlies 8.7 *)
   enter1 "bh.r" (Conju (Primary, vmp) [ (Plural, [ (Third, code "bhriyantu") ]) ])
and compute_extra_bhram () = (* MW: Mah *)
  enter1 "bhram" (Conju perfa [ (Plural, [ (Third, code "bhremur") ]) ])
and compute_extra_muc () = do
  { (* ved precative 'fasse que je sois libéré' *)
    enter1 "muc#1" (Conju benem [ (Singular, [ (First, code "muk.siiya") ]) ])
  ; build_infinitive Causative (revcode "moci") "muc#1" (* Whitney§1051c *)
  }
and compute_extra_vadh () = (* no present - use "han#1" *)
  let root = "vadh"
  and rstem = revcode "vadh" in do
  { compute_aorist root
  ; compute_future_gen rstem root
  ; compute_passive_raw root
  (* record_pfp root rstem is computed by compute_extra_participles *)
  }
and compute_extra_zaas () =
   let e = "zaas" in do (* epics zaasyate + Renou gram §29 *)
     { let stem = revcode e in compute_passive Primary e stem
     ; enter1 e (Conju (Primary, via 2) [ (Singular, [ (Second, code "azaat") ]) ])
     }
and compute_extra_zru () =
  enter1 "zru" (* ved écoute *)
         (Conju (impera 5) [ (Singular, [ (Second, code "zrudhi") ]) ])
and compute_extra_sanj () = (* WR Oberlies p LI but maybe prm of variant sajj *)
  let root = "sa~nj"
  and conj = Primary
```

```
   and pastem = revcode "sajj" (* "y" replaced by j in passive *) in
   compute_passive_system conj root pastem
and compute_extra_skand () = do (* WR *)
   { enter1 "skand" (Invar (Primary, Infi) (code "skanditum"))
   ; record_abso_ya (code "skadya") "skand"
   }
and compute_extra_syand () = do (* WR *)
   { record_abso_tvaa (code "syattvaa") "syand"
   ; record_abso_ya (code "syadya") "syand"
   }
and compute_extra_hims () = do
   { (* Renou gram §29 *) enter1 "hi.ms"
      (Conju (Primary, via 7) [ (Singular, [ (Second, code "ahi.msat") ]) ])
   ; (* MW *) enter1 "hi.ms"
      (Conju (presa 7) [ (Singular, [ (Second, code "hi.msi") ]) ])
   }
and compute_extra_huu () = do (* WR *)
    { compute_futurem Primary (revstem "hvaasy") "huu"
    ; enter1 "huu" (Invar (Primary, Infi) (code "hvayitum"))
    }
;
(* Extra participial forms - intensive, desiderative, no present, etc *)
value record_extra_participles () = do
   { record_part_ppp (revstem "gupta") "gup" (* gup gana 10 *)
   ; record_part_ppp (revstem "zaata") "zaa"
   ; record_part_ppp (revstem "kaanta") "kam"
   ; record_part_ppp (revstem "k.sita") "k.sii"
   ; record_part_ppp (revstem "diipita") "diip"
   ; record_part_ppp (revstem "spa.s.ta") "spaz#1"
   ; record_part (Ppra_ 1 Intensive (revstem "jaajam") (revstem "jaajamat") "jam")
   ; record_pfp "d.r#1" (revcode "d.r")
   ; record_pfp "vadh" (revcode "vadh")
   ; record_part (Pprm_ 1 Primary (revcode "gacchamaana") "gam")
   ; record_part (Pprm_ 4 Primary (revcode "kaayamaana") "kan")
   ; record_part (Ppra_ 1 Primary (revstem ".dam") (revstem ".damat") ".dam")
   }
;
(* For verbs without present forms and variants, *)
(* called by Make_roots.roots_to_conjugs at generation time *)
value compute_extra () = do
```

```
{ compute_perfect "ah" (* verbs with no present system *)
; compute_aorist "kan"
; compute_perfect "kam"
; compute_perfect "ghas"
; compute_perfect "ta.d"
; compute_perfect "spaz#1"
; compute_aorist "spaz#1"
; compute_aorist "k.r#2"
; compute_passive_raw "d.r#1"
(* Now for specific extra forms *)
; compute_extra_rc ()
; compute_extra_khan ()
; compute_extra_car ()
; compute_extra_jnaa ()
; compute_extra_tri ()
; compute_extra_dhaa ()
; compute_extra_nind ()
; compute_extra_prr ()
; compute_extra_bhaas ()
; compute_extra_bhuj2 ()
; compute_extra_bhr ()
; compute_extra_bhram ()
; compute_extra_muc ()
; compute_extra_vadh ()
; compute_extra_zaas ()
; compute_extra_zru ()
; compute_extra_sanj ()
; compute_extra_skand ()
; compute_extra_syand ()
; compute_extra_hims ()
; compute_extra_huu ()
; build_infinitive Primary (revcode "rami") "ram"
; build_infinitive Primary (revcode "aas") "aas#2" (* Whitney§968d *)
; build_infinitive Causative (revcode "bhaavi") "bhuu#1" (* Whitney§1051c *)
; build_infinitive Causative (revcode "dhaari") "dh.r" (* Whitney§1051c *)
; build_infinitive Causative (revcode "ze.si") "zi.s" (* Whitney§1051c *)
; build_infinitive Causative (revcode "j~naap") "j~naa#1" (* WR epics *)
  (* Infinitives in -as (kasun k.rt) P{3,4,17} *)
; enter1 "s.rp" (Invar (Primary, Infi) (code "s.rpas")) (* vi.s.rpas *)
; enter1 "t.rd" (Invar (Primary, Infi) (code "t.rdas")) (* aat.rdas *)
```

```
  ; let st = revcode "si.saadhayi.s" in (* des of ca of sidh1 *)
    compute_desiderativea st "saadh" []
  ; record_extra_participles ()
  ; compute_participles ()
  ; compute_auxi_kridantas ()
  ; compute_subjunctives ()
  }
;
(* Called by Conjugation.look_up and Morpho_debug.test_conj *)
(* Remark. For the present system only the queried gana is displayed, *)
(* but all forms of other systems are displayed after it. *)
(* It is for the moment impossible to list forms of roots without present. *)
value fake_compute_conjugs (gana : int) (entry : string) = do
  { morpho_gen.val := False (* Do not generate phantom forms *)
  ; let no_third = [] and pada = True in (* hacks to disable check warning *)
    let vmorph = Conj_infos.Prim gana pada no_third in do
    { compute_conjugs_stems entry (vmorph, False) (* False since no-op in fake *)
    ; match entry with (* extra forms - to be completed from compute_extra *)
      [ ".rc#1" → compute_extra_rc ()
      | "k.sii" → record_part_ppp (revcode "k.sita") entry
      | "khan" → compute_extra_khan ()
      | "gup" → record_part_ppp (revcode "gupta") entry
      | "car" → compute_extra_car ()
      | "j~naa#1"→ compute_extra_jnaa ()
      | "t.rr" → compute_extra_tri ()
      | "dhaa#1" → compute_extra_dhaa ()
      | "nind" → compute_extra_nind ()
      | "p.rr" → compute_extra_prr ()
      | "bhaa.s" → compute_extra_bhaas ()
      | "bhuj#2" → compute_extra_bhuj2 ()
      | "bh.r" → compute_extra_bhr ()
      | "bhram" → compute_extra_bhram ()
      | "muc#1" → compute_extra_muc ()
      | "vadh" → compute_extra_vadh ()
      | "zaa" → record_part_ppp (revcode "zaata") entry
      | "zaas" → compute_extra_zaas ()
      | "zru" → compute_extra_zru ()
      | "sa~nj" → compute_extra_sanj ()
      | "skand" → compute_extra_skand ()
      | "spaz#1" → record_part_ppp (revcode "spa.s.ta") entry
```

```
      | "syand" → compute_extra_syand ()
      | "hi.ms" → compute_extra_hims ()
      | "huu" → compute_extra_huu ()
      | _ → ()
      ]
    }
  }
;
```

Module Parts

Computes the declensions of participles from stored stems.

```
open Skt_morph;
open Encode; (* rev_code_string, code_string *)
open Phonetics; (* monosyllabic aug *)
open Inflected; (* enter enter1 enter_form enter_forms access_krid register_krid *)
value mirror = Word.mirror
;
(* Used for storing participial stems in the participles list. *)
(* This structure is essential for fast online computation of verbal forms. *)
(* Beware - the stem argument is a reversed word, the string is the root. *)
type memo_part =
  [ Ppp_ of conjugation and Word.word and string (* Past Passive Part *)
  | Pppa_ of conjugation and Word.word and string (* Past Active Part *)
  | Ppra_ of gana and conjugation and Word.word and Word.word and string (* Present
Active Part *)
  | Pprared_ of conjugation and Word.word and string (* idem reduplicated *)
  | Pprm_ of gana and conjugation and Word.word and string (* Present Middle Part *)
  | Pprp_ of conjugation and Word.word and string (* Present Passive Part *)
  | Ppfta_ of conjugation and Word.word and string (* Perfect Active Part *)
  | Ppftm_ of conjugation and Word.word and string (* Perfect Middle Part *)
  | Pfutm_ of conjugation and Word.word and string (* Future Middle Part *)
  | Pfuta_ of conjugation and Word.word and string (* Future Active Part *)
  | Pfutp_ of conjugation and Word.word and string (* Future Passive Part *)
  ]
;
(* Special gana values for present forms of secondary conjugations *)
value cau_gana = 12
and des_gana = 13
```

```
and int_gana = 14
;
(* This is to avoid redundant generation of present system participles when stems may come
from a distinct gana. *)
value redundant_gana k = fun
  [ "svap" → k = 1
  | "rud#1" → k = 6
  | _ → False
  ]
;
(* Affixing a suffix to a (reversed) stem *)
(* fix : Word.word → string → Word.word *)
value fix revstem suff =
  Int_sandhi.int_sandhi revstem (code_string suff)
;
value rfix revstem suff = mirror (fix revstem suff)
;
value fix_augment revstem suff = aug (fix revstem suff)
;
(* NB. Internal sandhi will take care of consonant elision in e.g. ppp "tak.s" = "tak.s"+"ta"="ta.s.t
idem for cak.s tvak.s Pan8,2,29 *)
```

Generation of unique names for kridantas, specially participial stems

```
value gensym stem n =
  if n = 0 then stem
  else mirror [ (n + 50) :: mirror stem ]
;
(* We look up in the kridantas database if the given stem has been registered (possibly with
some homo index) for the same (verbal,root). If not, we generate the name affixing to stem
the next available homo *)
value gen_stem (v, root) stem = (* stem is a bare stem with no homo index *)
  if morpho_gen.val then
     let etym = (v, code_string root) in
     let alist = access_krid stem in
     try gensym stem (List.assoc etym alist) with
       [ Not_found → match alist with
          [ [] → (* no current homonym of stem *) do
            { register_krid stem (etym, 0)
            ; stem
            }
```

```
        | [ (_, n) :: _ ] → (* last homonym entered stem_n *)
          let p = n + 1 in
          if p > 9 then failwith "Gensym␣exceeds␣homo␣index"
          else do { register_krid stem (etym, p); gensym stem p}
        ]
      ]
  else stem
;
(* Now for participle forming paradigms *)
```

Similar to *Nouns.build_mas_at* [1 :: *stem*] if vat=False and to *Nouns.build_mas_mat stem* if vat=True

```
value build_part_at_m vat verbal stem stem_at root = (* invoked by Ppra_ *)
  let gen_entry = gen_stem (verbal, root) stem_at in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff) in
  enter_forms gen_entry
   [ Declined krid Mas
   [ (Singular,
         [ decline Voc "an"
         ; decline Nom (if vat then "aan" else "an")
         ; decline Acc "antam"
         ; decline Ins "ataa"
         ; decline Dat "ate"
         ; decline Abl "atas"
         ; decline Gen "atas"
         ; decline Loc "ati"
         ])
   ; (Dual,
         [ decline Voc "antau"
         ; decline Nom "antau"
         ; decline Acc "antau"
         ; decline Ins "adbhyaam"
         ; decline Dat "adbhyaam"
         ; decline Abl "adbhyaam"
         ; decline Gen "atos"
         ; decline Loc "atos"
         ])
   ; (Plural,
         [ decline Voc "antas"
```

```
          ; decline Nom "antas"
          ; decline Acc "atas"
          ; decline Ins "adbhis"
          ; decline Dat "adbhyas"
          ; decline Abl "adbhyas"
          ; decline Gen "ataam"
          ; decline Loc "atsu"
          ])
    ]
    ; Bare krid stem_at (* e.g. b.rhadazva *)
    ]
;
(* Similar to Nouns.build_mas_red *)
value build_part_at_m_red verbal stem stem_at root =
  let gen_entry = gen_stem (verbal, root) stem_at in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff) in
  enter_forms gen_entry
    [ Declined krid Mas
    [ (Singular,
          [ decline Voc "at"
          ; decline Nom "at"
          ; decline Acc "atam"
          ; decline Ins "ataa"
          ; decline Dat "ate"
          ; decline Abl "atas"
          ; decline Gen "atas"
          ; decline Loc "ati"
          ])
    ; (Dual,
          [ decline Voc "atau"
          ; decline Nom "atau"
          ; decline Acc "atau"
          ; decline Ins "adbhyaam"
          ; decline Dat "adbhyaam"
          ; decline Abl "adbhyaam"
          ; decline Gen "atos"
          ; decline Loc "atos"
          ])
    ; (Plural,
```

```
        [ decline Voc "atas"
        ; decline Nom "atas"
        ; decline Acc "atas"
        ; decline Ins "adbhis"
        ; decline Dat "adbhyas"
        ; decline Abl "adbhyas"
        ; decline Gen "ataam"
        ; decline Loc "atsu"
        ])
    ]
  ; Bare krid stem_at
  ]
;
(* Similar to Nouns.build_neu_at *)
value build_part_at_n verbal stem stem_at root =
  let gen_entry = gen_stem (verbal, root) stem_at in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff) in
  enter_forms gen_entry
   [ Declined krid Neu
   [ (Singular,
        [ decline Voc "at"
        ; decline Nom "at"
        ; decline Acc "at"
        ; decline Ins "ataa"
        ; decline Dat "ate"
        ; decline Abl "atas"
        ; decline Gen "atas"
        ; decline Loc "ati"
        ])
   ; (Dual,
        [ decline Voc "atii"
        ; decline Voc "antii"
        ; decline Nom "atii"
        ; decline Nom "antii"
        ; decline Acc "atii"
        ; decline Acc "antii"
        ; decline Ins "adbhyaam"
        ; decline Dat "adbhyaam"
        ; decline Abl "adbhyaam"
```

```
        ; decline Gen "atos"
        ; decline Loc "atos"
        ])
   ; (Plural,
        [ decline Voc "anti"
        ; decline Nom "anti"
        ; decline Acc "anti"
        ; decline Ins "adbhis"
        ; decline Dat "adbhyas"
        ; decline Abl "adbhyas"
        ; decline Gen "ataam"
        ; decline Loc "atsu"
        ])
   ]
   ; Bare krid stem_at
   ]
;
(* Similar to Nouns.build_neu_red *)
value build_part_at_n_red verbal stem stem_at root =
  let gen_entry = gen_stem (verbal, root) stem_at in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff) in
  enter_forms gen_entry
   [ Declined krid Neu
   [ (Singular,
        [ decline Voc "at"
        ; decline Nom "at"
        ; decline Acc "atam"
        ; decline Ins "ataa"
        ; decline Dat "ate"
        ; decline Abl "atas"
        ; decline Gen "atas"
        ; decline Loc "ati"
        ])
   ; (Dual,
        [ decline Voc "atii"
        ; decline Nom "atii"
        ; decline Acc "atii"
        ; decline Ins "adbhyaam"
        ; decline Dat "adbhyaam"
```

```
          ; decline Abl "adbhyaam"
          ; decline Gen "atos"
          ; decline Loc "atos"
          ])
    ; (Plural,
          [ decline Voc "ati"
          ; decline Voc "anti"
          ; decline Nom "ati"
          ; decline Nom "anti"
          ; decline Acc "ati"
          ; decline Acc "anti"
          ; decline Ins "adbhis"
          ; decline Dat "adbhyas"
          ; decline Abl "adbhyas"
          ; decline Gen "ataam"
          ; decline Loc "atsu"
          ])
    ]
    ; Bare krid stem_at
    ]
;
(* Similar to Nouns.build_fem_ii *)
value build_part_ii verbal stem prati root =
  let stem_ii = mirror [ 4 :: stem ] in
  let gen_entry = gen_stem (verbal, root) prati in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff) in
  enter_forms gen_entry
    [ Declined krid Fem
    [ (Singular,
          [ decline Voc "i"
          ; decline Nom "ii"
          ; decline Acc "iim"
          ; decline Ins "yaa"
          ; decline Dat "yai"
          ; decline Abl "yaas"
          ; decline Gen "yaas"
          ; decline Loc "yaam"
          ])
    ; (Dual,
```

```
          [ decline Voc "yau"
          ; decline Nom "yau"
          ; decline Acc "yau"
          ; decline Ins "iibhyaam"
          ; decline Dat "iibhyaam"
          ; decline Abl "iibhyaam"
          ; decline Gen "yos"
          ; decline Loc "yos"
          ])
    ; (Plural,
          [ decline Voc "yas"
          ; decline Nom "yas"
          ; decline Acc "iis"
          ; decline Ins "iibhis"
          ; decline Dat "iibhyas"
          ; decline Abl "iibhyas"
          ; decline Gen "iinaam"
          ; decline Loc "ii.su"
          ])
    ]
    ; Bare krid stem_ii (* productive ? *)
    ]
;
(* Similar to Nouns.build_mas_a *)
value build_part_a_m verbal stem prati root =
  let gen_entry = gen_stem (verbal, root) prati in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff) in
  enter_forms gen_entry
    [ Declined krid Mas
    [ (Singular,
          [ decline Voc "a"
          ; decline Nom "as"
          ; decline Acc "am"
          ; decline Ins "ena"
          ; decline Dat "aaya"
          ; decline Abl "aat"
          ; decline Gen "asya"
          ; decline Loc "e"
          ])
```

```
    ; (Dual,
          [ decline Voc "au"
          ; decline Nom "au"
          ; decline Acc "au"
          ; decline Ins "aabhyaam"
          ; decline Dat "aabhyaam"
          ; decline Abl "aabhyaam"
          ; decline Gen "ayos"
          ; decline Loc "ayos"
          ])
    ; (Plural,
          [ decline Voc "aas"
          ; decline Nom "aas"
          ; decline Acc "aan"
          ; decline Ins "ais"
          ; decline Dat "ebhyas"
          ; decline Abl "ebhyas"
          ; decline Gen "aanaam"
          ; decline Loc "esu"
          ])
    ]
    ; Bare krid prati
    (* what follows needs adapting Inflected.enter_form ; Avyayaf (fix stem "am") (* yathaav.rddham
*) possible Cvi usage: see Nouns.iiv_krids *)
    ]
;
(* Similar to Nouns.build_neu_a *)
value build_part_a_n verbal stem prati root =
  let gen_entry = gen_stem (verbal, root) prati in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff) in
  enter_forms gen_entry
    [ Declined krid Neu
    [ (Singular,
          [ decline Voc "a"
         (* decline Voc "am" - rare - disconnected for avoiding overgeneration *)
          ; decline Nom "am"
          ; decline Acc "am"
          ; decline Ins "ena"
          ; decline Dat "aaya"
```

```
          ; decline Abl "aat"
          ; decline Gen "asya"
          ; decline Loc "e"
          ])
    ; (Dual,
          [ decline Voc "e"
          ; decline Nom "e"
          ; decline Acc "e"
          ; decline Ins "aabhyaam"
          ; decline Dat "aabhyaam"
          ; decline Abl "aabhyaam"
          ; decline Gen "ayos"
          ; decline Loc "ayos"
          ])
    ; (Plural,
          [ decline Voc "aani"
          ; decline Nom "aani"
          ; decline Acc "aani"
          ; decline Ins "ais"
          ; decline Dat "ebhyas"
          ; decline Abl "ebhyas"
          ; decline Gen "aanaam"
          ; decline Loc "esu"
          ])
    ]
    ; Bare krid prati
    ]
;
(* Similar to Nouns.build_fem_aa *)
value build_part_aa verbal stem prati root =
  let gen_entry = gen_stem (verbal, root) prati in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff) in
  enter_forms gen_entry
    [ Declined krid Fem
    [ (Singular,
          [ decline Voc "e"
          ; decline Nom "aa"
          ; decline Acc "aam"
          ; decline Ins "ayaa"
```

```
          ; decline Dat "aayai"
          ; decline Abl "aayaas"
          ; decline Gen "aayaas"
          ; decline Loc "aayaam"
          ])
    ; (Dual,
          [ decline Voc "e"
          ; decline Nom "e"
          ; decline Acc "e"
          ; decline Ins "aabhyaam"
          ; decline Dat "aabhyaam"
          ; decline Abl "aabhyaam"
          ; decline Gen "ayos"
          ; decline Loc "ayos"
          ])
    ; (Plural,
          [ decline Voc "aas"
          ; decline Nom "aas"
          ; decline Acc "aas"
          ; decline Ins "aabhis"
          ; decline Dat "aabhyas"
          ; decline Abl "aabhyas"
          ; decline Gen "aanaam"
          ; decline Loc "aasu"
          ])
    ] ]
;
(* Similar to Nouns.build_mas_vas *)
(* Except for proper intercalation of i *)
value build_mas_ppfa verbal stem inter stem_vas root =
  let gen_entry = gen_stem (verbal, root) stem_vas in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff)
  and declinev case suff = (case, fix stem suffi) where
      suffi = if inter then "i" ^ suff else suff in
  enter_forms gen_entry
    [ Declined krid Mas
    [ (Singular,
          [ declinev Voc "van"
          ; declinev Nom "vaan"
```

```
          ; declinev Acc "vaa.msam"
          ; decline Ins "u.saa"
          ; decline Dat "u.se"
          ; decline Abl "u.sas"
          ; decline Gen "u.sas"
          ; decline Loc "u.si"
          ])
    ; (Dual,
          [ declinev Voc "vaa.msau"
          ; declinev Nom "vaa.msau"
          ; declinev Acc "vaa.msau"
          ; declinev Ins "vadbhyaam"
          ; declinev Dat "vadbhyaam"
          ; declinev Abl "vadbhyaam"
          ; decline Gen "u.sos"
          ; decline Loc "u.sos"
          ])
    ; (Plural,
          [ declinev Voc "vaa.msas"
          ; declinev Nom "vaa.msas"
          ; decline Acc "u.sas"
          ; declinev Ins "vadbhis"
          ; declinev Dat "vadbhyas"
          ; declinev Abl "vadbhyas"
          ; decline Gen "u.saam"
          ; declinev Loc "vatsu"
          ])
    ]
    ; Bare krid (fix stem "vat") (* eg vidvat- *)
(* ; Avyayaf (fix stem "vas") - Not dealt with by Inflected.enter_form *)
    ]
;
(* Similar to Nouns.build_neu_vas *)
value build_neu_ppfa verbal stem inter stem_vas root =
  let gen_entry = gen_stem (verbal, root) stem_vas in
  let krid = Krid verbal root in
  let decline case suff = (case, fix stem suff)
  and declinev case suff = (case, fix stem suffi) where
        suffi = if inter then "i" ^ suff else suff in
  enter_forms gen_entry
```

```
[ Declined krid Neu
[ (Singular,
      [ declinev Voc "vat"
      ; declinev Nom "vat"
      ; declinev Acc "vat"
      ; decline Ins "u.saa"
      ; decline Dat "u.se"
      ; decline Abl "u.sas"
      ; decline Gen "u.sas"
      ; decline Loc "u.si"
      ])
; (Dual,
      [ decline Voc "u.sii"
      ; decline Nom "u.sii"
      ; decline Acc "u.sii"
      ; declinev Ins "vadbhyaam"
      ; declinev Dat "vadbhyaam"
      ; declinev Abl "vadbhyaam"
      ; decline Gen "u.sos"
      ; decline Loc "u.sos"
      ])
; (Plural,
      [ declinev Voc "vaa.msi"
      ; declinev Nom "vaa.msi"
      ; declinev Acc "vaa.msi"
      ; declinev Ins "vadbhis"
      ; declinev Dat "vadbhyas"
      ; declinev Abl "vadbhyas"
      ; decline Gen "u.saam"
      ; declinev Loc "vatsu"
      ])
]
; Bare krid (fix stem "vat") (* eg vidvat- *)
(* ; Avyayaf (fix stem "vas") - Not dealt with by Inflected.enter_form *)
]
;
(* Supplementary forms with intercalation of i in later language Whitney§805b *)
value build_late_ppfa verbal stem stem_vas root =
  let gen_entry = gen_stem (verbal, root) stem_vas in
  let krid = Krid verbal root in
```

```
let declinev case suff = (case, fix stem ("i" ^ suff)) in do
 { enter_forms gen_entry
 [ Declined krid Mas
 [ (Singular,
       [ declinev Voc "van"
       ; declinev Nom "vaan"
       ; declinev Acc "vaa.msam"
       ])
 ; (Dual,
       [ declinev Voc "vaa.msau"
       ; declinev Nom "vaa.msau"
       ; declinev Acc "vaa.msau"
       ; declinev Ins "vadbhyaam"
       ; declinev Dat "vadbhyaam"
       ; declinev Abl "vadbhyaam"
       ])
 ; (Plural,
       [ declinev Voc "vaa.msas"
       ; declinev Nom "vaa.msas"
       ; declinev Ins "vadbhis"
       ; declinev Dat "vadbhyas"
       ; declinev Abl "vadbhyas"
       ; declinev Loc "vatsu"
       ])
 ]
 ; Declined krid Neu
 [ (Singular,
       [ declinev Voc "vat"
       ; declinev Nom "vat"
       ; declinev Acc "vat"
       ])
 ; (Dual,
       [ declinev Ins "vadbhyaam"
       ; declinev Dat "vadbhyaam"
       ; declinev Abl "vadbhyaam"
       ])
 ; (Plural,
       [ declinev Voc "vaa.msi"
       ; declinev Nom "vaa.msi"
       ; declinev Acc "vaa.msi"
```

```
        ; declinev Ins "vadbhis"
        ; declinev Dat "vadbhyas"
        ; declinev Abl "vadbhyas"
        ; declinev Loc "vatsu"
        ])
    ]
    ; Bare krid (fix stem "vat")
    ]
    }
;
value build_part_a part_kind stem root =
  let prati = mirror [ 1 :: stem ] in do
  { build_part_a_m part_kind stem prati root
  ; build_part_a_n part_kind stem prati root
  ; build_part_aa part_kind stem prati root
  }
and build_part_at part_kind stem stemf root =
  let prati = fix stem "at" in do (* Ppra_ *)
  { build_part_at_m False part_kind stem prati root
  ; build_part_at_n part_kind stem prati root
  ; build_part_ii part_kind stemf prati root
  }
and build_part_at_red part_kind stem stemf root =
  let prati = mirror [ 32 :: [ 1 :: stem ] ] in do (* Pprared_ *)
  { build_part_at_m_red part_kind stem prati root
  ; build_part_at_n_red part_kind stem prati root
  ; build_part_ii part_kind stemf prati root
  }
and build_part_vat part_kind stem stemf root =
  let prati = mirror [ 32 :: [ 1 :: stem ] ] in do
  { build_part_at_m True part_kind stem prati root
  ; build_part_at_n part_kind stem prati root
  ; build_part_ii part_kind stemf prati root
  }
and build_part_vas c stem inter stemf root =
    let prati = fix stem (if inter then "ivas" else "vas")
    and verbal = (c, Ppfta) in do
  { build_mas_ppfa verbal stem inter prati root (* (i)vas *)
  ; build_neu_ppfa verbal stem inter prati root (* (i)vas *)
  ; if (root ="d.rz#1" ∨ root ="vid#2" ∨ root ="viz#1") ∧ c = Primary
```

```
    then build_late_ppfa verbal stem prati root (* i supplement Whitney§805b *)
    else ()
  ; build_part_ii verbal stemf prati root (* u.sii *)
  }
;
(* Participle stems are stored here by calls in Verbs to record_part below; *)
(* this is necessary for the conjugation cgi to display participle stems *)
(* That is, the internal morphology generation is done in a first pass generating kridanta stems. The stems are declined in a second pass, reading from the participles list. This data structure holds just the lemmas of kridanta stems corresponding to one root. Then compute_participles invoked from Verbs.compute_conjugs declines the stems to fill in the morphology data banks for each root. This mechanism is also used by the conjugation engine, in order to display the kridanta stems associated to the argument root. Thus participles is always a short list just used as a stack and not searched, so no need of sophisticated data structure. Of course it is bigger at generation time. *)
value participles = ref ([] : list memo_part)
and participles_aa = ref ([] : list memo_part)
(* the second case is to provide extra phantom forms for kridantas of roots accepting aa- as a preverb, in order to recognize eg meghairaacchannaa.h *)
;
value record_part memo = (* called from Verbs *)
  (* This different treatment in full generation and in single evocation by Conjugation between aa-prefixable roots and others is terrible *)
  (* The structure of Conj_infos is wrong and should be a pair of all admissible preverbs and of an a-list of conjugation patterns *)
  if morpho_gen.val then (* generation of forms: phantom phonemes matter *)
     if admits_aa.val then (* ugly distinction *)
          participles_aa.val := List2.union1 memo participles_aa.val
     else participles.val := List2.union1 memo participles.val
  else participles.val := List2.union1 memo participles.val (* Conjugation *)
;
(* Called by compute_participles *)
value build_part = fun
  [ Ppp_ c stem root → match stem with
    [ [ 1 :: r ] → build_part_a (c, Ppp) r root
    | _ → failwith ("Weird␣Ppp:␣" ^ Canon.rdecode stem)
    ]
  | Pfutp_ c stem root → (* k below ought to be carried by Pfutp_ *)
    match stem with
    [ [ 1 :: r ] →
```

```
        let k = match r with
                [ [ 42 :: [ 45 :: [ 1 :: [ 32 :: _ ] ] ] ] → 3 (* -tavya *)
                | [ 42 :: [ 4 :: _ ] ] → 2 (* -aniiya *)
                | [ 42 :: _ ] → 1 (* -ya *) (* ambiguité possible avec -iiya ? *)
                | _ → failwith ("Weird␣Pfp:␣" ^ Canon.rdecode stem)
                ] in
        build_part_a (c, Pfutp k) r root
      | _ → failwith ("Weird␣Pfp:␣" ^ Canon.rdecode stem)
      ]
  | Pppa_ c ppstem root →
      let m_stem = [ 45 :: ppstem ] (* pp-v *) in
      let f_stem = rfix m_stem "at" (* vatii *) in
      build_part_vat (c, Pppa) m_stem f_stem root
  | Ppra_ k c m_stem f_stem root →
      if redundant_gana k root then ()
      else build_part_at (c, Ppra k) m_stem f_stem root
  | Pprared_ c stem root →
      let k = if c = Intensive then int_gana else 3 in
      let f_stem = rfix stem "at" (* atii *) in
      build_part_at_red (c, Ppra k) stem f_stem root
  | Pprm_ k c stem root → build_part_a (c, Pprm k) stem root
  | Pprp_ c stem root → build_part_a (c, Pprp) stem root
  | Ppfta_ c stem root →
      let inter = if monosyllabic stem then (* intercalating i *)
                      if root ="vid#1" then False
                         (* vid#1 stem=vid vid#2 stem=vivid *)
                      else True
                  else False
      and f_stem = rfix stem "u.s" (* u.sii *) in
      build_part_vas c stem inter f_stem root
  | Ppftm_ c stem root → build_part_a (c, Ppftm) stem root
  | Pfuta_ c stem root →
      let f_stem = rfix stem "ant" (* antii *) in
      build_part_at (c, Pfuta) stem f_stem root
  | Pfutm_ c stem root → build_part_a (c, Pfutm) stem root
  ]
;
```

(∗ Called by *Verbs.compute_conjugs*, in order to create *Install.parts_file* globally for all roots by *Make_roots.make_roots*. It is also invoked by *Conjugation.look_up_and_display* through *Verbs.fake_compute_conjugs*. ∗)

```
value compute_participles () = do
  { List.iter build_part participles.val
  (* Now for roots admitting preverb aa *)
  ; admits_aa.val := True (* triggering phantom generation - ugly global *)
  ; List.iter build_part participles_aa.val
  }
;
```

Interface for module Conj_infos

NB no module value, *Conj_infos* is a purely defining types signature

```
type vmorph =
  [ Prim of int and bool and Word.word (* primary conjugation *)
          (* gana pada form of present 3rd sg for checking *)
          (* pada=True Paradmaipada pada=False AAtmanepada *)
  | Causa of Word.word (* causative 3rd sg form *)
  | Inten of Word.word (* intensive 3rd sg form *)
  | Desid of Word.word (* desiderative 3rd sg form *)
  ]
;
type root_infos = (vmorph × bool) (* True means root admits preverb aa- *)
; (* NB could be (list vmorph * bool) for better factorisation *)
```

Module Automaton

ZZZ special version of Automata, for use for the cache - should disappear

```
open Canon; (* decode rdecode *)
open Phonetics;
open Auto.Auto; (* rule auto stack *)
open Deco;
```

Generalises the structure of trie, seen as a representation of deterministic automaton (recognizer for prefix-shared set of strings), into the graph of a non-deterministic automaton, chaining external sandhi with recognition of inflected forms from the inflected lexicon.
Algorithm. For every inflected form f, and for every external sandhi rule $r: u \mid v \rightarrow w$ such that $f = x.u$, construct a choice point from state $S\ x$ to an iterating block B(r). $S\ x$ is the state reachable from the initial state (top of the trie) by input x, going on the deterministic subgraph, copy of the trie. The set of iterating blocks pertaining to a node are grouped in

a list of non-deterministic choice points.
Parser operation. The parser traverses the state tree while scanning the input. Assume it is at state $S\ x$ looking at input z. It has the choice of either staying in the deterministic part (word lookup) by going to the deterministic transition corresponding to the first symbol in z, with no output, or else choosing in the non-deterministic part a choice block B(r) as an epsilon move (no scanning of z), and then, with $r:\ u \mid v\ \rightarrow\ w$, recognize that w is a prefix of z (scan it or else backtrack), emit the parse trace $< f > -r-$ where $f = inflected(x.u)$, and iterate by jumping to state $S\ v$ (we assume that sandhi rules are stripped so that $S\ v$ always exists). A stack of $(choices, input\_index)$ permits to backtrack on input failure. The final sandhi rules $u \mid \#\ \rightarrow\ y$ are treated similarly, with # matching end of input, but instead of jumping we accept and propose the parse trace as a legal tagging of the sentence (with possible continuation into backtracking for additional solutions). On backtracking a stack of failed attempts may be kept, in order to restart gracefully when a word is missing from the lexicon. This robustification will be essential to turn the parser into a bootstrapping lexicon acquisition device.

Construction of the automaton.

Remark that it is linear in one bottom-up traversal of the inflected trie.

```
type rules = array stack
;
```

(∗ A sandhi entry is a list $[l1;\ l2;\ ...\ ln]$ with $li = [si1;\ si2;\ ...\ sini]$ ∗)
(∗ with $sij = (c1, c2, c3)$ $where\ c1 = code\ w,\ c2 = rev\ (code\ u),\ c3 = code\ v$ ∗)
(∗ such that $u \mid v\ \rightarrow\ w$ by external sandhi, with $i = |u|] \times)(\times$ $[sandhis]$ $concerns\ u\ ended\ by\ s \vee .h$, and $i = 1 \vee 2 \times)(\times$ $[sandhir]$ $concerns\ u\ ended\ by\ r$, and $i = 1 \vee 2 \times)(\times$ $[sandhin]$ $concerns\ u\ ended\ by\ n$, and $i = 1 \vee 2 \times)(\times$ $[sandhif]$ $concerns\ u\ ended\ by\ f$, and $i = 1 \vee 2 \times)(\times$ $[sandhio]$ $concerns\ u\ ended\ by\ other\ letters$, and $i = 1 \times)$

We read sandhi rules compiled by *compile_sandhi*

```
value (sandhis, sandhir, sandhin, sandhif, sandhio) =
  (Gen.gobble Data.sandhis_file : (rules × rules × rules × rules × rules))
;
value get_sandhi = fun (* argument is mirror (code u) *)
  [ [] → failwith "get_sandhi␣0"
  | [ 43 (* r *) :: before ] → match before with
        [ [] → failwith "get_sandhi␣1"
        | [ penu :: _ ] → sandhir.(penu)
        ]
  | [ 48 (* s *) :: before ]
  | [ 16 (* .h *) :: before ] → match before with
        [ [] → failwith "get_sandhi␣2"
        | [ penu :: _ ] → sandhis.(penu)
```

```
          ]
  | [ 36 (* n *) :: before ] → match before with
          [ [] → failwith "get_sandhi␣3"
          | [ penu :: _ ] → sandhin.(penu)
          ]
  | [ 21 (* f *) :: before ] → match before with
          [ [] → failwith "get_sandhi␣4"
          | [ penu :: _ ] → sandhif.(penu)
          ]
  | [ c :: _ ] → if c < 0 then failwith "get_sandhi␣5"
                    else if c > 49 then failwith "get_sandhi␣6"
                    else sandhio.(c)
  ]
;
(* Same as Compile_sandhi.merge *)
value rec merge st1 st2 = match st1 with
  [ [] → st2
  | [ l1 :: r1 ] → match st2 with
      [ [] → st1
      | [ l2 :: r2 ] → [ (List2.union l1 l2) :: (merge r1 r2) ]
      ]
  ]
;
(* We add to the stack arrays a deco rewrite set *)
```

A rewrite deco maps revu to a list of rules (w,revu,v)

```
type rewrite_set = Deco.deco rule
;
value project n = fun
  [ Deco (_, arcs) → try List.assoc n arcs
                      with [ Not_found → empty ] ]
and get_rules = fun
  [ Deco (rules, _) → rules ]
;
(* Union of two decos *)
value rec merger d1 d2 = match d1 with
    [ Deco (i1, l1) → match d2 with
    [ Deco (i2, l2) → Deco (i1 @ i2, mrec l1 l2)
  where rec mrec l1 l2 = match l1 with
    [ [] → l2
```

```
    | [ (n, d) :: l ] → match l2 with
    [ [] → l1
    | [ (n', d') :: l' ] → if n < n' then [ (n, d) :: mrec l l2 ]
                             else if n' < n then [ (n', d') :: mrec l1 l' ]
                                  else [ (n, merger d d') :: mrec l l' ]
    ] ] ] ]
;
```

Automaton construction with state minimization.

```
value hash_max = 9689 (* Mersenne 21 *)
;
exception Overlap
;
module Auto = Share.Share (struct type domain = auto; value size = hash_max; end)
;
(* Remark - it would be incorrect to share states State (b, d, nd) having the same b and d,
since nd may depend on upper nodes because of contextual rules. *)
value hash0 = 1
and hash1 letter key sum = sum + letter × key
and hash b arcs rules = (* NB. abs needed because possible integer overflow *)
    (abs (arcs + Gen.dirac b + List.length rules)) mod hash_max
;
value build_auto (rewrite : rewrite_set) = traverse
  (* traverse : word → lexicon → (auto × stack × rewrite_set × int) *)
  (* The occurrence list occ is the reverse of the access word. *)
  where rec traverse occ = fun
  [ Trie.Trie (b, arcs) →
      let local_stack = if b then get_sandhi occ else []
      and local_rewrite = if b then rewrite else empty in
      let f (deter, stack, rewrite, span) (n, t) =
          let current = [ n :: occ ] in (* current occurrence *)
          let (auto, st, rew, k) = traverse current t in
          ([ (n, auto) :: deter ], merge st stack,
            merger (project n rew) rewrite, hash1 n k span) in
      let (deter, stack, rewrite, span) =
          List.fold_left f ([], [], local_rewrite, hash0) arcs in
      let (h, l) = match stack with
          [ [] → ([], []) | [ h :: l ] → (h, l) ] in
            (* the tail l of stack initialises the stack for upper nodes, its head h contains
the list of current choice points *)
```

```
    let key = hash b span h in
    let s = Auto.share (State (b, List.rev deter, get_rules rewrite @ h)) key in
    (s, merge local_stack l, rewrite, key)
  ]
;
```

(∗ *** IMPORTANT *** The arcs in deter are in decreasing order, because of *fold_left*. We put them back in increasing order by *List.rev deter*. This is not strictly needed, and order of siblings is not important since access is done with *assoc*. However, it is crucial to maintain proper order for operations such as *split*, which splits an automaton into vowel-initial and consonant-initial subparts. Thus reversal was enforced when split was introduced in V2.43. ∗)

Compile builds a tagging transducer from a lexicon index.
compile : *bool* → *rewrites* → *Trie.trie* → *Auto.auto*

```
value compile rewrite lexicon =
  let (transducer, stack, _, _) = build_auto rewrite [] lexicon in
  match stack with
  [ [] → transducer
  | _ → (* Error: some sandhi rule has action beyond one word in the lexicon *)
        raise Overlap
  ]
;
```

Module Automata

same as Platform's Automaton, except debugging tools, and taking the sandhi rules locally from Data rather than dynamically from Web

```
open Canon; (* decode rdecode *)
open Phonetics;
open Auto.Auto; (* rule auto stack *)
open Deco;
```

Generalises the structure of trie, seen as a representation of deterministic automaton (recognizer for prefix-shared set of strings), into the graph of a non-deterministic automaton, chaining external sandhi with recognition of inflected forms from the inflected lexicon.
Algorithm. For every inflected form f, and for every external sandhi rule $r:\ u \mid v \ \rightarrow\ w$ such that $f = x.u$, construct a choice point from state $S\ x$ to an iterating block B(r). $S\ x$ is the state reachable from the initial state (top of the trie) by input x, going on the deterministic subgraph, copy of the trie. The set of iterating blocks pertaining to a node are grouped in

a list of non-deterministic choice points.
Parser operation. The parser traverses the state tree while scanning the input. Assume it is at state $S\ x$ looking at input z. It has the choice of either staying in the deterministic part (word lookup) by going to the deterministic transition corresponding to the first symbol in z, with no output, or else choosing in the non-deterministic part a choice block B(r) as an epsilon move (no scanning of z), and then, with $r:\ u \mid v\ \rightarrow\ w$, recognize that w is a prefix of z (scan it or else backtrack), emit the parse trace $< f > -r-$ where $f = \mathit{inflected}(x.u)$, and iterate by jumping to state $S\ v$ (we assume that sandhi rules are stripped so that $S\ v$ always exists). A stack of $(\mathit{choices}, \mathit{input\_index})$ permits to backtrack on input failure. The final sandhi rules $u \mid \#\ \rightarrow\ y$ are treated similarly, with # matching end of input, but instead of jumping we accept and propose the parse trace as a legal tagging of the sentence (with possible continuation into backtracking for additional solutions). On backtracking a stack of failed attempts may be kept, in order to restart gracefully when a word is missing from the lexicon. This robustification will be essential to turn the parser into a bootstrapping lexicon acquisition device.

Construction of the automaton.

Remark that it is linear in one bottom-up traversal of the inflected trie.

```
type rules = array stack
;
(* A sandhi entry is a list [l1; l2; ... ln] with li = [si1; si2; ... sini] *)
(* with sij = (c1, c2, c3) where c1 = code w, c2 = rev (code u), c3 = code v *)
(* such that u | v → w by external sandhi, with i = |u|] ×)(× [sandhis] concerns u ended by s ∨
 .h, and i = 1 ∨ 2 × )(× [sandhir] concerns u ended by r, and i = 1 ∨ 2 ×
)(× [sandhin] concerns u ended by n, and i = 1 ∨ 2 ×)(× [sandhif] concerns u ended by f, and i = 1 ∨
 2 × )(× [sandhio] concerns u ended by other letters, and i = 1 × )
```

We read sandhi rules compiled by *compile_sandhi*

```
value (sandhis, sandhir, sandhin, sandhif, sandhio) =
  (Gen.gobble Data.sandhis_file : (rules × rules × rules × rules × rules))
;
(* Special patch for preverbs automaton to ignore cesura rules *)
value filter_cesura sandhis =
  List.map (List.filter (fun (w, _, _) → ¬ (List.mem 50 w))) sandhis
;
(* flag_pv = True for preverbs automaton *)
value get_sandhi flag_pv = fun (* argument is mirror (code u) *)
  [ [] → failwith "get_sandhi␣0"
  | [ 43 (* r *) :: before ] → match before with
        [ [] → failwith "get_sandhi␣1"
        | [ penu :: _ ] → sandhir.(penu)
```

```
        ]
  | [ 48 (* s *) :: before ]
  | [ 16 (* .h *) :: before ] → match before with
        [ [] → failwith "get_sandhi␣2"
        | [ penu :: _ ] → sandhis.(penu)
        ]
  | [ 36 (* n *) :: before ] → match before with
        [ [] → failwith "get_sandhi␣3"
        | [ penu :: _ ] → sandhin.(penu)
        ]
  | [ 21 (* f *) :: before ] → match before with
        [ [] → failwith "get_sandhi␣4"
        | [ penu :: _ ] → sandhif.(penu)
        ]
  | [ c :: _ ] → if c < 0 then failwith "get_sandhi␣5"
                  else if c > 49 then failwith "get_sandhi␣6"
                  else let sandhis = sandhio.(c) in
                       if flag_pv then filter_cesura sandhis else sandhis
  ]
;
(* Same as Compile_sandhi.merge *)
value rec merge st1 st2 = match st1 with
  [ [] → st2
  | [ l1 :: r1 ] → match st2 with
      [ [] → st1
      | [ l2 :: r2 ] → [ (List2.union l1 l2) :: (merge r1 r2) ]
      ]
  ]
;
(* We add to the stack arrays a deco rewrite set *)
```

A rewrite deco maps revu to a list of rules (w,revu,v)

```
type rewrite_set = Deco.deco rule
;
value project n = fun
  [ Deco (_, arcs) → try List.assoc n arcs
                      with [ Not_found → empty ] ]
and get_rules = fun
  [ Deco (rules, _) → rules ]
;
```

```
(* Union of two decos *)
value rec merger d1 d2 = match d1 with
    [ Deco (i1, l1) → match d2 with
    [ Deco (i2, l2) → Deco (i1 @ i2, mrec l1 l2)
  where rec mrec l1 l2 = match l1 with
    [ [] → l2
    | [ (n, d) :: l ] → match l2 with
    [ [] → l1
    | [ (n', d') :: l' ] → if n < n' then [ (n, d) :: mrec l l2 ]
                                else if n' < n then [ (n', d') :: mrec l1 l' ]
                                     else [ (n, merger d d') :: mrec l l' ]
    ] ] ] ]
;
```

Automaton construction with state minimization.

```
value hash_max = 9689 (* Mersenne 21 *)
;
exception Overlap
;
module Auto = Share.Share (struct type domain = auto; value size = hash_max; end)
;
(* Remark - it would be incorrect to share states State (b, d, nd) having the same b and d,
since nd may depend on upper nodes because of contextual rules. *)
value hash0 = 1
and hash1 letter key sum = sum + letter × key
and hash b arcs rules = (* NB. abs needed because possible integer overflow *)
    (abs (arcs + Gen.dirac b + List.length rules)) mod hash_max
;
(* flag_pv = True for preverbs automaton *)
value build_auto flag_pv (rewrite : rewrite_set) = traverse
  (* traverse : word → lexicon → (auto × stack × rewrite_set × int) *)
  (* The occurrence list occ is the reverse of the access word. *)
  where rec traverse occ = fun
  [ Trie.Trie (b, arcs) →
      let local_stack = if b then get_sandhi flag_pv occ else []
      and local_rewrite = if b then rewrite else empty in
      let f (deter, stack, rewrite, span) (n, t) =
          let current = [ n :: occ ] in (* current occurrence *)
          let (auto, st, rew, k) = traverse current t in
          ([ (n, auto) :: deter ], merge st stack,
```

```
          merger (project n rew) rewrite, hash1 n k span) in
      let (deter, stack, rewrite, span) =
          List.fold_left f ([], [], local_rewrite, hash0) arcs in
      let (h, l) = match stack with
          [ [] → ([], []) | [ h :: l ] → (h, l) ] in
            (* the tail l of stack initialises the stack for upper nodes, its head h contains
the list of current choice points *)
      let key = hash b span h in
      let s = Auto.share (State (b, List.rev deter, get_rules rewrite @ h)) key in
      (s, merge local_stack l, rewrite, key)
  ]
;
(* *** IMPORTANT *** The arcs in deter are in decreasing order, because of fold_left. We
put them back in increasing order by List.rev deter. This is not strictly needed, and order of
siblings is not important since access is done with assoc. However, it is crucial to maintain
proper order for operations such as split, which splits an automaton into vowel-initial and
consonant-initial subparts. Thus reversal was enforced when split was introduced in V2.43.
*)
```

What follows is for debugging.

```
value print_sandhi ps (w, u, v) = do
    { ps (rdecode u)
    ; ps "|"
    ; ps (decode v)
    ; ps " -> "
    ; ps (decode w)
    ; ps "\n"
    }
;
value print_cont_sandhi ps word rule = do
    { ps ("[" ^ (rdecode word) ^ "]")
    ; print_sandhi ps rule
    }
;
value record degree assoc = (* assoc is a list [ (deg1, count1); ... ] *)
  let (left, right) = List2.zip degree assoc in
  let update = match right with
        [ [] → [ (degree, 1) :: right ]
        | [ (d, n) :: rest ] → if d = degree then [ (d, n + 1) :: rest ]
                                              else [ (degree, 1) :: right ]
```

```
        ] in
    List2.unstack left update
;
value inspect_bool states =
    let count (t,f) = fun
        [ State (b,_,_) → if b then (t+1,f) else (t,f+1) ] in
    List.fold_left count (0,0) states
;
value add2 (sumt,sumf) states =
    let (t,f) = inspect_bool states in (sumt+t,sumf+f)
;
value inspect_choices spans states =
    let degrees sp = fun
        [ State (_,_,choices) → record (List.length choices) sp ] in
    List.fold_left degrees spans states
;
(* Print automaton statistics in file automaton.txt. *)
value print_stats ps memo =
    let pi n = ps (string_of_int n) in
    let (tt,tf) = Array.fold_left add2 (0,0) memo
    and spans = Array.fold_left inspect_choices [] memo
    and print_spans = List.iter psp
        where psp (sp,count) = do
            { pi count; ps " states of degree "; pi sp; ps "\n" }
    and print_sandhis =
            List.iter (fun l → List.iter (print_sandhi ps) l)
    and print_cont_sandhis cont = fun
            [ [ lev1; lev2 ] → do
                { List.iter (print_cont_sandhi ps cont) lev1
                ; List.iter (print_sandhi ps) lev2
                }
            | [ lev1 ] → List.iter (print_cont_sandhi ps cont) lev1
            | [] → ()
            | _ → failwith "ill-formed sandhi table"
            ] in
    let print_sandhi_rules () =
        let print_table tab =
            for i = 1 to 49 do { print_sandhis tab.(i) }
        and print_table_in_context tab =
            for i = 1 to 49 do { print_cont_sandhis [ i ] tab.(i) } in do
```

```
        { print_table sandhio
        ; print_table_in_context sandhir
        ; print_table_in_context sandhis
        } in do
  { ps "total␣number␣of␣states:␣"; pi (tt + tf)
  ; ps "␣of␣which␣"; pi tt; ps "␣accepting\n"
  ; ps "nondeterminism␣degrees:\n"; print_spans spans
  ; ps "sandhi␣rules:\n"; print_sandhi_rules ()
  }
;
(* Compile builds a tagging transducer from a lexicon index. *)
(* compile : bool → rewrites → Trie.trie → Auto.auto called by Make_automaton and
Make_preverb_automaton *)
value compile stats_flag rewrite lexicon =
  let (transducer, stack, _, _) = build_auto False rewrite [] lexicon in
  match stack with
  [ [] → do { if stats_flag then (* optional monitoring *)
                    let cho = open_out Data.automaton_stats in
                    let ps = output_string cho in do
                    { print_stats ps Auto.memo; close_out cho }
                 else ()
              ; transducer
              }
  | _ → (* Error: some sandhi rule has action beyond one word in the lexicon *)
        raise Overlap
  ]
;
(* Special case: preverb automaton *)
value compile_pv stats_flag rewrite lexicon =
  let (transducer, stack, _, _) = build_auto True rewrite [] lexicon in
  match stack with
  [ [] → do { if stats_flag then (* optional monitoring *)
                    let cho = open_out Data.automaton_stats in
                    let ps = output_string cho in do
                    { print_stats ps Auto.memo; close_out cho }
                 else ()
              ; transducer
              }
  | _ → (* Error: some sandhi rule has action beyond one word in the lexicon *)
        raise Overlap
```

```
  ]
;
(* (For debugging) visualise automaton (undoing sharing). *)
value rec visual = fun
  [ State (b, deter, choices) → do
      { print_string "<State"
      ; print_string (if b then "*" else "␣")
      ; List.iter print_det deter
      ; print_string "␣!␣"
      ; List.iter (print_sandhi print_string) choices
      ; print_string "␣>"
      }
  ]
and print_det (n, state) = do { print_int n; visual state }
;
```

Module Make_automaton

Compiles a trie of inflected forms into a tagging automaton structure
Reads on standard input a trie of inflected forms. Prints on standard output an fsm automaton usable by the segmenter. This executable is invoked in DATA/Makefile. It is executed at morphology/automata construction time.
Instrumentation flag for printing file *automaton_stats*

```
value stats = ref False
;
```

We add to the stack arrays a deco rewrite set
A rewrite deco maps revu to a list of rules (w,revu,v)

```
type rewrite_set = Deco.deco (Word.word × Word.word × Word.word)
;
(* phantom processing with extra rules *)
value extra_phantom_rules = (* persistent precompiled by Compile_sandhi *)
  (Gen.gobble Data.sandhis_ph_file : rewrite_set)
;
value make_transducer inflected =
  let flag = False in (* no monitoring *)
  try Automata.compile flag extra_phantom_rules inflected (* return transducer *)
  with
  [ Sys_error s →
```

```
    let mess = s ^ "\n\n␣***␣First␣call␣make_inflected␣***\n" in do
    { Gen.notify_error mess
    ; exit 1
    }
  | Automata.Overlap → do
    { Gen.notify_error "Conflict␣lexicon/sandhi\n"
    ; exit 1
    }
  ]
;
```

Module Make_preverb_automaton

Compiles a trie of inflected forms into a tagging automaton structure – case of preverb automata, for which special sandhi applies.
We add to the stack arrays a deco rewrite set
A rewrite deco maps revu to a list of rules (w,revu,v)

```
type rewrite_set = Deco.deco (Word.word × Word.word × Word.word)
;
(* preverb glueing may incur retroflexion *)
value extra_preverbs_rules = (* persistent precompiled by Compile_sandhi *)
  (Gen.gobble Data.sandhis_pv_file : rewrite_set)
;
value make_transducer inflected =
  let flag = False in (* no monitoring *)
  try Automata.compile_pv flag extra_preverbs_rules inflected
  with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_inflected␣***\n" in do
    { Gen.notify_error mess
    ; exit 1
    }
  | Automata.Overlap → do
    { Gen.notify_error "Conflict␣lexicon/sandhi\n"
    ; exit 1
    }
  ]
;
```

Interface for module Automata_vector

The vector of automata constructed by *Make_transducers* at make time and loaded by *Load_transducers* at cgi running time
This bizarre datatype is a sum type of transducers used by one of the modes Simple or Full. Thus nouns2, iics2 and ifcs2 are used only in mode Simple. This should be cleaned up, or the Simple mode deprecated

```
open Auto.Auto; (* auto *)
type transducers_datatype =
  { nouns : auto
  ; nouns2 : auto
  ; kama : auto
  ; pronouns : auto
  ; roots : auto
  ; lopas : auto
  ; parts : auto
  ; lopaks : auto
  ; partvocs : auto
  ; iics : auto
  ; iics2 : auto
  ; iifcs : auto
  ; avyayais : auto
  ; avyayafs : auto
  ; vocas : auto
  ; invs : auto
  ; piics : auto
  ; ifcs : auto
  ; ifcs2 : auto
  ; indecls : auto
  ; inftu : auto
  ; absya : auto
  ; abstvaa : auto
  ; iivs : auto
  ; peris : auto
  ; auxis : auto
  ; auxiinvs : auto
  ; auxiks : auto
  ; auxiicks : auto
  ; preverbs : auto
  }
```

;

Module Make_transducers

Prepares the transducers from the morphology banks databases of inflected forms in Resources/DATA

The general scheme is that Resources morphology creates a revmap *nouns*, its underlying minimized trie is used for constructing a segmenting transducer *transn*, and *nouns* is used for lemmatizing, for instance to give the tags of the segments.

A. One-automaton logic for recognizing noun phrases (segmenter,tagger): (This corresponds to our historical prototype, with a unique phase) 1. *make_nouns* creates *nouns_file* from *genders_file* (* Resources *) 2. *Make_automaton.make transducer* creates a shared trie from *nouns_file* and compiles it into a transducer dumped in Resources DATA directory 3. make segmenter uses *transn_file* for segmenting, *nouns_file* for tagging

B. The multi-automata logic, used for more general sentences, does instead: 1. *make_nouns* creates *nouns_file*,*pronouns_file*,*iics_file*,*iivs_file*, *invars_file*,*voca_file*, *inv_file* and *ifcs_file* from *genders_file* 2. *make_roots* creates *roots_file*,*parts_file*,*piics_file*,*piivs_file*, *abstvaa_file*,*absya_file* and *eorts_file* 3. *make_preverbs* creates *preverbs_file* from *verblinks_file* All these files contain decos with morphological lemmas 4. *Make_automaton.make transducer* creates corresponding shared tries and compiles it into transducers dumped in Resources DATA directory but preverbs uses *Make_preverb_automaton.make transducer* instead 5. make segmenter uses *transn_file*, *transr_file*, ... for segmenting, and *nouns_file*, *roots_file*, ... for tagging/lemmatizing.

```
open Auto.Auto; (* auto State  *)
open Automata_vector; (* transducers_datatype *)
```

Takes a morphology map and contracts it to its underlying trie

```
value make_inflected inflected_file =
  try let inflected = (Gen.gobble inflected_file : Morphology.inflected_map) in
      Mini.minimize (Deco.forget_deco inflected)
  with
    [ Sys_error s → do
      { let mess = s ^ "\n\n␣***␣Inflected␣file␣missing␣***\n"
                   ^ inflected_file in
        output_string stderr mess
      ; flush stderr
      ; failwith "Make_inflected"
      }
    ]
```

```
;
value make_preverbs preverbs_file =
  try let preverbs = (Gen.gobble preverbs_file : Deco.deco Word.word) in
      (* minimize as dag *)
      Mini.minimize (Deco.forget_deco preverbs)
  with
    [ Sys_error s → do
       { let mess = s ^ "\n\n␣***␣Preverbs␣file␣missing␣***\n"
                      ^ preverbs_file in
         output_string stderr mess
       ; flush stderr
       ; failwith "Make_inflected"
       }
    ]
;
```

Creates the transducers files
For each lexical category, the trie obtained by forgetting the lemmas is then decorated as transducer of type auto

```
value transducer_of_lemmas deco =
  make_inflected deco |> Make_automaton.make_transducer
and transducer_of_preverbs = (* special sandhi rules *)
  make_preverbs Data.preverbs_file |> Make_preverb_automaton.make_transducer
and empty_trans = State (False, [], []) (* dummy empty transducer *)
;
value make_transducers =
  (* Remark. We could interleave with calls of Mini.reset () for minimize speedup *)
  let nouns = transducer_of_lemmas Data.nouns_file
  and nouns2 = transducer_of_lemmas Data.nouns2_file
  and kama = transducer_of_lemmas Data.kama_file
  and pronouns = transducer_of_lemmas Data.pronouns_file
  and roots = transducer_of_lemmas Data.roots_file
  and lopas = transducer_of_lemmas Data.lopas_file
  and parts = transducer_of_lemmas Data.parts_file
  and lopaks = transducer_of_lemmas Data.lopaks_file
  and partvocs = transducer_of_lemmas Data.partvocs_file
  and iics = transducer_of_lemmas Data.iics_file
  and iics2 = transducer_of_lemmas Data.iics2_file
  and iifcs = transducer_of_lemmas Data.iifcs_file
  and avyayais = transducer_of_lemmas Data.avyayais_file
```

```
and avyayafs = transducer_of_lemmas Data.avyayafs_file
and vocas = transducer_of_lemmas Data.vocas_file
and invs = transducer_of_lemmas Data.invs_file
and piics = transducer_of_lemmas Data.piics_file
and ifcs = transducer_of_lemmas Data.ifcs_file
and ifcs2 = transducer_of_lemmas Data.ifcs2_file
and indecls = transducer_of_lemmas Data.indecls_file
and inftu = transducer_of_lemmas Data.inftu_file
and absya = transducer_of_lemmas Data.absya_file
and abstvaa = transducer_of_lemmas Data.abstvaa_file
and iivs = transducer_of_lemmas Data.iivs_file
and peris = transducer_of_lemmas Data.peris_file
and auxis = transducer_of_lemmas Data.auxis_file
and auxiinvs = transducer_of_lemmas Data.auxiinvs_file
and auxiks = transducer_of_lemmas Data.auxiks_file
and auxiicks = transducer_of_lemmas Data.auxiicks_file
and preverbs = transducer_of_preverbs in
let (transducers_data : transducers_datatype) =
{ nouns = nouns
; nouns2 = empty_trans
; kama = kama
; pronouns = pronouns
; roots = roots
; lopas = lopas
; parts = parts
; lopaks = lopaks
; partvocs = partvocs
; iics = iics
; iics2 = empty_trans
; iifcs = iifcs
; avyayais = avyayais
; avyayafs = avyayafs
; vocas = vocas
; invs = invs
; piics = piics
; ifcs = ifcs
; ifcs2 = empty_trans
; indecls = indecls
; inftu = inftu
; absya = absya
```

```
; abstvaa = abstvaa
; iivs = iivs
; peris = peris
; auxis = auxis
; auxiinvs = auxiinvs
; auxiks = auxiks
; auxiicks = auxiicks
; preverbs = preverbs
}
and transducers_data2 =
{ nouns = empty_trans
; nouns2 = nouns2
; kama = empty_trans
; pronouns = pronouns
; roots = roots
; lopas = lopas
; parts = empty_trans
; lopaks = empty_trans
; partvocs = empty_trans
; iics = empty_trans
; iics2 = iics2
; iifcs = empty_trans
; avyayais = empty_trans
; avyayafs = empty_trans
; vocas = empty_trans
; invs = invs
; piics = empty_trans
; ifcs = empty_trans
; ifcs2 = ifcs2
; indecls = indecls
; inftu = empty_trans
; absya = absya
; abstvaa = abstvaa
; iivs = iivs
; peris = empty_trans
; auxis = auxis
; auxiinvs = auxiinvs
; auxiks = empty_trans
; auxiicks = empty_trans
; preverbs = preverbs
```

```
  } in do
  { Gen.dump transducers_data Data.transducers_file (* Complete mode *)
  ; Gen.dump transducers_data2 Data.transducers_file2 (* Simple mode *)
  }
;
```

Module Make_xml_data

Prepares XML data banks from the databases of inflected forms in Resources, conformant to *WX_morph.dtd* or *SL_morph.dtd* according to transliteration CAUTION. Update the dtd files when changing the tags or adding new tags.
What follows merges previous *Print_inflected* and *Morpho_xml* modules

```
open Skt_morph;
open Morphology; (* inflected and its constructors Noun_form ,... *)
value read_inflected file =
  (Gen.gobble file : Morphology.inflected_map)
;
value abort mess = do
  { output_string stderr mess
  ; flush stderr
  ; failwith "Print_inflected"
  }
;
value read_nouns () =
  try read_inflected Data.nouns_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_nouns␣***\n"
               ^ "␣to␣create␣" ^ Data.nouns_file ^ "\n" in
    abort mess
  ]
and read_pronouns () =
  try read_inflected Data.pronouns_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_nouns␣***\n"
               ^ "␣to␣create␣" ^ Data.nouns_file ^ "\n" in
    abort mess
  ]
and read_roots () =
  try read_inflected Data.roots_file with
```

```
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_roots␣***\n"
                 ^ "␣to␣create␣" ^ Data.roots_file ^ "\n" in
    abort mess
  ]
and read_parts () =
  try read_inflected Data.parts_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_parts␣***\n"
                 ^ "␣to␣create␣" ^ Data.parts_file ^ "\n" in
    abort mess
  ]
and read_indecls () =
  try read_inflected Data.indecls_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_parts␣***\n"
                 ^ "␣to␣create␣" ^ Data.indecls_file ^ "\n" in
    abort mess
  ]
and read_absya () =
  try read_inflected Data.absya_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_parts␣***\n"
                 ^ "␣to␣create␣" ^ Data.absya_file ^ "\n" in
    abort mess
  ]
and read_abstvaa () =
  try read_inflected Data.abstvaa_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_parts␣***\n"
                 ^ "␣to␣create␣" ^ Data.abstvaa_file ^ "\n" in
    abort mess
  ]
and read_iics () =
  try read_inflected Data.iics_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_nouns␣***\n"
                 ^ "␣to␣create␣" ^ Data.iics_file ^ "\n" in
    abort mess
  ]
```

```
and read_voca () =
  try read_inflected Data.vocas_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_nouns␣***\n"
                 ^ "␣to␣create␣" ^ Data.vocas_file ^ "\n" in
    abort mess
  ]
and read_ifcs () =
  try read_inflected Data.ifcs_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_nouns␣***\n"
                 ^ "␣to␣create␣" ^ Data.ifcs_file ^ "\n" in
    abort mess
  ]
and read_iivs () =
  try read_inflected Data.iivs_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_nouns␣***\n"
                 ^ "␣to␣create␣" ^ Data.iivs_file ^ "\n" in
    abort mess
  ]
and read_avyayafs () =
  try read_inflected Data.iivs_file with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_nouns␣***\n"
                 ^ "␣to␣create␣" ^ Data.avyayafs_file ^ "\n" in
    abort mess
  ]
(* NB auxis_file not needed - its forms are included in roots_file and similarly ayayais forms are included in indecls. *)
and read_prevs () =
  try (Gen.gobble Data.preverbs_file : Deco.deco Word.word) with
  [ Sys_error s →
    let mess = s ^ "\n\n␣***␣First␣call␣make_prevs␣***\n"
                 ^ "␣to␣create␣" ^ Data.preverbs_file ^ "\n" in
    abort mess
  ]
;
(*************************************)
(* Now printing in XML format on stdout *)
```

```
(************************************)
value ps = print_string
;
value pl s = ps (s ^ "\n")
;
(* Examples (in SL1 transliteration) ¡f form="AGAtAt"¿¡na¿¡abl/¿¡sg/¿¡mas/¿¡/na¿¡s stem="AGAta"/¿¡/
¡f form="aham"¿¡na¿¡nom/¿¡sg/¿¡dei/¿¡/na¿¡s stem="aham"/¿¡/f¿ ¡f form="patati"¿¡v¿¡cj¿¡prim/¿¡/cj
stem="pat"/¿¡/f¿ *)
```

parametrization of transliteration scheme

```
value decode = fun
  [ "SL" → Canon.decode_SL
  | "WX" → Canon.decode_WX
  | "VH" → Canon.decode2 (* takes care of possible hiatus *)
  | s → failwith ("Unknown␣transliteration␣scheme" ^ s)
  ]
 ;
(* Paradigm parameters *)
(* kind attributes: present class of primary conjug aka ga.na, from 1 to 11 (vn) aorist kind,
from 1 to 7 pfp kind (gerundive) 1 -ya ¡gya¿ 2 -iiya ¡iya¿ 3 -tavya ¡tav¿ *)
value kind_attr k = "␣gn=\"" ^ string_of_int k ^ "\""
;
(* present class aka ga.na, from 1 to 11 *)
value pg k = if k > 11 ∨ k = 0 (* redundant with conjugation *) then ()
             else kind_attr k |> ps
;
value print_number = fun
  [ Singular → "<sg/>" —> ps
  | Dual → "<du/>" —> ps
  | Plural → "<pl/>" —> ps
  ]
and print_gender = fun
  [ Mas → "<mas/>" —> ps
  | Neu → "<neu/>" —> ps
  | Fem → "<fem/>" —> ps
  | Deictic _ → "<dei/>" —> ps
  ]
and print_case = fun
  [ Nom → "<nom/>" —> ps
  | Acc → "<acc/>" —> ps
```

```
  | Ins → "<ins/>" —> ps
  | Dat → "<dat/>" —> ps
  | Abl → "<abl/>" —> ps
  | Gen → "<gen/>" —> ps
  | Loc → "<loc/>" —> ps
  | Voc → "<voc/>" —> ps
  ]
and print_person = fun
  [ First → "<fst/>" —> ps
  | Second → "<snd/>" —> ps
  | Third → "<trd/>" —> ps
  ]
and print_voice = fun
  [ Active → "<para/>" —> ps
  | Middle → "<atma/>" —> ps
  | Passive → "<pass/>" —> ps
  ]
and print_conjugation cg = do
  { "<cj>" —> ps
  ; match cg with
        [ Primary → "<prim/>" —> ps
        | Causative → "<ca/>" —> ps
        | Intensive → "<int/>" —> ps
        | Desiderative → "<des/>" —> ps
        ]
  ; "</cj>" —> ps
  }
and print_pr_mode pr = do
  { "<md>" —> ps
  ; match pr with
        [ Present → "<pr/>" —> ps
        | Imperative → "<ip/>" —> ps
        | Optative → "<op/>" —> ps
        | Imperfect → "<im/>" —> ps
        ]
  ; "</md>" —> ps
  }
and print_tense = fun
  [ Future → "<fut/>" —> ps
  | Perfect → "<prf/>" —> ps
```

```
  | Aorist k → do { "<aor" —> ps; kind_attr k |> ps; "/>" —> ps }
  | Injunctive k → do { "<inj" —> ps; kind_attr k |> ps; "/>" —> ps }
  | Benedictive → "<ben/>" —> ps
  | Conditional → "<cnd/>" —> ps
  | Subjunctive → "<subj/>" —> ps
  ]
;
value pfutp_kind = fun
  [ 1 → "<gya/>"
  | 2 → "<iya/>"
  | 3 → "<tav/>"
  | n → failwith ("Unknown␣pfutp␣kind␣" ^ string_of_int n)
  ]
;
value print_nominal = fun
  [ Ppp → "<ppp/>" —> ps
  | Pppa → "<ppa/>" —> ps
  | Ppra k → do { "<ppr" —> ps; pg k; ">" —> ps; print_voice Active; "</ppr>" —> ps }
  | Pprm k → do { "<ppr" —> ps; pg k; ">" —> ps; print_voice Middle; "</ppr>" —> ps }
  | Pprp → "<pprp/>" —> ps
  | Ppfta → do { "<ppft>" —> ps; print_voice Active; "</ppft>" —> ps }
  | Ppftm → do { "<ppft>" —> ps; print_voice Middle; "</ppft>" —> ps }
  | Pfuta → do { "<pfut>" —> ps; print_voice Active; "</pfut>" —> ps }
  | Pfutm → do { "<pfut>" —> ps; print_voice Middle; "</pfut>" —> ps }
  | Pfutp k → do { "<pfutp>" —> ps; pfutp_kind k |> ps; "</pfutp>" —> ps }
  | _ → "<act/>" —> ps (* action verbal nouns *)
  ]
;
value print_system = fun
  [ Conjug t v → do { "<tp>" —> ps; print_tense t; print_voice v; "</tp>" —> ps }
  | Presenta k pr → do { "<prs" —> ps; pg k; ">" —> ps;
                         print_pr_mode pr; "<para/></prs>" —> ps }
  | Presentm k pr → do { "<prs" —> ps; pg k; ">" —> ps;
                         print_pr_mode pr; "<atma/></prs>" —> ps }
  | Presentp pr → do { "<pas>" —> ps; print_pr_mode pr; "</pas>" —> ps }
  | Perfut v → do { "<pef>" —> ps; print_voice v; "</pef>" —> ps }
  ]
and print_invar = fun
  [ Infi → "<inf/>" —> ps
  | Absoya → "<abs/>" —> ps
```

```
  | Perpft → "<per/>" —> ps
  ]
;
(* Next 3 functions print conjugation in different order than Print_dict *)
value print_finite (c, p) = do
  { print_conjugation c
  ; "<sys>" —> ps; print_system p; "</sys>" —> ps
  }
 and print_verbal (c, n) = do
  { print_conjugation c
  ; "<no>" —> ps; print_nominal n; "</no>" —> ps
  }
 and print_modal (c, t) = do
  { print_conjugation c
  ; "<iv>" —> ps; print_invar t; "</iv>" —> ps
  }
;
value print_morph = fun
  [ Noun_form g n c → do
       { "<na>" —> ps
       ; print_case c
       ; print_number n
       ; print_gender g
       ; "</na>" —> ps
       }
  | Part_form v g n c → do
       { "<pa><na>" —> ps
       ; print_case c
       ; print_number n
       ; print_gender g
       ; "</na>" —> ps
       ; "<kr>" —> ps
       ; print_verbal v
       ; "</kr></pa>" —> ps
       }
  | Verb_form f n p → do
       { "<v>" —> ps
       ; print_finite f
       ; "<np>" —> ps
       ; print_number n
```

```
      ; print_person p
      ; "</np></v>" —> ps
      }
  | Ind_form k → do
      { "<uf>" —> ps
      ; match k with
            [ Adv | Avya | Default → "<ind/>" —> ps
            | Interj → "<interj/>" —> ps
            | Part → "<parti/>" —> ps
            | Prep → "<prep/>" —> ps
            | Conj → "<conj/>" —> ps
            | Tas → "<tasil/>" —> ps
            | Abs → () (* redundant absolutive forms *)
            | Infl → () (* redundant inflected form *)
            | Nota → () (* skipped grammatical notation *)
            ]
      ; "</uf>" —> ps
      }
  | Avyayaf_form → "<avya/>" —> ps
  | Abs_root c → do { "<ab>" —> ps; print_conjugation c; "</ab>" —> ps }
  | Bare_stem | Avyayai_form → "<iic/>" —> ps
  | Gati → "<iiv/>" —> ps
  | Ind_verb m → do { "<vu>" —> ps; print_modal m ; "</vu>" —> ps }
  | _ → failwith "Anomaly␣print_morph"
  ]
;
value print_inverse_map_xml trans form (delta, morphs) =
  let print_skt s = "\"" ^ s ^ "\"" —> ps in
    if Phonetics.phantomatic form then ((* phantomatic forms skipped *)) else do
  { "<f␣form=" —> ps
  ; print_skt (decode trans form)
  ; ">" —> ps
  ; List.iter print_morph morphs
  ; "<s␣stem=" —> ps
  ; print_skt (decode trans (Word.patch delta form))
  ; "/></f>\n" —> ps
  }
;
(* Outputs an XML stream on stdout *)
value print_header trans = do
```

```
  { "<?xml␣version=\"1.0\"␣encoding=\"UTF-8\"?>" —> pl
  ; "<!DOCTYPE␣forms␣SYSTEM␣\"" ^ trans ^ "_morph.dtd\">" —> pl
  ; "<!--␣Header" —> pl
  ; "<meta␣name=\"title\"␣content=\"Sanskrit␣Morphology\"" —> ps; "\">" —> pl
  ; "<meta␣name=\"author\"␣content=\"" —> ps; Html.author_name |> ps; "\">" —> ps
  ; "<meta␣name=\"date\"␣content=\"" —> ps; Date.dico_date |> ps; "\">" —> pl
  ; "<meta␣name=\"copyright\"␣content=\"" —> ps; Html.copyright |> ps; "\">" —> pl
  ; "<meta␣name=\"keywords\"␣content=\"sanskrit;␣morphology\">␣-->" —> pl
  }
;
value print_xml trans inflected_map = do
  { print_header trans
  ; "<forms>" —> pl
  ; Deco.iter (print_inverse_map_xml trans) inflected_map
  ; "</forms>" —> pl
  }
;
(* For printing preverb lists *)
value print_xml_word trans (w, _) = do
  { "<pv␣form=" —> ps
  ; "\"" ^ (decode trans w) ^ "\"/>" —> pl
  }
;
value print_xml_list trans banks prevs = do
  { print_header trans
  ; "<forms>" —> pl
  ; let print_bank inflected_map =
          Deco.iter (print_inverse_map_xml trans) inflected_map in
    List.iter print_bank banks
  ; List.iter (print_xml_word trans) (Deco.contents prevs)
  ; "</forms>" —> pl
  }
;
```

Prints big XML stream to stdout

```
value print_xml_morphology trans =
  let nouns = read_nouns ()
  and pronouns = read_pronouns ()
  and verbs = read_roots ()
  and parts = read_parts ()
```

```
  and indecls = read_indecls ()
  and abstva = read_abstvaa ()
  and absya = read_absya ()
  and iics = read_iics ()
  and voca = read_voca ()
  and ifcs = read_ifcs ()
  and avya = read_avyayafs ()
  and iivs = read_iivs ()
  and prevs = read_prevs () in
  print_xml_list trans [ nouns; pronouns; verbs; parts; indecls; abstva; absya;
                         iics; ifcs; avya; voca; iivs ] prevs
;
```

Analyse the transliteration argument to command *make_xml_data*

```
try Arg.parse [ ("-trans", Arg.String print_xml_morphology, "") ]
              (fun s → raise (Arg.Bad s))
              "Usage␣:␣make_xml_data␣-trans␣t␣(where␣t␣is␣SL␣WX␣or␣VH)"
with [ Failure _ → () ]
;
```

Module Morpho_string

Linearizes morphological information as a string. Used in *Morpho*, *Morpho_tex*, *Lexer*.

```
open Skt_morph;
open Morphology; (* inflected, Noun_form, ... *)
value gana_str k =
  if k = 11 then "␣[vn.]"
  else if k > 10 (* redundant with conjugation *) then ""
  else if k = 0 then failwith "gana_str"
  else "␣[" ^ string_of_int k ^ "]"
;
value string_voice = fun
  [ Active → "␣ac."
  | Middle → "␣md."
  | Passive → "␣ps."
  ]
and string_conjugation = fun
  [ Primary → ""
  | Causative → "ca.␣"
```

```
  | Intensive → "int.␣"
  | Desiderative → "des.␣"
  ]
and string_nominal = fun
  [ Ppp → "pp."
  | Pppa → "ppa."
  | Ppra k → "ppr." ^ (gana_str k) ^ "␣ac."
  | Pprm k → "ppr." ^ (gana_str k) ^ "␣md."
  | Pprp → "ppr." ^ "␣ps."
  | Ppfta → "ppf." ^ "␣ac."
  | Ppftm → "ppf." ^ "␣md."
  | Pfuta → "pfu." ^ "␣ac."
  | Pfutm → "pfu." ^ "␣md."
  | Pfutp k → "pfp." ^ (gana_str k)
  | Action_noun → "act."
  ]
and string_tense = fun
  [ Future → "fut."
  | Perfect → "pft."
  | Aorist k → "aor." ^ (gana_str k)
  | Injunctive k → "inj." ^ (gana_str k)
  | Conditional → "cond."
  | Benedictive → "ben."
  | Subjunctive → "subj."
  ]
and string_case = fun
  [ Nom → "nom."
  | Acc → "acc."
  | Ins → "i."
  | Dat → "dat."
  | Abl → "abl."
  | Gen → "g."
  | Loc → "loc."
  | Voc → "voc."
  ]
and string_number = fun
  [ Singular → "␣sg.␣"
  | Dual → "␣du.␣"
  | Plural → "␣pl.␣"
  ]
```

```
and string_gender = fun
  [ Mas → "m."
  | Neu → "n."
  | Fem → "f."
  | Deictic _ → "*"
  ]
and string_pr_mode = fun
  [ Present → "pr."
  | Imperative → "imp."
  | Optative → "opt."
  | Imperfect → "impft."
  ]
and string_person = fun
  [ First → "1"
  | Second → "2"
  | Third → "3"
  ]
and string_ind_kind = fun
  [ Part → "part."
  | Prep → "prep."
  | Conj → "conj."
  | Abs → "abs."
  | Adv → "adv."
  | Tas → "tasil"
  | _ → "ind."
  ]
and string_invar = fun
  [ Infi → "inf."
  | Absoya → "abs."
  | Perpft → "per.␣pft."
  ]
;
value string_paradigm = fun
  [ Conjug t v → (string_tense t) ^ (string_voice v)
  | Presenta k pr → (string_pr_mode pr) ^ (gana_str k) ^ "␣ac."
  | Presentm k pr → (string_pr_mode pr) ^ (gana_str k) ^ "␣md."
  | Presentp pr → (string_pr_mode pr ) ^ "␣ps."
  | Perfut v → "per.␣fut." ^ (string_voice v)
  ]
;
```

```
value string_finite (c, p) = (string_conjugation c) ^ (string_paradigm p)
and string_verbal (c, n) = (string_conjugation c) ^ (string_nominal n)
and string_modal (c, i) = (string_conjugation c) ^ (string_invar i)
;
value string_morph = fun
  [ Noun_form g n c
  | Part_form _ g n c → (string_gender g) ^ (string_number n) ^ (string_case c)
  | Bare_stem | Avyayai_form → "iic."
  | Avyayaf_form → "ind."
  | Verb_form f n p → (string_finite f) ^ (string_number n) ^ (string_person p)
  | Ind_form k → string_ind_kind k
  | Abs_root c → (string_conjugation c) ^ "abs."
  | Gati → "iiv."
  | Ind_verb m → string_modal m
  | Unanalysed → "?"
  | PV pvs → "pv."
  ]
;
(* end; *)
```

Module Morpho

Prints morphological information, including derivative morphology. Used in *Morpho_html* and *Morpho_ext*

```
open Skt_morph;
open Morphology;
  (* inflected and its constructors Noun_form, ..., homo_krid *)
open Naming; (* homo_undo look_up_homo unique_kridantas lexical_kridantas *)
open Morpho_string (* string_morph string_verbal *);
```

```
module Morpho_out (Chan : sig value chan : ref out_channel; end)
 = struct
```

```
value ps s = output_string Chan.chan.val s
;
value pl s = ps (s ^ "\n")
;
value pr_word w = ps (Canon.decode w)
;
value print_morph m = string_morph m |> ps
```

```
and print_verbal vb = string_verbal vb |> ps
;
value select_morph (seg_num, sub, seg_count) morph = do
  { let string_num = string_of_int seg_num
    and seg = (string_of_int sub) ^ "," ^ (string_of_int seg_count) in
    let radio_cond = Html.radio_input_dft string_num seg "" in
    match (sub, seg_count) with
    [ (1,1) → radio_cond True ^ "␣" —> ps
                      (* NB: only the first button is selected *)
    | _ → radio_cond False ^ "␣" —> ps
    ]
  ; print_morph morph
  }
;
value rec select_morphs (seg_num, sub) seg_count = fun
  [ [] → ()
  | [ last :: [] ] → select_morph (seg_num, sub, seg_count) last
  | [ first :: rest ] → do
      { select_morph (seg_num, sub, seg_count) first
      ; ps "␣|␣"
      ; select_morphs (seg_num, sub) (seg_count + 1) rest
      }
  ]
;
value print_morphs (seg_num, sub) morphs = match seg_num with
  [ 0 → let bar () = ps "␣|␣" in
          List2.process_list_sep print_morph bar morphs
  | _ → select_morphs (seg_num, sub) 1 morphs
  ]
;
```

(* The following print functions insert in the HTML output links to entries in the lexicon, also radio buttons and other marks for user choices. *)

pe : *word* → *unit* is *Morpho_html.print_entry* with hyperlink, *pne* : *word* → *unit* is *Morpho_html.print_stem*, *pu* : *word* → *unit* prints un-analysed chunks.

```
value print_inv_morpho pe pne pu form (seg_num, sub) generative (delta, morphs) =
  let stem = Word.patch delta form in do (* stem may have homo index *)
    { ps "["
    ; if generative then (* interpret stem as unique name *)
        let (homo, bare_stem) = homo_undo stem in
```

```
        let krit_infos = Deco.assoc bare_stem unique_kridantas in
        try let (verbal, root) = look_up_homo homo krit_infos in do
        { match Deco.assoc bare_stem lexical_kridantas with
          [ [] (* not in lexicon *) →
              if stem = [ 3; 32; 1 ] (* ita ifc *) then pe stem
                                                   else pne bare_stem
          | entries (* bare stem is lexicalized *) →
              if List.exists (fun (_, h) → h = homo) entries
                 then pe stem (* stem with exact homo is lexical entry *)
              else pne bare_stem
          ]
        ; ps "␣{␣"; print_verbal verbal; ps "␣}["; pe root; ps "]"
        } with [ _ → pu bare_stem ]
      else match morphs with
           [ [ Unanalysed ] → pu stem
           | _ → pe stem
           ]
    ; ps "]{"
    ; print_morphs (seg_num, sub) morphs
    ; ps "}"
    }
;
(* Decomposes a preverb sequence into the list of its components *)
value decomp_pvs pvs =
  Deco.assoc pvs Naming.preverbs_structure
;
(* Used in Morpho_html *)
value print_inv_morpho_link pvs pe pne pu form =
  let pv = if Phonetics.phantomatic form then [ 2 ] (* aa- *) else pvs in
  let encaps print e = (* encapsulates prefixing with possible preverbs *)
      if pv = [] then print e else
         let pr_pv pv = do { pe pv; ps "-" } in do
               { List.iter pr_pv (decomp_pvs pvs)
               ; print e
               } in
         print_inv_morpho (encaps pe) (encaps pne) pu form
(* Possible overgeneration when derivative of a root non attested with pv since only existential test in Dispatcher.validate_pv. Thus anusandhiiyate should show dhaa#1, not dhaa#2, dhii#1 or dhyaa *)
;
```

Used in *Lexer.record_tagging* for regression analysis

```
value report_morph gen form (delta, morphs) =
  let stem = Word.patch delta form in do (* stem may have homo index *)
    { ps "{␣"
    ; print_morphs (0, 0) morphs
    ; ps "␣}["
    ; if gen then (* interpret stem as unique name *)
        let (homo, bare_stem) = homo_undo stem in
        let krid_infos = Deco.assoc bare_stem unique_kridantas in
        let (vb, root) = look_up_homo homo krid_infos in do
        { match Deco.assoc stem lexical_kridantas with
          [ [] (* not in lexicon *) → do { ps "G:"; pr_word bare_stem }
          | _ (* stem is lexical entry *) → do { ps "L:"; pr_word stem }
          ]
        ; ps "␣{␣"; print_verbal vb; ps "␣}["; pr_word root; ps "]"
        }
      else pr_word stem
    ; ps "]"
    }
;
end;
```

Module Declension

CGI-bin declension for computing declensions.
This CGI is triggered by page *grammar_page* in *dico_dir*.
Reads its input in shell variable *QUERY_STRING* URI-encoded.
Prints an html document of substantive declinations on *stdout*.

```
open Skt_morph;
open Morphology; (* Noun_form etc. *)
open Html; (* narrow_screen html_red etc. *)
open Web; (* ps pl etc. *)
open Cgi; (* create_env etc. *)
open Multilingual;
    (* font Deva Roma declension_title compound_name avyaya_name *)
```

```
value dtitle font = h1_title (declension_title narrow_screen font)
and meta_title = title "Sanskrit␣Grammarian␣Declension␣Engine"
and back_ground = background Chamois
and hyperlink_title font link =
  if narrow_screen then link
  else declension_caption font ^ "␣" ^ link
;
value pr code =
  html_red (Canon.uniromcode code) |> ps (* roman with diacritics *)
and pr_deva code =
  html_devared (Canon.unidevcode code) |> ps (* devanagari *)
;
value pr_f font word =
  let code = Morpho_html.final word (* visarga correction *) in do
  { match font with
    [ Deva → pr_deva code
    | Roma → pr code
    ]
  ; ps "␣"
  }
and pr_i font word = do (* special for iic *)
  { match font with
    [ Deva → do { pr_deva word; pr_deva [ 0 ] }
    | Roma → do { pr word; pr [ 0 ] }
    ]
  ; ps "␣"
  }
;
value prlist_font font =
  let pr = pr_f font
  and bar = html_green "␣|␣" in
  prlistrec
     where rec prlistrec = fun
       [ [] → ()
       | [ x ] → pr x
       | [ x :: l ] → do { pr x; ps bar; prlistrec l }
       ]
;
value display_title font = do
  { html_paragraph |> pl
```

```
  ; table_begin (centered Mauve) |> pl
  ; tr_begin |> ps
  ; th_begin |> ps
  ; dtitle font |> ps
  ; th_end |> ps
  ; tr_end |> ps
  ; table_end |> pl (* Mauve *)
  ; html_paragraph |> pl
  }
and display_subtitle title = do
  { html_paragraph |> pl
  ; table_begin (centered Deep_sky) |> pl
  ; tr_begin |> ps
  ; th_begin |> ps
  ; title |> ps
  ; th_end |> ps
  ; tr_end |> ps
  ; table_end |> pl (* Centered *)
  ; html_paragraph |> pl
  }
;
value cases_of decls =
  let reorg (v, n, a, i, d, ab, g, l) (c, form) = match c with
      [ Voc → ([ form :: v ], n, a, i, d, ab, g, l)
      | Nom → (v, [ form :: n ], a, i, d, ab, g, l)
      | Acc → (v, n, [ form :: a ], i, d, ab, g, l)
      | Ins → (v, n, a, [ form :: i ], d, ab, g, l)
      | Dat → (v, n, a, i, [ form :: d ], ab, g, l)
      | Abl → (v, n, a, i, d, [ form :: ab ], g, l)
      | Gen → (v, n, a, i, d, ab, [ form :: g ], l)
      | Loc → (v, n, a, i, d, ab, g, [ form :: l ])
      ]
  and init = ([], [], [], [], [], [], [], [])
  in List.fold_left reorg init decls (* (v,n,a,i,d,ab,g,l) *)
;
value print_ro1 caption s d p = do
  { tr_begin |> ps
  ; th_begin |> ps
  ; caption |> ps
  ; xml_next "th" —> ps
```

```
  ; s |> ps
  ; xml_next "th" —> ps
  ; d |> ps
  ; xml_next "th" —> ps
  ; p |> ps
  ; th_end |> ps
  ; tr_end |> ps
  }
;
value print_row_font font case s d p =
  let prlist = prlist_font font in do
  { tr_mouse_begin (color Light_blue) (color Pale_yellow) |> ps
  ; th_begin |> ps
  ; case |> ps
  ; xml_next "th" —> ps
  ; s |> prlist
  ; xml_next "th" —> ps
  ; d |> prlist
  ; xml_next "th" —> ps
  ; p |> prlist
  ; th_end |> ps
  ; tr_end |> pl
  }
;
value display_gender font gender = fun
  [ [] → ()
  | l →
    let reorg (sg, du, pl) (n, c, form) = match n with
        [ Singular → ([ (c, form) :: sg ], du, pl)
        | Dual → (sg, [ (c, form) :: du ], pl)
        | Plural → (sg, du, [ (c, form) :: pl ])
        ]
    and init = ([], [], []) in
    let (s, d, p) = List.fold_left reorg init l in
    let (v1, n1, a1, i1, d1, b1, g1, l1) = cases_of s
    and (v2, n2, a2, i2, d2, b2, g2, l2) = cases_of d
    and (v3, n3, a3, i3, d3, b3, g3, l3) = cases_of p
    and caption = gender_caption gender font
    and print_row = print_row_font font in do
    { pl html_paragraph
```

```
      ; pl (table_begin_style Inflexion [ ("border","2"); padding5 ])
      ; let sing = number_caption Singular font
         and dual = number_caption Dual font
         and plur = number_caption Plural font in
         print_ro1 caption sing dual plur
      ; print_row (case_caption Nom font) n1 n2 n3
      ; print_row (case_caption Voc font) v1 v2 v3
      ; print_row (case_caption Acc font) a1 a2 a3
      ; print_row (case_caption Ins font) i1 i2 i3
      ; print_row (case_caption Dat font) d1 d2 d3
      ; print_row (case_caption Abl font) b1 b2 b3
      ; print_row (case_caption Gen font) g1 g2 g3
      ; print_row (case_caption Loc font) l1 l2 l3
      ; ps table_end
      ; pl html_paragraph
      }
   ]
;
value display_iic font = fun
   [ [] → ()
   | l → do
      { html_paragraph |> pl
      ; h3_begin C3 |> ps
      ; compound_name font |> ps; ps "␣"
      ; let print_iic w = pr_i font w in
         List.iter print_iic l
      ; h3_end |> ps
      }
   ]
;
value display_avy font = fun
   [ [] → ()
   | l → do
      { html_paragraph |> pl
      ; h3_begin C3 |> ps
      ; avyaya_name font |> ps; ps "␣"
      ; let ifc_form w = [ 0 ] (* - *) @ w in
         let print_iic w = pr_f font (ifc_form w) in
         List.iter print_iic l
      ; h3_end |> ps
```

```
    }
  ]
;
value sort_out accu form = fun
      [ [ (_, morphs) ] → List.fold_left (reorg form) accu morphs
        where reorg f (mas, fem, neu, any, iic, avy) = fun
          [ Noun_form g n c → let t = (n, c, f) in
               match g with
                 [ Mas → ([ t :: mas ], fem, neu, any, iic, avy)
                 | Fem → (mas, [ t :: fem ], neu, any, iic, avy)
                 | Neu → (mas, fem, [ t :: neu ], any, iic, avy)
                 | Deictic _ → (mas, fem, neu, [ t :: any ], iic, avy)
                 ]
          | Bare_stem | Gati → (mas, fem, neu, any, [ f :: iic ], avy)
          | Avyayaf_form → (mas, fem, neu, any, iic, [ f :: avy ])
          | Ind_form _ | Verb_form _ _ _ | Ind_verb _ | Abs_root _
          | Avyayai_form | Unanalysed | PV _
          | Part_form _ _ _ _ →
            failwith "Unexpected form in declensions"
          ]
      | _ → failwith "Weird table"
      ]
and init = ([], [], [], [], [], [])
;
value display_inflected font (gen_deco, pn_deco, voca_deco, iic_deco, avy_deco) =
  let nouns = Deco.fold sort_out init gen_deco in
  let non_vocas = Deco.fold sort_out nouns pn_deco in
  let (mas, fem, neu, any, _, _) = Deco.fold sort_out non_vocas voca_deco
  and iic = List.map fst (Deco.contents iic_deco)
  and avy = List.map fst (Deco.contents avy_deco) in do
  { center_begin |> pl
  ; display_gender font Mas mas
  ; display_gender font Fem fem
  ; display_gender font Neu neu
  ; display_gender font (Deictic Numeral) any (* arbitrary *)
  ; display_iic font iic
  ; display_avy font avy
  ; center_end |> pl
  ; html_paragraph |> pl
  }
```

```
;
(* entry : skt part :string *)
value emit_decls font entry decli part =
  let inflected = Nouns.fake_compute_decls (entry, decli) part in
  display_inflected font inflected
;
value look_up font entry decli part =
  let code = Encode.code_string entry in (* normalisation *)
  let e = Canon.decode code in (* coercion skt to string *)
  emit_decls font e decli part
;
(* This is very fragile: lexicon update induces code adaptation. *)
(* Temporary - should be subsumed by unique naming structure. *)
value resolve_homonym stem = match stem with
  [ "atra" | "ad" | "abhii" | "iiz" | ".rc" | "chad" | "dam" | "dah" | "daa"
  | "diz" | "diiv" | "duh" | "d.rz" | "druh" | "dvi.s" | "dhii" | "nas" | "nii"
  | "pad" | "budh" | "bhii" | "bhuu" | "math" | "yaa" | "yuj" | "raa" | "raaj"
  | "luu" | "viraaj" | "viz" | "vii" | "zubh" | "sa" | "sah" | "saa" | "s.rj"
  | "snih" | "snuh" | "han"
    → stem ^ "#2"
  | "agha" | "afga" | "aja" | "aaza" | "e.sa" | "ka" | "kara" | "tapas"
  | "dhaavat" | "nimita" | "pa" | "bhavat" | "bhaama" | "ya" | "yama"
  | "yaat.r" (* 1/2 *) | "vaasa" | "zaava" | "zrava.na" | "zvan" | "sthaa"
    → stem ^ "#1"
  | "paa" → stem ^ "#3"
  | _ → stem
  ]
;
value in_lexicon entry = (* entry as a string in VH transliteration *)
  Index.is_in_lexicon (Encode.code_string entry)
and doubt s = "?" ^ s
;
value gender_of = fun
  [ "Mas" → Mas
  | "Fem" → Fem
  | "Neu" → Neu
  | "Any" → Deictic Numeral (* arbitrary *)
  | s → failwith ("Weird␣gender" ^ s)
  ]
;
```

```
value decls_engine () = do
  { pl http_header
  ; page_begin meta_title
  ; pl (body_begin back_ground)
  ; let query = Sys.getenv "QUERY_STRING" in
    let env = create_env query in
    let url_encoded_entry = try List.assoc "q" env
                            with [ Not_found → failwith "Entry␣name␣missing" ]
    and url_encoded_gender = get "g" env "Mas"
    and url_encoded_participle = get "p" env ""
    and url_encoded_source = get "r" env ""
        (* optional root origin - used by participles in conjugation tables *)
    and font = font_of_string (get "font" env Paths.default_display_font)
    and translit = get "t" env "VH" (* DICO created in VH trans *)
    and lex = get "lex" env "SH" (* default Heritage *) in
    let entry_tr = decode_url url_encoded_entry (* : string in translit *)
    and gender = gender_of (decode_url url_encoded_gender)
    and part = decode_url url_encoded_participle
    and code = Encode.switch_code translit
    and lang = language_of lex
    and source = decode_url url_encoded_source (* cascading from conjug *)
    and () = toggle_lexicon lex in
    try do
      { display_title font
      ; let word = code entry_tr in
        let entry_VH = Canon.decode word in (* ugly detour via VH string *)
                        (* will be avoided by unique name lookup *)
        let entry = resolve_homonym entry_VH in (* compute homonymy index *)
        let link =
          if in_lexicon entry then Morpho_html.skt_anchor False font entry
              (* We should check it is indeed a substantive entry and that Any is used for
deictics/numbers (TODO) *)
          else let root = if source = "" then "?" (* unknown in lexicon *)
                          else "␣from␣" ^
                if in_lexicon source then Morpho_html.skt_anchor False font source
                else doubt (Morpho_html.skt_roma source) in
                Morpho_html.skt_roma entry ^ root in
        let subtitle = hyperlink_title font link in do
        { display_subtitle (h1_center subtitle)
        ; try look_up font entry (Nouns.Gender gender) part
```

```
          with [ Stream.Error s → failwith s ]
        }
      ; page_end lang True
      }
    with [ Stream.Error _ →
             abort lang ("Illegal␣" ^ translit ^ "␣transliteration␣") entry_tr ]
  }
;
value safe_engine () =
  let abor = abort default_language in
  try decls_engine () with
  [ Sys_error s → abor Control.sys_err_mess s (* file pb *)
  | Stream.Error s → abor Control.stream_err_mess s (* file pb *)
  | Invalid_argument s → abor Control.fatal_err_mess s (* sub *)
  | Failure s → abor Control.fatal_err_mess s (* anomaly *)
  | Control.Fatal s → abor Control.fatal_err_mess s (* anomaly *)
  | Not_found → abor Control.fatal_err_mess "assoc" (* assoc *)
  | Control.Anomaly s → abor Control.fatal_err_mess ("Anomaly:␣" ^ s)
  | Nouns.Report s → abor "Gender␣anomaly␣-␣" s
  | End_of_file → abor Control.fatal_err_mess "EOF" (* EOF *)
  | Encode.In_error s → abor "Wrong_input␣" s
  | Exit → abor "Wrong␣character␣in␣input␣-␣" "use␣ASCII" (* Sanskrit *)
  | _ → abor Control.fatal_err_mess "anomaly" (* ? *)
  ]
;
safe_engine ()
;
```

Module Conjugation

CGI-bin conjugation for computing root conjugations.
This CGI is triggered by page *grammar_page* in *dico_dir*.
Reads its input in shell variable *QUERY_STRING* URI-encoded.
Reads its grammatical information from *public_roots_infos_file*
Prints an html document of root conjugations on *stdout*.

```
open Skt_morph;
open Morphology; (* inflected Verb_form etc. *)
open Conj_infos; (* vmorph Causa Inten Desid root_infos *)
open Inflected; (* roots.val indecls.val etc. *)
```

```
open Html;
open Web; (* ps pl etc. *)
open Cgi;
open Multilingual; (* font gentense tense_name Deva Roma captions *)
value ctitle font = h1_title (conjugation_title narrow_screen font)
and meta_title = title "Sanskrit␣Grammarian␣Conjugation␣Engine"
and back_ground = background Chamois
(* obs if Install.narrow_screen then background Mauve else Pict_gan *)
and hyperlink_title font link =
  if narrow_screen then link
  else conjugation_caption font ^ "␣" ^ link
;
exception Wrong of string
;
```

For non-unicode compliant browsers replace Canon.uniromcode by Canon.decode

```
value pr code =
  ps (html_red (Canon.uniromcode code) ^ "␣") (* roman with diacritics *)
and pr_deva code =
  ps (html_devared (Canon.unidevcode code) ^ "␣") (* devanagari *)
;
value pr_f font word =
  let code = Morpho_html.final word in (* visarga correction *)
  match font with
  [ Deva → pr_deva code
  | Roma → pr code
  ]
;
value prlist_font font =
  let pr = pr_f font
  and bar = html_green "␣|␣" in
  prlistrec
     where rec prlistrec = fun
       [ [] → ()
       | [ x ] → pr x
       | [ x :: l ] → do { pr x; ps bar; prlistrec l }
       ]
;
value persons_of decls =
  let reorg (one, two, three) (p, form) = match p with
```

```
      [ First  →  ([ form  ::  one ], two, three)
      | Second  →  (one, [ form  ::  two ], three)
      | Third  →  (one, two, [ form  ::  three ])
      ]
  and init = ([ ], [ ], [ ]) in
  List.fold_left reorg init decls (* (one,two,three) *)
;
value numbers_of l =
  let reorg (sg, du, pl) (n, p, form) = match n with
       [ Singular  →  ([ (p, form)  ::  sg ], du, pl)
       | Dual  →  (sg, [ (p, form)  ::  du ], pl)
       | Plural  →  (sg, du, [ (p, form)  ::  pl ])
       ]
   and init = ([ ], [ ], [ ]) in
   List.fold_left reorg init l
;
value acell display s = do
  { ps th_begin
  ; display s
  ; ps th_end
  }
;
value print_row1 caption s d p = do
  { ps tr_begin
  ; acell ps caption
  ; acell ps s
  ; acell ps d
  ; acell ps p
  ; pl tr_end
  }
and print_row_font font caption s d p =
  let prlist = prlist_font font in do
  { ps (tr_mouse_begin (color Light_blue) (color Pale_yellow))
  ; acell ps caption
  ; acell prlist s
  ; acell prlist d
  ; acell prlist p
  ; pl tr_end
  }
;
```

```
value display font ovoice l =
  let (s, d, p) = numbers_of l in
  let (f1, s1, t1) = persons_of s
  and (f2, s2, t2) = persons_of d
  and (f3, s3, t3) = persons_of p
  and caption = voice_name ovoice font
  and print_row = print_row_font font in do
      { pl html_break
      ; pl (table_begin_style Inflexion [ ("border","2"); padding5 ])
      ; let sing = number_caption Singular font
        and dual = number_caption Dual font
        and plur = number_caption Plural font in
        print_row1 caption sing dual plur
      ; match font with
        [ Deva → do (* Indian style *)
            { print_row (person_name Third Deva) t1 t2 t3
            ; print_row (person_name Second Deva) s1 s2 s3
            ; print_row (person_name First Deva) f1 f2 f3
            }
        | Roma → do (* Western style *)
            { print_row (person_name First Roma) f1 f2 f3
            ; print_row (person_name Second Roma) s1 s2 s3
            ; print_row (person_name Third Roma) t1 t2 t3
            }
        ]
      ; ps table_end
      ; pl html_break
      }
;
value display_table font ovoice = fun
  [ [] → ()
  | l → do { ps th_begin; display font ovoice l; ps th_end }
  ]
;
value print_caption font tense = ps (tense_name tense font)
;
value display_amp font otense da dm dp = do
  { pl (table_begin (centered Mauve))
  ; ps tr_begin
  ; ps th_begin
```

```
  ; Gen.optional (print_caption font) otense
  ; pl (xml_begin "table")
  ; ps tr_begin
  ; display_table font Active da
  ; display_table font Middle dm
  ; display_table font Passive dp
  ; pl tr_end
  ; pl table_end
  ; ps th_end
  ; pl tr_end
  ; pl table_end (* Mauve *)
  }
and display_perfut font pfa = do
  { pl (table_begin_style (centered Mauve) [])
  ; ps tr_begin
  ; ps th_begin
  ; ps (perfut_name font)
  ; pl (xml_begin "table")
  ; ps tr_begin
  ; display_table font Active pfa
  ; pl tr_end
  ; pl table_end
  ; ps th_end
  ; pl tr_end
  ; pl table_end (* Mauve *)
  }
;
value sort_out_v accu form = fun
  [ [ (_ (* delta *),morphs) ] → List.fold_left reorg accu morphs
      where reorg (pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, e
        [ Verb_form (_(* conj *),te) n p → let t = (n, p, form) in match te with
          [ Presenta _ Present →
    ([ t :: pa ], pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c
          | Presentm _ Present →
    (pa, [ t :: pm ], ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c
          | Presenta _ Imperfect →
    (pa, pm, [ t :: ia ], im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c
          | Presentm _ Imperfect →
    (pa, pm, ia, [ t :: im ], oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c
          | Presenta _ Optative →
```

(*pa, pm, ia, im,* [*t* :: *oa*], *om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Presentm _ Optative* →
(*pa, pm, ia, im, oa,* [*t* :: *om*], *ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Presenta _ Imperative* →
(*pa, pm, ia, im, oa, om,* [*t* :: *ea*], *em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Presentm _ Imperative* →
(*pa, pm, ia, im, oa, om, ea,* [*t* :: *em*], *fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug Future Active* →
(*pa, pm, ia, im, oa, om, ea, em,* [*t* :: *fa*], *fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug Future Middle* →
(*pa, pm, ia, im, oa, om, ea, em, fa,* [*t* :: *fm*], *pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug Perfect Active* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm,* [*t* :: *pfa*], *pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug Perfect Middle* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa,* [*t* :: *pfm*], *aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug* (*Aorist _*) *Active* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm,* [*t* :: *aa*], *am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug* (*Aorist _*) *Middle* | *Conjug* (*Aorist _*) *Passive* → (* passive-middle *)
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa,* [*t* :: *am*], *ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug* (*Injunctive _*) *Active* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am,* [*t* :: *ja*], *jm, ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug* (*Injunctive _*) *Middle* | *Conjug* (*Injunctive _*) *Passive* → (* passive-middle *)
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja,* [*t* :: *jm*], *ba, bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug Benedictive Active* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm,* [*t* :: *ba*], *bm, fpa, ps, ip, op, ep, ca, c*
| *Conjug Benedictive Middle* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba,* [*t* :: *bm*], *fpa, ps, ip, op, ep, ca, c*
| *Perfut Active* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm,* [*t* :: *fpa*], *ps, ip, op, ep, ca, c*
| *Presentp Present* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa,* [*t* :: *ps*], *ip, op, ep, ca, c*
| *Presentp Imperfect* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps,* [*t* :: *ip*], *op, ep, ca, c*
| *Presentp Optative* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip,* [*t* :: *op*], *ep, ca, c*
| *Presentp Imperative* →
(*pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op,* [*t* :: *ep*], *ca, c*
| *Conjug Conditional Active* →

```
      (pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, [ t :: ca ], c
          | Conjug Conditional Middle →
      (pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, [ t :: cr
          | _ → failwith "Unknown paradigm"
          ]
        | _ → raise (Control.Fatal "Unexpected verbal form")
        ]
  | _ → raise (Control.Fatal "Weird inverse map V")
  ]
and init_v = ([], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [])
;
value display_tense3 font tense la lm lp =
  if la = [] ∧ lm = [] ∧ lp = [] then ()
  else match target with
      [ Simputer → do
        { if la = [] then () else display_amp font (Some tense) la [] []
        ; let caption = if la = [] then Some tense else None in
          if lm = [] then () else display_amp font caption [] lm []
        ; let caption = if la = [] ∧ lm = [] then Some tense else None in
          if lp = [] then () else display_amp font caption [] [] lp
        }
      | _ → display_amp font (Some tense) la lm lp
      ]
and display_tense2 font tense la lm =
  if la = [] ∧ lm = [] then ()
  else match target with
      [ Simputer → do
        { if la = [] then () else display_amp font (Some tense) la [] []
        ; let caption = if la = [] then Some tense else None in
          if lm = [] then () else display_amp font caption [] lm []
        }
      | _ → display_amp font (Some tense) la lm []
      ]
;
value display_conjug font conj = do
  { pl html_paragraph
  ; pl (table_begin (centered Cyan))
  ; ps tr_begin
  ; ps th_begin
  ; ps (conjugation_name conj font)
```

```
  ; ps th_end
  ; ps tr_end
  ; pl table_end (* Cyan *)
  ; pl html_paragraph
  }
and display_title font = do
  { pl html_paragraph
  ; pl (table_begin (centered Mauve))
  ; ps tr_begin
  ; ps th_begin
  ; ps (ctitle font)
  ; ps th_end
  ; ps tr_end
  ; pl table_end (* Mauve *)
  ; pl html_paragraph
  }
and display_subtitle title = do
  { pl html_paragraph
  ; pl (table_begin (centered Deep_sky))
  ; ps tr_begin
  ; ps th_begin
  ; ps title
  ; ps th_end
  ; ps tr_end
  ; pl table_end (* Centered *)
  ; pl html_paragraph
  }
;
value display_inflected_v font
      (pa, pm, ia, im, oa, om, ea, em, fa, fm, pfa, pfm, aa, am, ja, jm, ba, bm, fpa, ps, ip, op, ep, ca, cm) =
 { pl center_begin
 ; let tense = Present_tense Present in
   display_tense3 font tense pa pm ps
 ; if ia = [] ∧ im = [] ∧ ip = [] then () else do
     { pl html_break; let tense = Present_tense Imperfect in
                         display_tense3 font tense ia im ip }
 ; if oa = [] ∧ om = [] ∧ op = [] then () else do
     { pl html_break; let tense = Present_tense Optative in
                         display_tense3 font tense oa om op }
 ; if ea = [] ∧ em = [] ∧ ep = [] then () else do
```

```
    { pl html_break; let tense = Present_tense Imperative in
                         display_tense3 font tense ea em ep }
 ; if fa = [] ∧ fm = [] then () else do
    { pl html_break; let tense = Other_tense Future in
                         display_tense2 font tense fa fm }
 ; if ca = [] ∧ cm = [] then () else do
    { pl html_break; let tense = Other_tense Conditional in
                         display_tense2 font tense ca cm }
 ; if fpa = [] then () else do
    { pl html_break; display_perfut font fpa }
 ; if pfa = [] ∧ pfm = [] then () else do
    { pl html_break; let tense = Other_tense Perfect in
                         display_tense2 font tense pfa pfm }
 ; if aa = [] ∧ am = [] then () else do
    { pl html_break; let tense = Other_tense (Aorist 0) in (* forget class *)
                         display_tense2 font tense aa am }
 ; if ja = [] ∧ jm = [] then () else do
    { pl html_break; let tense = Other_tense (Injunctive 0) in (* forget class *)
                         display_tense2 font tense ja jm }
 ; if ba = [] ∧ bm = [] then () else do
    { pl html_break; let tense = Other_tense Benedictive in
                         display_tense2 font tense ba bm }
 ; pl center_end
 ; pl html_paragraph
 }
;
value display_ind ind font = List.iter disp
  where disp (_conj, f) = do
  { ps (h3_begin B3)
  ; ps ind
  ; pl html_break
  ; pr_f font f
  ; pl html_break
  ; ps h3_end
  }
;
value display_inflected_u font inf absya per abstva = do
 { pl center_begin
 ; display_ind (infinitive_caption font) font inf
 ; display_ind (absolutive_caption True font) font abstva
```

```
 ; display_ind (absolutive_caption False font) font (List.map prefix_dash absya)
    where prefix_dash (c, w) = (c, [ 0 :: w ])
    (* NB will display twice absol in -am *)
 ; display_ind (peripft_caption font) font per
 ; pl center_end
 }
;
value encode_part = fun
  [ Ppp -> "Ppp"
  | Pppa -> "Pppa"
  | Ppra _ -> "Ppra"
  | Pprm _ -> "Pprm"
  | Pprp -> "Pprp"
  | Ppfta -> "Ppfta"
  | Ppftm -> "Ppftm"
  | Pfuta -> "Pfuta"
  | Pfutm -> "Pfutm"
  | Pfutp _ -> "Pfutp"
  | Action_noun -> "Act"
  ]
;
(* inspired from Print_html.decl_url *)
value decl_url g s f r part =
  let (gen, link) = match g with
      [ Mas -> ("Mas","m.")
      | Neu -> ("Neu","n.")
      | Fem -> ("Fem","f.")
      | _ -> failwith "Unexpected deictic"
      ] in
  let invoke = decls_cgi ^ "?q=" ^ (Transduction.encode_url s)
      ^ ";g=" ^ gen ^ ";font=" ^ f ^ ";r=" ^ (Transduction.encode_url r)
      ^ ";p=" ^ (encode_part part) ^ ";lex=" ^ lexicon_toggle.val (* Keeping the language
*) in
  anchor Red_ invoke link
;
value display_part font entry part stem_mn stem_f =
  let str_mn = Canon.decode stem_mn
  and str_f = Canon.decode stem_f
  and str_font = string_of_font font in do
  { ps (h3_begin B3)
```

```
  ; ps (participle_name part font)
  ; pl html_break
  ; pr_f font stem_mn
  ; ps (decl_url Mas str_mn str_font entry part)
  ; ps "␣"
  ; ps (decl_url Neu str_mn str_font entry part)
  ; ps "␣"
  ; pr_f font stem_f
  ; ps "␣"
  ; ps (decl_url Fem str_f str_font entry part)
  ; ps h3_end
  }
;
value abort_display mess = do
  { ps th_end
  ; ps tr_end
  ; pl table_end (* Mauve *)
  ; pl center_end
  ; failwith mess
  }
;
value look_up_and_display font gana entry =
  let print_conjug conj parts =
  let process_pp = p [] where rec p acc = fun
  [ [] → acc
  | [ x :: rest ] → match x with
      [ Parts.Ppp_ con rstem _ when con = conj → match rstem with
          [ [ 1 :: r ] →
              let sm = List.rev rstem
              and sf = List.rev [ 2 :: r ] in do
              { display_part font entry Ppp sm sf
              ; p acc rest
              }
          | _ → abort_display "Weird␣Ppp"
          ]
      | other → p [ other :: acc ] rest
      ]
  ]
  and process_ppa = p [] where rec p acc = fun
  [ [] → acc
```

```
| [ x :: rest ] → match x with
    [ Parts.Pppa_ con stem _ when con = conj →
      let sm = Parts.fix stem "vat"
      and sf = Parts.fix stem "vatii" in do
      { display_part font entry Pppa sm sf
      ; p acc rest
      }
    | other → p [ other :: acc ] rest
    ]
]
and process_pra = p [] where rec p acc = fun
[ [] → acc
| [ x :: rest ] → match x with
    [ Parts.Ppra_ k con m_stem f_stem _ when con = conj →
      let sm = Parts.fix m_stem "at"
      and sf = Parts.fix f_stem "ii" in do
      { display_part font entry (Ppra k) sm sf
      ; p acc rest
      }
    | Parts.Pprared_ con stem _ when con = conj →
      let k = if con = Intensive then Parts.int_gana else 3 in
      let sm = Parts.fix stem "at"
      and sf = Parts.fix stem "atii" in do
      { display_part font entry (Ppra k) sm sf
      ; p acc rest
      }
    | other → p [ other :: acc ] rest
    ]
]
and process_prm = p [] where rec p acc = fun
[ [] → acc
| [ x :: rest ] → match x with
    [ Parts.Pprm_ k con stem _ when con = conj →
      let sm = List.rev [ 1 :: stem ]
      and sf = List.rev [ 2 :: stem ] in do
      { display_part font entry (Pprm k) sm sf
      ; p acc rest
      }
    | other → p [ other :: acc ] rest
    ]
```

```
]
and process_prp = p [] where rec p acc = fun
[ [] → acc
| [ x :: rest ] → match x with
    [ Parts.Pprp_ con stem _ when con = conj →
      let sm = List.rev [ 1 :: stem ]
      and sf = List.rev [ 2 :: stem ] in do
      { display_part font entry Pprp sm sf
      ; p acc rest
      }
    | other → p [ other :: acc ] rest
    ]
]
and process_pfta = p [] where rec p acc = fun
[ [] → acc
| [ x :: rest ] → match x with
    [ Parts.Ppfta_ con stem _ when con = conj →
      let vstem = if Phonetics.monosyllabic stem then
                     if stem = [ 34; 3; 45 ] (* vid *) then stem (* should test entry
*)
                     else List.rev (Parts.fix stem "i") (* intercalating i *)
                  else stem in
      let sm = Parts.fix vstem "vas"
      and sf = Parts.fix stem "u.sii" in do
      { display_part font entry Ppfta sm sf
      ; if con = Primary ∧
            ( stem = [ 34; 3; 45 ] (* vid *) (* Parts.build_more_ppfa *)
            ∨ stem = [ 46; 3; 45; 3; 45 ] (* vivi's *) (* horrible code *)
            ∨ stem = [ 46; 7; 34; 1; 34 ] (* dad.r's *)
            )
        then let sm = Parts.fix vstem "ivas" in
             display_part font entry Ppfta sm sf
        else ()
      ; p acc rest
      }
    | other → p [ other :: acc ] rest
    ]
]
and process_pftm = p [] where rec p acc = fun
[ [] → acc
```

```
| [ x :: rest ] → match x with
    [ Parts.Ppftm_ con stem _ when con = conj →
      let sm = List.rev [ 1 :: stem ]
      and sf = List.rev [ 2 :: stem ] in do
      { display_part font entry Ppftm sm sf
      ; p acc rest
      }
    | other → p [ other :: acc ] rest
    ]
]
and process_futa = p [] where rec p acc = fun
[ [] → acc
| [ x :: rest ] → match x with
    [ Parts.Pfuta_ con stem _ when con = conj →
      let sm = Parts.fix stem "at"
      and sf = Parts.fix stem "antii" in do
      { display_part font entry Pfuta sm sf
      ; p acc rest
      }
    | other → p [ other :: acc ] rest
    ]
]
and process_futm = p [] where rec p acc = fun
[ [] → acc
| [ x :: rest ] → match x with
    [ Parts.Pfutm_ con stem _ when con = conj →
      let sm = List.rev [ 1 :: stem ]
      and sf = List.rev [ 2 :: stem ] in do
      { display_part font entry Pfutm sm sf
      ; p acc rest
      }
    | other → p [ other :: acc ] rest
    ]
]
and process_pfp = p [] where rec p acc = fun
[ [] → acc
| [ x :: rest ] → match x with
    [ Parts.Pfutp_ con rstem _ when con = conj → match rstem with
        [ [ 1 :: r ] →
          let k = match r with
```

```
            [ [ 42 :: [ 45 :: [ 1 :: [ 32 :: _ ] ] ] ] → 3 (* -tavya *)
            | [ 42 :: [ 4 :: _ ] ] → 2 (* -aniiya *)
            | [ 42 :: _ ] → 1 (* -ya *)
            | _ → failwith ("Weird Pfp: " ^ Canon.rdecode rstem)
            ] in
          let sm = List.rev rstem
          and sf = List.rev [ 2 :: r ] in do
          { display_part font entry (Pfutp k) sm sf
          ; p acc rest
          }
        | _ → failwith "Weird Pfutp"
        ]
    | other → p [ other :: acc ] rest
    ]
]
and sort_out_u accu form = fun
[ [ (_, morphs) ] → List.fold_left (reorg form) accu morphs
    where reorg f (inf, absya, per, abstva) = fun
      [ Ind_verb (c, Infi) when c = conj → ([ (c, f) :: inf ], absya, per, abstva)
      | Ind_verb (c, Absoya) when c = conj → (inf, [ (c, f) :: absya ], per, abstva)
      | Ind_verb (c, Perpft) when c = conj → (inf, absya, [ (c, f) :: per ], abstva)
      | Abs_root c when c = conj → (inf, absya, per, [ (c, f) :: abstva ])
      | _ → (inf, absya, per, abstva)
      ]
| _ → raise (Control.Fatal "Weird inverse map N")
]
and init_u = ([], [], [], [])
and buckets = Deco.fold sort_out_v init_v roots.val in do
(* Main print_conjug *)
    { display_conjug font conj
    ; display_inflected_v font buckets (* Display finite root forms *)
    ; pl html_paragraph
    ; pl center_begin (* Now display participial root forms *)
    ; pl (table_begin_style (centered Mauve) [])
    ; ps tr_begin
    ; ps th_begin
    ; ps (participles_caption font)
    ; let rest = process_pp parts in (* Past Passive *)
      let rest = process_ppa rest in (* Past Active *)
      let rest = process_pra rest in (* Present Active *)
```

```
      let rest = process_prm rest in (* Present Middle *)
      let rest = process_prp rest in (* Present Passive *)
      let rest = process_futa rest in (* Future Active *)
      let rest = process_futm rest in (* Future Middle *)
      let rest = process_pfp rest in (* Future Passive = gerundive *)
      let rest = process_pfta rest in (* Perfect Active *)
      let _ = process_pftm rest in (* Perfect Middle *) do
        { ps th_end
        ; ps tr_end
        ; pl table_end (* Mauve *)
        ; pl center_end
        ; pl html_paragraph (* Now display indeclinable root forms if any *)
        ; let (inf, _, _, abstvaa) = Deco.fold sort_out_u init_u abstvaa.val
          and (_, absya, _, _) = Deco.fold sort_out_u init_u absya.val
          and (_, _, per, _) = Deco.fold sort_out_u init_u peri.val in
          if absya = [] ∧ per = [] ∧ abstvaa = [] then () else do
          (* Display indeclinable forms *)
          { pl center_begin
          ; pl (table_begin_style (centered Mauve) [])
          ; ps tr_begin
          ; ps th_begin
          ; ps (indeclinables_caption font)
          ; display_inflected_u font inf absya per abstvaa
          ; ps th_end
          ; ps tr_end
          ; pl table_end (* Mauve *)
          ; pl center_end
          }
        }
  } (* end print_conjug *) in
 let compute_conjugs = List.iter (Verbs.compute_conjugs_stems entry) in
 let secondary_conjugs infos =
   let cau_filter = fun [ (Causa _, _) → True | _ → False ]
   and int_filter = fun [ (Inten _, _) → True | _ → False ]
   and des_filter = fun [ (Desid _, _) → True | _ → False ] in do
   { let causatives = List.filter cau_filter infos in
     if causatives = [] then () else do
       { roots.val := Deco.empty
       ; compute_conjugs causatives
       ; print_conjug Causative Parts.participles.val
```

```
          }
      ; let intensives = List.filter int_filter infos in
        if intensives = [] then () else do
          { roots.val := Deco.empty
          ; compute_conjugs intensives
          ; print_conjug Intensive Parts.participles.val
          }
      ; let desideratives = List.filter des_filter infos in
        if desideratives = [] then () else do
          { roots.val := Deco.empty
          ; compute_conjugs desideratives
          ; print_conjug Desiderative Parts.participles.val
          }
      } in do
   (* Main look_up_and_display *)
   { Verbs.fake_compute_conjugs gana entry (* builds temporaries roots.val etc *)
   ; let infos = (* should be a call to a service that gives one entry_infos *)
     (Gen.gobble Data.public_roots_infos_file : Deco.deco root_infos) in
     let entry_infos = Deco.assoc (Encode.code_string entry) infos in
     do { print_conjug Primary Parts.participles.val
        ; secondary_conjugs entry_infos
        }
   }
;
value in_lexicon entry = (* entry as a string in VH transliteration *)
  Index.is_in_lexicon (Encode.code_string entry)
(* Problem: may give link to a non-root entry if called from Grammar service *)
and doubt s = "?" ^ s
;
(* Compute homonym index for a given present class. *)
(* This is very fragile: lexicon update induces code adaptation. *)
(* Temporary - should be subsumed by unique naming structure. *)
value resolve_homonym entry =
  let first e = e ^ "#1"
  and second e = e ^ "#2"
  and third e = e ^ "#3"
  and fourth e = e ^ "#4" in fun
  [ 1 → match entry with
    [ ".rc"
    | "krudh"
```

```
| "kha~nj"
| "cit"
| "chad"
| "tyaj"
| "tvi.s"
| "dah"
| "daa" (* ambiguous with "daa#3" *)
| "diz"
| "d.rz"
| "dyut"
| "dru"
| "dhaa"
| "dhaav" (* ambiguous with "dhaav#2" *)
| "nii"
| "pat"
| "paa" (* ambiguous with "paa#2" *)
| "budh"
| "b.rh"
| "bhuu"
| "m.rd"
| "mud"
| "yaj"
| "yat"
| "raaj"
| "ruc"
| "rud"
| "rudh"
| "vas"
| "vah"
| "v.r"
| "v.rdh"
| "vi.s"
| "zii"
| "zuc"
| "zubh"
| "zcut"
| "sad"
| "sah"
| "suu"
| "sthaa"
```

```
    | "snih"
    | "spaz" (* no present *)
    | "h.r" → first entry
    | "gaa"
    | "yu"
    | "vap" (* ambiguous with vap#1 *)
    | "sidh"
    | "svid" → second entry
    | "maa" → fourth entry
    | "arc" → ".rc#1" (* link - bizarre *)
    | _ → entry
    ]
  | 2 → match entry with
    [ "ad"
    | "as"
    | "iiz"
    | "duh"
    | "draa" (* ambiguous with "draa#2" *)
    | "dvi.s"
    | "praa"
    | "praa.n"
    | "bhaa"
    | "maa"
    | "yaa"
    | "yu"
    | "raa"
    | "rud"
    | "lih"
    | "vid" (* ambiguous with "vid#2" *)
    | "vii"
    | "zii"
    | "zvas"
    | "suu"
    | "han" → first entry
    | "an"
    | "aas"
    | "daa"
    | "paa"
    | "vas"
    | "vaa" → second entry
```

```
    | "nii" → third entry
    | _ → entry
    ]
  | 3 → match entry with
    [ "gaa"
    | "daa"
    | "dhaa"
    | "p.r"
    | "bhii"
    | "maa" (* ambiguous with "maa#3" *)
    | "vi.s"
    | "haa" → first entry (* ambiguous with "haa#2" used in middle *)
    | "yu" → second entry
    | _ → entry
    ]
  | 4 → match entry with
    [ "k.sudh"
    | "dam"
    | "diiv"
    | "d.rz"
    | "druh"
    | "dhii"
    | "naz"
    | "pad"
    | "pu.s"
    | "budh"
    | "mad"
    | "yudh"
    | "zam"
    | "saa"
    | "sidh"
    | "snih"
    | "snuh" → first entry
    | "div" → first "diiv" (* since MW spells div *)
    | "as"
    | "i.s"
    | "tan"
    | "daa"
    | "draa"
    | "dhaa"
```

```
    | "pat"
    | "svid" → second entry
    | "vaa" → third entry
    | _ → entry
    ]
  | 5 → match entry with
    [ "az"
    | "k.r"
    | "dhuu"
    | "v.r" → first entry
    | "p.r"
    | "su"
    | "hi" → second entry
    | _ → entry
    ]
  | 6 → match entry with
    [ "i.s"
    | "k.rt"
    | "g.rr"
    | "tud"
    | "diz"
    | "d.r"
    | "pi"
    | "bhuj"
    | "muc"
    | "yu"
    | "rud"
    | "viz"
    | "suu"
    | "s.rj"
    | "sp.rz"
      → first entry
    | "p.r"
    | "b.rh"
    | "rudh"
    | "vid" (* ambiguous with "vid#1" *)
      → second entry
    | "vas" → fourth entry
    | _ → entry
    ]
```

```
| 7 → match entry with
  [ "chid"
  | "bhid"
  | "yuj" → first entry
  | "bhuj"
  | "rudh" → second entry
  | _ → entry
  ]
| 8 → match entry with
  [ "k.r"
  | "tan"
  | "san" → first entry
  | _ → entry
  ]
| 9 → match entry with
  [ "az"
  | "g.rr"
  | "j~naa"
  | "jyaa"
  | "puu"
  | "m.rd"
  | "luu" → first entry
  | "v.r"
  | "h.r" → second entry
  | _ → entry
  ]
| 10 → entry
| 11 → match entry with
  [ "mahii" → second entry
  | _ → entry
  ]
 | 0 → if in_lexicon entry (* ad-hoc disambiguation for secondary conjugs *)
          then entry
        else let fentry = first entry in
             if in_lexicon fentry then fentry else raise (Wrong entry)
| _ → ""
]
;
value conjs_engine () = do
  { pl http_header
```

```
; page_begin meta_title
; pl (body_begin back_ground)
; let query = Sys.getenv "QUERY_STRING" in
  let env = create_env query in
 try
  let url_encoded_entry = List.assoc "q" env
  and url_encoded_class = List.assoc "c" env
  and font = font_of_string (get "font" env Paths.default_display_font)
  (* OBS and stamp = get "v" env "" *)
  and translit = get "t" env "VH" (* DICO created in VH trans *)
  and lex = get "lex" env "SH" (* default Heritage *) in
  let entry_tr = decode_url url_encoded_entry (* : string in translit *)
  and lang = language_of lex
  and gana = match decode_url url_encoded_class with
     [ "1" → 1
     | "2" → 2
     | "3" → 3
     | "4" → 4
     | "5" → 5
     | "6" → 6
     | "7" → 7
     | "8" → 8
     | "9" → 9
     | "10" → 10
     | "11" → 11 (* denominative verbs *)
(* — "0" -¿ 0 (* secondary conjugations *) – obsolete *)
     | s → raise (Control.Fatal ("Weird␣present␣class:␣" ^ s))
     ]
  and encoding_function = Encode.switch_code translit
  and () = toggle_lexicon lex in
  try let word = encoding_function entry_tr in
      let entry_VH = Canon.decode word in (* ugly detour via VH string *)
      (* Beware - 46 decodes as "z" and 21 as "f" *)
      let entry = resolve_homonym entry_VH gana in (* VH string with homo *)
      let known = in_lexicon entry (* in lexicon? *)
         (* we should check it is indeed a root or denominative *) in do
      { display_title font
      ; let link = if known then Morpho_html.skt_anchor False font entry
                         else doubt (Morpho_html.skt_roma entry) in
        let subtitle = hyperlink_title font link in
```

```
          display_subtitle (h1_center subtitle)
        ; try look_up_and_display font gana entry
          with [ Stream.Error s -> raise (Wrong s) ]
        ; page_end lang True
        }
     with [ Stream.Error _ ->
               abort lang ("Illegal " ^ translit ^ " transliteration ") entry_tr ]
   with [ Not_found -> failwith "parameters q or c missing" ]
  }
;
value safe_engine () =
  let abor = abort default_language in
  try conjs_engine () with
  [ Sys_error s -> abor Control.sys_err_mess s (* file pb *)
  | Stream.Error s -> abor Control.stream_err_mess s (* file pb *)
  | Invalid_argument s -> abor Control.fatal_err_mess s (* sub *)
  | Wrong s -> abor "Error - wrong root or class ? - " s
  | Failure s -> abor "Wrong_input ? " s
  | Control.Fatal s -> abor Control.fatal_err_mess s (* anomaly *)
  | Not_found -> abor Control.fatal_err_mess "assoc" (* assoc *)
  | Control.Anomaly s -> abor Control.fatal_err_mess ("Anomaly: " ^ s)
  | End_of_file -> abor Control.fatal_err_mess "EOF" (* EOF *)
  | Encode.In_error s -> abor "Wrong_input " s
  | Exit (* Sanskrit *) -> abor "Wrong character in input - " "use ASCII"
  | _ -> abor Control.fatal_err_mess "anomaly" (* ? *)
  ]
;
safe_engine () (* Should always produce a legal xhtml page *)
;
```

Module Indexer

CGI-bin indexer for indexing in sanskrit dictionary.
This CGI is triggered by page *index.html* in *dico_dir*.
Reads its input in shell variable *QUERY_STRING* URI-encoded.

```
open Html; (* table_begin etc. *)
open Web; (* ps pl abort etc. *)
open Cgi;
```

```
value answer_begin () = do
  { pl (table_begin Yellow_cent)
  ; ps tr_begin
  ; ps th_begin
  }
;
value answer_end () = do
  { ps th_end
  ; ps tr_end
  ; pl table_end
  ; pl html_paragraph
  }
;
value ok (mess, s) = do { ps mess; pl (Morpho_html.skt_anchor_R False s) }
 and ok2 (mess, s1, s2) = do { ps mess; pl (Morpho_html.skt_anchor_R2 s1 s2) }
```

(∗ ok2 prints the entry under the spelling given by the user, i.e. without normalisation, thus e.g. sandhi is not written sa.mdhi, and possibly suffixed by homonymy index 1, e.g. b.rh. ∗)

```
;
(* Should share Lemmatizer.load_inflected *)
value load_inflected file = (Gen.gobble file : Morphology.inflected_map)
;
value load_nouns () = load_inflected Data.public_nouns_file
and load_roots () = load_inflected Data.public_roots_file
and load_vocas () = load_inflected Data.public_vocas_file
and load_indecls () = load_inflected Data.public_inde_file
and load_parts () = load_inflected Data.public_parts_file
;
value back_ground = background Chamois
;
value display word l = do
  { ps " found as inflected form:"
  ; pl html_break
  ; let pi inv = Morpho_html.print_inflected False word inv in
    List.iter pi l
  }
and report_failure s = do
  { ps " not found in dictionary"
  ; pl html_break
  ; ps "Closest entry in lexical order: "
```

```
  ; ps (Morpho_html.skt_anchor_R False s)
  ; pl html_break
  }
;
value try_declensions word before =
  (* before is last lexical item before word in lexical order *)
  (* This is costly because of the size of inverted inflected databases *)
  let inflectedn = load_nouns () in
  match Deco.assoc word inflectedn with
  [ [] → (* Not found; we try vocative forms *)
    let inflectedv = load_vocas () in
    match Deco.assoc word inflectedv with
    [ [] → (* Not found; we try root forms *)
      let inflectedr = load_roots () in
      match Deco.assoc word inflectedr with
      [ [] → (* Not found; we try adverbial forms *)
        let inflecteda = load_indecls () in
        match Deco.assoc word inflecteda with
        [ [] → report_failure before
          (* NB - no look-up in parts forms since big and partly lexicalized *)
        | l → display word l
        ]
      | l → display word l
      ]
    | l → display word l
    ]
  | l → display word l
  ]
;
value print_word_unique word (entry, lex, page) = (* lex="other" allowed *)
  let link = Morpho_html.skt_anchor_M word entry page False in
  pl (link ^ "␣[␣" ^ lex ^ "␣]" ^ xml_empty "br")
  (* this allows access to a pseudo-entry such as "hvaaya" *)
;
value print_word word (entry, lex, page) = match lex with
  [ "other" → ()
  | _ → print_word_unique word (entry, lex, page)
  ]
;
value read_mw_index () =
```

```
  (Gen.gobble Data.public_mw_index_file : Deco.deco (string × string × string))
;
value index_engine () = do
  { pl http_header
  ; page_begin heritage_dictionary_title
  ; pl (body_begin back_ground)
  ; let query = Sys.getenv "QUERY_STRING" in
  let env = create_env query in
  let translit = get "t" env Paths.default_transliteration
  and lex = get "lex" env Paths.default_lexicon (* default by config *)
  and url_encoded_entry = get "q" env "" in
  let lang = language_of lex in do
  { print_title_solid Mauve (Some lang) (dico_title lang)
  ; let str = decode_url url_encoded_entry (* in translit *)
    and encode = Encode.switch_code translit
    and () = toggle_lexicon lex in
    try let word = encode str (* normalization *) in
        let str_VH = Canon.decode word in do
        { answer_begin ()
        ; ps (div_begin Latin12)
        ; match lex with
          [ "MW" →
            let mw_index = read_mw_index () in
            let words = Deco.assoc word mw_index in
            match words with
              [ [] → do { ps (Morpho_html.skt_red str_VH)
                        ; ps " not found in MW dictionary"
                        ; pl html_break
                        }
              | [ unique ] → print_word_unique str_VH unique
              | _ → List.iter (print_word str_VH) (List.rev words)
              ]
          | "SH" → do (* richer search engine *)
            { let sh_index = Index.read_entries () in
              try let (s, b, h) = Index.search word sh_index in
                  if b ∨ h then
                     let r = Canon.decode word in
                     let hr = if h then r ^ "_1" else r in
                     ok2 ("Entry found: ",s, hr)
                  else ok ("First matching entry: ",s)
```

```
                    (* remark that s may be str with some suffix, *)
                    (* even though str may exist as inflected form *)
              with (* Matching entry not found - we try declensions *)
                  [ Index.Last last → do
                      { ps (Morpho_html.skt_red str_VH)
                      ; try_declensions word last
                      }
                  ]
             }
          | _ → failwith "Unknown␣lexicon"
          ]
        ; ps div_end (* Latin12 *)
        ; answer_end ()
        ; page_end lang True
        } (* do *)
    with [ Stream.Error _ → abort lang "Illegal␣transliteration␣" str ]
  } (* do *)
  } (* do *)
;
value safe_index_engine () =
  let abor = abort Html.French (* may not preserve the current language *) in
  try index_engine () with
  [ Sys_error s → abor Control.sys_err_mess s (* file pb *)
  | Stream.Error s → abor Control.stream_err_mess s (* file pb *)
  | Invalid_argument s → abor Control.fatal_err_mess s (* sub *)
  | Failure s → abor Control.fatal_err_mess s (* anomaly *)
  | Control.Fatal s → abor Control.fatal_err_mess s (* anomaly *)
  | Not_found → abor Control.fatal_err_mess "assoc" (* assoc *)
  | End_of_file → abor Control.fatal_err_mess "EOF" (* EOF *)
  | Encode.In_error s → abor "Wrong_input␣" s
  | Exit → abor "Wrong␣character␣in␣input␣-␣" "use␣ASCII" (* Sanskrit *)
  | _ → abor Control.fatal_err_mess "Unexpected␣anomaly" (* ? *)
  ]
;
(* typical invocation is http://skt_server_url/cgi-bin/sktindex?t=VH∧lex=SH∧q=input *)
safe_index_engine ()
;
```

Module Indexerd

CGI-bin indexerd for indexing in sanskrit dico without diacritics.
This CGI is triggered by page *index.html* in *dico_dir*.
Reads its input in shell variable *QUERY_STRING* URI-encoded.

```
open Html;
open Web; (* ps pl etc. *)
open Cgi;

value answer_begin () = do
  { pl (table_begin Yellow_cent)
  ; ps tr_begin
  ; ps th_begin
  }
;
value answer_end () = do
  { ps th_end
  ; ps tr_end
  ; pl table_end
  ; pl html_paragraph
  }
;
value back_ground = background Chamois
;
value prelude () = do
  { pl http_header
  ; page_begin heritage_dictionary_title
  ; pl (body_begin back_ground)
  ; pl html_paragraph
  ; print_title_solid Mauve (Some Html.French) dico_title_fr
  }
;
value postlude () = do
  { ()
  ; page_end Html.French True
  }
;
value print_word c = pl (Morpho_html.skt_anchor_R False (Canon.decode_ref c))
;
(* Each dummy is mapped to a list of words - all the words which give back the dummy by
```

```
normalisation such as removing diacritics *)
value read_dummies () =
  (Gen.gobble Data.public_dummies_file : Deco.deco Word.word)
;
value index_engine () =
  let abor = abort Html.French (* may not preserve the current lang *) in
  try let dummies_deco = read_dummies () in do
      { prelude ()
      ; let query = Sys.getenv "QUERY_STRING" in
        let alist = create_env query in
        (* We do not assume transliteration, just ordinary roman letters *)
        (* TODO: adapt to MW search along Indexer *)
        let url_encoded_entry = List.assoc "q" alist in
        let str = decode_url url_encoded_entry in
        try let dummy = Encode.code_skt_ref_d str (* normalization *) in do
              { answer_begin ()
              ; ps (div_begin Latin12)
              ; let words = Deco.assoc dummy dummies_deco in
                match words with
                  [ [] → do { ps (Morpho_html.skt_red str)
                            ; ps "␣not␣found␣in␣Heritage␣dictionary"
                            ; ps html_break; pl html_break
                            }
                  | _ → List.iter print_word words
                  ]
              ; ps div_end (* Latin12 *)
              ; answer_end ()
              ; postlude ()
              }
        with [ Stream.Error _ → abor "Illegal␣input␣" str ]
      }
   with
   [ Sys_error s → abor Control.sys_err_mess s (* file pb *)
   | Stream.Error s → abor Control.stream_err_mess s (* file pb *)
   | Invalid_argument s → abor Control.fatal_err_mess s (* sub *)
   | Failure s → abor Control.fatal_err_mess s (* anomaly *)
   | Control.Fatal s → abor Control.fatal_err_mess s (* anomaly *)
   | Not_found → abor Control.fatal_err_mess "assoc" (* assoc *)
   | End_of_file → abor Control.fatal_err_mess "EOF" (* EOF *)
   | Encode.In_error s → abor "Wrong_input␣" s
```

```
  | Exit → abor "Wrong character in input - " "use ASCII" (* Sanskrit *)
  | _ → abor Control.fatal_err_mess "Unexpected anomaly" (* ? *)
  ]
;
index_engine ()
;
```

Module Phases

```
module Phases = struct
(* Lexical sorts as phases, i.e. states of the modular transducer *)
type phase =
  [ Noun | Noun2
  | Pron
  | Root
  | Inde (* indeclinable forms *)
  | Absv (* vowel-initial abs-tvaa *)
  | Absc (* consonant-initial abs-tvaa *)
  | Abso (* abs in -ya *)
  | Voca (* vocatives *)
  | Inv (* invocations *)
  | Iic | Iic2 (* first part of compounds *)
  | Iiif (* iic of ifc, atteinable from previous iic eg -vartin iic -varti- *)
  | Iiv | Iivv | Iivc (* inchoatives - cvi verbal compounds *)
  | Auxi | Auxiinv | Auxik | Auxiick (* forms of auxiliary verbs as bhuu k.r *)
  | Ifc | Ifc2 (* second part of compounds *)
  | Peri (* periphrastic perfect *)
  | Lopa (* e/o conjugated root forms with lopa *)
  | Lopak (* e/o kridantas forms with lopa *)
  | Pv (* Preverb optional before Root or Lopa or mandatory before Abso *)
  | Pvc | Pvv (* privative Abso *)
  | Pvkc | Pvkv (* Preverb optional before Krid or Iik or Lopak *)
  | A | An (* privative nan-compounds formations in a- or -an *)
  | Ai | Ani (* initial privative nan-compounds *)
  | Iicv | Iicc (* split of Iic by first letter resp. vowel or consonant *)
  | Ifcv | Ifcc (* idem for Ifc *)
  | Nouv | Nouc (* idem for Noun *)
  | Krid (* Kridantas eg participles *)
  | Vok (* Kridanta vocatives *)
```

```
  | Iik (* Kridanta iics *)
  | Iikv | Iikc | Kriv | Kric | Vocv | Vocc | Vokv | Vokc
  | Iiy | Avy (* Avyayiibhaavas *)
  | Inftu | Kama (* vaktukaama cpds *)
  | Cache | Cachei (* Lexicon acquisition *)
  | Unknown (* Unrecognized chunk *)
  (* now pseudo phase tagging root/kridanta forms with preverbs *)
  | Comp of tag and (* pv *) Word.word and (* root/krid in tag *) Word.word
  ]
and tag = (phase × phase) (* preverb phase and root/taddhita phase *)
(* NB. It is essential to keep both phases to identify transition checkpoints *)
and phases = list phase
(* NB. In Simplified mode, we use only 10 phases: [ Noun2; Pron; Iic2; Ifc2; Root; Inde; Pv; Iiv; Abso
*)
;
(* Marshalling for cgi invocations *)
value rec string_of_phase = fun
  [ Noun → "Noun"
  | Noun2 → "Noun2"
  | Pron → "Pron"
  | Root → "Root"
  | Inde → "Inde"
  | Absv → "Absv"
  | Absc → "Absc"
  | Abso → "Abso"
  | Voca → "Voca"
  | Inv → "Inv"
  | Iic → "Iic"
  | Iic2 → "Iic2"
  | Iiif → "Iiif"
  | Iiv → "Iiv"
  | Iivv → "Iivv"
  | Iivc → "Iivc"
  | Auxi → "Auxi"
  | Auxiinv → "Auxiinv"
  | Auxik → "Auxik"
  | Auxiick → "Auxiick"
  | Ifc → "Ifc"
  | Ifc2 → "Ifc2"
  | Lopa → "Lopa"
```

```
  | Lopak → "Lopak"
  | Pv → "Pv"
  | Pvc → "Pvc"
  | Pvv → "Pvv"
  | Pvkc → "Pvkc"
  | Pvkv → "Pvkv"
  | A → "A"
  | An → "An"
  | Ai → "Ai"
  | Ani → "Ani"
  | Iicv → "Iicv"
  | Iicc → "Iicc"
  | Ifcv → "Ifcv"
  | Ifcc → "Ifcc"
  | Nouv → "Nouv"
  | Nouc → "Nouc"
  | Krid → "Krid"
  | Vok → "Vok"
  | Vokv → "Vokv"
  | Vokc → "Vokc"
  | Iik → "Iik"
  | Iikv → "Iikv"
  | Iikc → "Iikc"
  | Iiy → "Iiy"
  | Avy → "Avya"
  | Kriv → "Kriv"
  | Kric → "Kric"
  | Vocv → "Vocv"
  | Vocc → "Vocc"
  | Peri → "Peri"
  | Inftu → "Inftu"
  | Kama → "Kama"
  | Cache → "Cache"
  | Cachei → "Cachei"
  | Unknown → "Unknown"
  | _ → failwith "string_of_phase"
  ]
and phase_of_string = fun (* unsafe *)
  [ "Noun" → Noun
  | "Noun2" → Noun2
```

| "Pron" → *Pron*
| "Root" → *Root*
| "Inde" → *Inde*
| "Abso" → *Abso*
| "Absv" → *Absv*
| "Absc" → *Absc*
| "Voca" → *Voca*
| "Inv" → *Inv*
| "Iic" → *Iic*
| "Iic2" → *Iic2*
| "Iiif" → *Iiif*
| "Iiv" → *Iiv*
| "Iivv" → *Iivv*
| "Iivc" → *Iivc*
| "Auxi" → *Auxi*
| "Auxiinv" → *Auxiinv*
| "Auxik" → *Auxik*
| "Auxiick" → *Auxiick*
| "Ifc" → *Ifc*
| "Ifc2" → *Ifc2*
| "Lopa" → *Lopa*
| "Lopak" → *Lopak*
| "Pv" → *Pv*
| "Pvv" → *Pvv*
| "Pvc" → *Pvc*
| "Pvkc" → *Pvkc*
| "Pvkv" → *Pvkv*
| "A" → *A*
| "An" → *An*
| "Ai" → *Ai*
| "Ani" → *Ani*
| "Iicv" → *Iicv*
| "Iicc" → *Iicc*
| "Ifcv" → *Ifcv*
| "Ifcc" → *Ifcc*
| "Nouv" → *Nouv*
| "Nouc" → *Nouc*
| "Krid" → *Krid*
| "Vokv" → *Vokv*
| "Vokc" → *Vokc*

```
  | "Iik" → Iik
  | "Iikv" → Iikv
  | "Iikc" → Iikc
  | "Iiy" → Iiy
  | "Avya" → Avy
  | "Kriv" → Kriv
  | "Kric" → Kric
  | "Vocv" → Vocv
  | "Vocc" → Vocc
  | "Peri" → Peri
  | "Inftu" → Inftu
  | "Kama" → Kama
  | "Unknown" → Unknown
  | "Cache" → Cache
  | "Cachei" → Cachei
  | s → failwith ("Unknown␣phase␣" ^ s)
  ]
;
value unknown = Unknown
and aa_phase = fun (* phase of preverb "aa" according to following phase *)
    [ Root | Abso | Peri | Inftu → Pv | _ → Pvkv ]
and un_lopa = fun (* phase of origin of lopa *)
    [ Lopa → Root | Lopak → Kriv | _ → failwith "un_lopa" ]
and preverb_phase = fun
    [ Pv | Pvv | Pvc | Pvkc | Pvkv → True | _ → False ]
and krid_phase = fun [ Krid | Kric | Kriv → True | _ → False ]
and ikrid_phase = fun [ Iik | Iikc | Iikv → True | _ → False ]
and vkrid_phase = fun [ Vokc | Vokv → True | _ → False ]
and ii_phase = fun [ Iicv | Iicc | Iikv | Iikc | A | An → True | _ → False ]
and is_cache phase = (phase = Cache) ∨ (phase = Cachei)
;
(* Needed as argument of Morpho.print_inv_morpho *)
value rec generative =
  [ Krid | Kriv | Kric | Vokv | Vokc | Iik | Iikv | Iikc | Auxik → True
  | Comp (_, ph) _ _ → generative ph
  | _ → False
  ]
;

open Html;
value rec color_of_phase = fun
```

```
  [ Noun | Noun2 | Lopak | Nouc | Nouv | Kriv | Kric | Krid | Auxik | Kama
         | Cache → Deep_sky
  | Pron → Light_blue
  | Root | Auxi | Lopa → Carmin
  | Inde | Abso | Absv | Absc | Auxiinv | Ai | Ani → Mauve
  | Iiy → Lavender
  | Avy → Magenta
  | Inftu → Salmon
  | Iic | Iic2 | A | An | Iicv | Iicc | Iik | Iikv | Iikc | Iiif
        | Auxiick | Cachei → Yellow
  | Peri | Iiv | Iivv | Iivc → Orange
  | Voca | Vocv | Vocc | Inv | Vok | Vokv | Vokc → Lawngreen
  | Ifc | Ifcv | Ifcc | Ifc2 → Cyan
  | Unknown → Grey
  | Comp (_, ph) _ _ → color_of_phase ph
  | Pv | Pvv | Pvc | Pvkc | Pvkv → failwith "Illegal preverb segment"
(*| _ → raise (Control.Anomaly "Unexpected color") *)
  ]
;
end; (* Phases *)
```

Module Lemmatizer

CGI-bin lemmatizer for searching the inflected forms databases
This CGI is triggered by page *index_page* in *dico_dir*.
Reads its input in shell variable *QUERY_STRING* URI-encoded.
Prints an HTML document of lemma information on *stdout*.

```
open Html;
open Web; (* ps pl etc. *)
open Cgi;
value ps = print_string
;
value pl s = do { ps s; print_newline () }
;
value display_rom_red s = html_red (Transduction.skt_to_html s)
;
value back_ground = background Chamois
;
```

```
value prelude lang = do
  { pl http_header
  ; page_begin heritage_dictionary_title
  ; pl (body_begin back_ground)
  ; print_title_solid Mauve (Some lang) stem_title_en
  }
;
value postlude lang =
  page_end lang True
;
value abor = abort default_language
;
value give_up phase =
  let mess = "Missing␣" ^ phase ^ "␣morphology" in do
  { abor Control.sys_err_mess mess; exit 0 }
;
value load_inflected phase =
  let file = match phase with
      [ "Noun" → Data.public_nouns2_file (* bigger than nouns *)
      | "Pron" → Data.public_pronouns_file
      | "Verb" → Data.public_roots_file
      | "Part" → Data.public_parts_file
      | "Inde" → Data.public_inde_file
      | "Absya" → Data.public_absya_file
      | "Abstvaa" → Data.public_abstvaa_file
      | "Iic" → Data.public_iics2_file (* bigger than iics *)
      | "Iiv" → Data.public_iivs_file
      | "Ifc" → Data.public_ifcs2_file (* bigger than ifcs *)
      | "Piic" → Data.public_piics_file
      | "Voca" → Data.public_vocas_file
      | _ → raise (Control.Fatal "Unexpected␣phase") (* Pv Auxi Eort *)
      ] in
  try (Gen.gobble file : Morphology.inflected_map)
  with [ _ → give_up phase ]
;
value generative = fun
  [ "Part" | "Piic" → True | _ → False ]
;
value answer_begin () = do
  { pl center_begin
```

```
  ; pl (table_begin_style (centered Yellow) [ noborder; ("cellspacing","20pt") ])
  ; ps tr_begin
  ; ps th_begin
  }
;
value answer_end () = do
  { ps th_end
  ; ps tr_end
  ; pl table_end
  ; pl center_end
  ; pl html_break
  }
;
value unvisarg_rev = fun (* we revert a final visarga to s *)
  [ [ 16 :: w ] → [ 48 :: w ]
  | w → w
  ]
;
value unvisarg word = Word.mirror (unvisarg_rev (Word.mirror word))
(* thus we may input raama.h and search for raamas in the morphological tables but we
can't input puna.h or anta.h and search for punar or antar or also verbal ninyu.h, stored as
ninyur even though it is displayed as ninyu.h *)
;
(* Main *)
value lemmatizer_engine () =
  let query = Sys.getenv "QUERY_STRING" in
  let env = create_env query in
  let translit = get "t" env Paths.default_transliteration
  and lex = get "lex" env Paths.default_lexicon
  and url_encoded_entry = get "q" env ""
  and url_encoded_cat = get "c" env "Noun" in
  let str = decode_url url_encoded_entry (* in translit *)
  and cat = decode_url url_encoded_cat
  and lang = language_of lex
  and encode = Encode.switch_code translit (* normalized input *) in do
  { prelude lang
  ; try let word = unvisarg (encode str) in
        let inflected_cat = load_inflected cat
        and gen = generative cat in
        let react inflected = do
```

```
          { ps (display_rom_red (Canon.decode word)) (* in romanized *)
          ; ps (span_begin Latin12)
          ; match Deco.assoc word inflected with
            [ [] → do
              { ps ("␣not␣found␣as␣a␣" ^ cat ^ "␣form")
              ; pl html_break
              }
            | le → do
              { ps "␣lemmatizes␣as:"
              ; pl html_break
              ; let pi = Morpho_html.print_inflected gen word in
                List.iter pi le
              }
            ]
          ; ps span_end
          } in do
        { answer_begin ()
        ; react inflected_cat
        ; answer_end ()
        ; postlude lang
        }
    with [ Stream.Error _ → abor "Illegal␣transliteration␣" str ]
  }
;
value safe_lemmatizer_engine () =
  try lemmatizer_engine ()
  with (* sanitized service *)
  [ Encode.In_error s → abor "Wrong␣input␣" s
  | Exit (* Sanskrit *) → abor "Wrong␣character␣in␣input␣-␣" "use␣ASCII"
  | Sys_error s → abor Control.sys_err_mess s (* file pb *)
  | Stream.Error s → abor Control.stream_err_mess s (* file pb *)
  | Invalid_argument s → abor Control.fatal_err_mess s (* sub *)
  | Failure s → abor Control.fatal_err_mess s (* anomaly *)
  | Control.Fatal s → abor Control.fatal_err_mess s (* anomaly *)
  | Not_found → abor Control.fatal_err_mess "assoc" (* assoc *)
  | End_of_file → abor Control.fatal_err_mess "EOF" (* EOF *)
  | _ → abor Control.fatal_err_mess "Unexpected␣anomaly"
  ]
;
safe_lemmatizer_engine ()
```

;

Interface for module Auto

The auto structure

module *Auto* : sig

type *rule* = (*Word.word* × *Word.word* × *Word.word*);
(∗ (w, u, v) such that (*rev* *u*) | *v* → *w* ∗)

type *auto* = [*State* of (*bool* × *deter* × *choices*)]
(∗ *bool* is *True* for accepting states ∗)
(∗ Possible refinement - order choices by right-hand sides of sandhi rules ∗)
and *deter* = *list* (*Word.letter* × *auto*)
and *choices* = *list* *rule*;

type *stack* = *list* *choices*; (∗ choice points stack ∗)

end;

Module Load_transducers

Load_transducers
Used for loading the transducers as well as root informations
Caution. This is an executable, that actually loads *roots_usage* at link time. It also defines a function *load_transducers* that loads *transducers_data* according to parameter *Lexer_control.full*, and build the relevant *transducter_vect* with *mk_transducers*. All this complication is due to the two modes Simple and Full. The actual transducter vector is stored in *Lexer_control.transducers_ref* by a call to *mk_transducers* in *Interface.graph_engine* or *Reader.reader_engine*.

open *Auto.Auto*; (∗ auto State ∗)
open *Automata_vector*; (∗ *transducers_datatype* ∗)

There are two different vector of transducers. The raw transducers constructed as make time are of type *transducers_datatype* described in *Automata_vector*. Then at cgi time the actual vector of transducers indexed by phases is of type *transducer_vect* below. Some of its slots are relevant to Full mode, some are relevant to Simple, some are relevant to both. This is really ugly, and the Simple mode may be deprecated in the future.

```
type transducer_vect =
  { nouv : auto (* vowel-initial nouns *)
  ; nouc : auto (* consonant-initial nouns *)
(*; noun : auto (* declined nouns and undeclinables *) *)
  ; noun2 : auto (* idem in mode non gen *)
  ; pron : auto (* declined pronouns *)
  ; root : auto (* conjugated root forms *)
(*; krid : auto (* kridantas forms *) *)
  ; lopa : auto (* e/o conjugated root forms with lopa *)
  ; lopak : auto (* e/o kridantas forms with lopa *)
  ; inde : auto (* indeclinables + infinitives *)
  ; abso : auto (* abso-ya *)
  ; absv : auto (* vowel-initial abso-tvaa *)
  ; absc : auto (* consonant-initial abso-tvaa *)
  ; peri : auto (* periphrastic perfect *)
  ; vokv : auto (* kridanta vocatives *)
  ; vokc : auto (* id *)
  ; inv : auto (* invocations *)
(*; iic : auto (* iic stems *) *)
  ; iic2 : auto (* iic stems in mode non gen *)
  ; iifc : auto (* iic forms of ifc stems *)
(*; iik : auto (* iik stems *) *)
  ; iiv : auto (* iiv periphrastic stems *)
  ; auxi : auto (* as k.r and bhuu finite forms *)
  ; auxiinv : auto (* as k.r and bhuu abs and inf forms *)
  ; auxik : auto (* their k.r and bhuu kridanta forms supports *)
  ; auxiick : auto (* their k.r and bhuu iic kridanta forms supports *)
(*; ifc : auto (* ifc forms *) *)
  ; ifcv : auto (* vowel-initial ifc *)
  ; ifcc : auto (* consonant-initial ifc *)
  ; ifc2 : auto (* ifc forms in mode non gen *)
  ; iiy : auto (* iic avyayiibhava *)
  ; avya : auto (* ifc avyayiibhava *)
  ; inftu : auto (* infinitives in -tu *)
  ; kama : auto (* forms of kaama *)
  ; prev : auto (* preverb sequences *)
  ; pvc : auto (* preverb sequences starting with consonant *)
  ; pvv : auto (* preverb sequences starting with vowel *)
  ; a : auto (* privative a *)
  ; an : auto (* privative an *)
```

```
  ; iicv : auto (* vowel-initial iic *)
  ; iicc : auto (* consonant-initial iic *)
  ; iivv : auto (* vowel-initial iiv *)
  ; iivc : auto (* consonant-initial iiv *)
  ; vocv : auto (* vowel-initial vocatives *)
  ; vocc : auto (* consonant-initial vocatives *)
  ; iikv : auto (* vowel-initial iik *)
  ; iikc : auto (* consonant-initial iik *)
  ; kriv : auto (* vowel-initial krids *)
  ; kric : auto (* consonant-initial krids *)
  ; cache : auto (* user-defined supplement to noun *)
  ; cachei : auto (* user-defined supplement to iic *)
  }
;
value empty_trans = State (False, [], []) (* dummy empty transducer *)
;
value dummy_transducer_vect = (* needed for initialisation of transducers_ref*)
  { noun2 = empty_trans
  ; root = empty_trans
  ; pron = empty_trans
  ; peri = empty_trans
  ; lopa = empty_trans
  ; lopak = empty_trans
  ; inde = empty_trans
  ; abso = empty_trans
  ; iic2 = empty_trans
  ; iifc = empty_trans
  ; ifc2 = empty_trans
  ; iiv = empty_trans
  ; auxi = empty_trans
  ; auxiinv = empty_trans
  ; auxik = empty_trans
  ; auxiick = empty_trans
  ; inv = empty_trans
  ; iiy = empty_trans
  ; avya = empty_trans
  ; inftu = empty_trans
  ; kama = empty_trans
  ; prev = empty_trans
  ; pvc = empty_trans
```

```
  ; pvv = empty_trans
  ; a = empty_trans
  ; an = empty_trans
  ; iicv = empty_trans
  ; iicc = empty_trans
  ; ifcv = empty_trans
  ; ifcc = empty_trans
  ; iivv = empty_trans
  ; iivc = empty_trans
  ; nouv = empty_trans
  ; nouc = empty_trans
  ; vocv = empty_trans
  ; vocc = empty_trans
  ; vokv = empty_trans
  ; vokc = empty_trans
  ; kriv = empty_trans
  ; kric = empty_trans
  ; iikv = empty_trans
  ; iikc = empty_trans
  ; absv = empty_trans
  ; absc = empty_trans
  ; cache = empty_trans
  ; cachei = empty_trans
  }
;
value abort_load cat =
  let mess = "Missing␣" ^ cat ^ "␣database" in
  raise (Control.Anomaly mess)
;
module Trans (* takes its prelude as parameter *)
  (Prel : sig value prelude : unit → unit; end)
 = struct

value transducers_data full =
  if full then Data.public_transducers_file (* Full mode *)
          else Data.public_transducers_file2 (* Simple mode *)
;
value load_transducers full =
   let file = transducers_data full in
   try (Gen.gobble file : transducers_datatype)
   with [ _ → do { Prel.prelude (); abort_load "Transducers"} ]
```

```
;
value load_cache () =
    let file = Data.public_trans_cache_file in
    try (Gen.gobble file : auto)
    with [ _ → empty_trans ] (* initialised to empty transducer *)
and load_cachei () =
    let file = Data.public_trans_cachei_file in
    try (Gen.gobble file : auto)
    with [ _ → empty_trans ] (* initialised to empty transducer *)
;
(* privative prefixes automata *)
value a_trans = State(False, [(1, State(True, [], [ cch ]))], [])
  where cch = (([ 22; 23 ], [], [ 23 ]) : rule) (* a-ch →acch *)
and an_trans = let n_trans = State(False, [(36, State(True, [], []))], []) in
                State(False, [(1, n_trans)], [])
;
(* Splitting an automaton into vowel-initial and consonant-initial solutions *)
(* with maximum sharing. Assumes deter is in increasing order of phonemes. *)
value split deter =
  let (rv, c) = split_rec [] deter
    where rec split_rec vow con = match con with
    [ [] → (vow, [])
    | [ ((c, _) as arc) :: rest ] →
        if c > 16 then (vow, con) else split_rec [ arc :: vow ] rest
    ] in
  (List.rev rv, c)
;
value split_auto = fun
  [ State (False, det, []) →
      let (vow, con) = split det in
      (State (False, vow, []), State (False, con, []))
    (* This assumes no non-determinism at the top node *)
  | State (False, det, rules) →
      let (vow, con) = split det in
      (State (False, vow, rules), State (False, con, []))
    (* This assumes non-determinism at the top node, and is needed for the preverb automaton. It assumes that the rules concern the vowel part. *)
  | _ → failwith "Split_auto"
  ]
;
```

```
value mk_transducers full = (* : transducter_vect *)
  let transducers_data = load_transducers full in
  if full then
  let transn = transducers_data.nouns
  and transi = transducers_data.iics
  and transf = transducers_data.ifcs
  and transk = transducers_data.parts
  and transik = transducers_data.piics
  and transv = transducers_data.vocas
  and vok = transducers_data.partvocs
  and iiv = transducers_data.iivs
  and abstvaa = transducers_data.abstvaa
  and pv = transducers_data.preverbs in
  (* now we split the subanta stems and forms starting with vowel or consonant *)
  let (transnv, transnc) = split_auto transn
  and (transiv, transic) = split_auto transi
  and (ifcv, ifcc) = split_auto transf
  and (kriv, kric) = split_auto transk
  and (iikv, iikc) = split_auto transik
  and (iivv, iivc) = split_auto iiv
  and (vocv, vocc) = split_auto transv
  and (vokv, vokc) = split_auto vok
  and (absv, absc) = split_auto abstvaa
  and (pvkv, pvkc) = split_auto pv in
  { noun2 = empty_trans
  ; root = transducers_data.roots
  ; pron = transducers_data.pronouns
  ; peri = transducers_data.peris
  ; lopa = transducers_data.lopas
  ; lopak = transducers_data.lopaks
  ; inde = transducers_data.indecls
  ; abso = transducers_data.absya
  ; iic2 = empty_trans
  ; iifc = transducers_data.iifcs
  ; ifc2 = empty_trans
  ; iiv = iiv
  ; auxi = transducers_data.auxis
  ; auxiinv = transducers_data.auxiinvs
  ; auxik = transducers_data.auxiks
  ; auxiick = transducers_data.auxiicks
```

```
; inv = transducers_data.invs
; iiy = transducers_data.avyayais
; avya = transducers_data.avyayafs
; inftu = transducers_data.inftu
; kama = transducers_data.kama
; prev = pv
; pvc = pvkc
; pvv = pvkv
; a = a_trans
; an = an_trans
; iicv = transiv
; iicc = transic
; ifcv = ifcv
; ifcc = ifcc
; iivv = iivv
; iivc = iivc
; nouv = transnv
; nouc = transnc
; vocv = vocv
; vocc = vocc
; vokv = vokv
; vokc = vokc
; kriv = kriv
; kric = kric
; iikv = iikv
; iikc = iikc
; absv = absv
; absc = absc
; cache = load_cache ()
; cachei = load_cachei ()
}
else (* Simple mode *)
{ noun2 = transducers_data.nouns2
; root = transducers_data.roots
; pron = transducers_data.pronouns
; peri = empty_trans
; lopa = transducers_data.lopas
; lopak = empty_trans
; inde = transducers_data.indecls
; abso = transducers_data.absya
```

```
  ; iic2 = transducers_data.iics2
  ; iifc = transducers_data.iifcs
  ; ifc2 = transducers_data.ifcs2
  ; iiv = empty_trans
  ; auxi = transducers_data.auxis
  ; auxiinv = transducers_data.auxiinvs
  ; auxik = empty_trans
  ; auxiick = empty_trans
  ; inv = transducers_data.invs
  ; iiy = empty_trans
  ; avya = empty_trans
  ; inftu = empty_trans
  ; kama = empty_trans
  ; prev = transducers_data.preverbs
  ; pvc = empty_trans
  ; pvv = empty_trans
  ; a = empty_trans
  ; an = empty_trans
  ; iicv = empty_trans
  ; iicc = empty_trans
  ; ifcv = empty_trans
  ; ifcc = empty_trans
  ; iivv = empty_trans
  ; iivc = empty_trans
  ; nouv = empty_trans
  ; nouc = empty_trans
  ; vocv = empty_trans
  ; vocc = empty_trans
  ; vokv = empty_trans
  ; vokc = empty_trans
  ; kriv = empty_trans
  ; kric = empty_trans
  ; iikv = empty_trans
  ; iikc = empty_trans
  ; absv = empty_trans
  ; absc = empty_trans
  ; cache = load_cache ()
  ; cachei = load_cachei ()
  }
;
```

Lexicalized root informations needed for Dispatcher

```
value roots_usage = (* attested preverb sequences *)
  try (Gen.gobble Data.public_roots_usage_file : Deco.deco string)
  with [ _ → do { Prel.prelude (); abort_load "RU" } ]
;
end (* Trans *)
;
```

Interface for module Dispatcher

Dispatcher: Sanskrit Engine in 55 phases automaton (plus 2 fake ones)
The Dispatch functor maps a transducer vector of 39 aums into
- a dispatch automaton implementing a regular description over
45 phases of lexical analysis
- an initial vector of initial resumptions
- a final test for lexical acceptance
- a consistency check of the output of the segmenting transducer
Dispatch, instantiated by Transducers, is used as parameter of the Segment functor from Segmenter or Interface.

```
open Auto.Auto;
open Load_transducers; (* transducer_vect *)
open Morphology; (* inflexion_tag Verb_form pada_tag morphology *)
open Phases.Phases; (* phase etc. *)

module Dispatch : functor
  (Trans : sig value roots_usage : Deco.deco string; end) → functor
  (Lem : sig value morpho : morphology; end) → functor
  (Segment_control : sig value transducers_ref : ref transducer_vect; end) → sig

value transducer : phase → auto
;
value initial : bool → phases
;
value dispatch : bool → Word.word → phase → phases
;
value accepting : phase → bool
;
type input = Word.word (* input sentence represented as a word *)
and transition = (* Reflexive relation *)
    [ Euphony of rule (* (w, rev u, v) such that u | v → w *)
```

```
    | Id (* identity or no sandhi *)
    ]
and segment = (phase × Word.word × transition)
and output = list segment;

value valid_morpho :
  bool → string → Word.word → Morphology.inflexion_tag → bool
;
value trim_tags :
  bool → Word.word → string → Morphology.multitag → Morphology.multitag
;
value validate : output → output (* consistency check and glueing *)
;
value sanitize_sa : bool → output → output
;
end;
```

Module Dispatcher

Dispatcher: Sanskrit Engine in 53 phases automaton (plus 2 fake ones)
The Dispatch functor maps a transducer vector of 39 aums into
- a dispatch automaton implementing a regular description over
45 phases of lexical analysis
- an initial vector of initial resumptions
- a final test for lexical acceptance
- a consistency check of the output of the segmenting transducer

Dispatch, instantiated by Transducers, is used as parameter of the Segment functor from Segmenter or Interface. It defines the phase automaton transitions. There are two versions: 1 for Complete, 2 for Simple.

```
open Auto.Auto;
open Load_transducers; (* transducer_vect Trans mk_transducers roots_usage *)
open Skt_morph;
open Phonetics; (* phantomatic amuitic *)
open Morphology; (* inflected inflected_map Verb_form morphology *)
open Naming; (* homo_undo look_up_homo unique_kridantas *)
open Phases.Phases; (* phase etc. *)

module Dispatch
  (* To be instantiated by Transducers from Lexer or Interface *)
  (Trans : sig value roots_usage : Deco.deco string; end)
```

```
  (Lem : sig value morpho : morphology; end)
  (Segment_control : sig value transducers_ref : ref transducer_vect; end)
  = struct
open Trans; (* transducers_ref *)
open Lem;
```

transducer : *phase* → *auto*

```
value transducer phase =
  let transducers = Segment_control.transducers_ref.val in match phase with
  [ Nouv → transducers.nouv (* vowel-initial noun *)
  | Nouc → transducers.nouc (* consonant-initial noun *)
  | Noun2 → transducers.noun2 (* idem in mode non gen *)
  | Pron → transducers.pron (* declined pronouns *)
  | Root → transducers.root (* conjugated root forms *)
  | Vokv → transducers.vokv (* vowel-initial vocative k.rdaantas *)
  | Vokc → transducers.vokc (* consonant-initial vocative k.rdaantas *)
  | Inde → transducers.inde (* indeclinables, particles *)
  | Absv → transducers.absv (* vowel-initial absolutives in -tvaa *)
  | Absc → transducers.absc (* consonant-initial absolutives in -tvaa *)
  | Abso → transducers.abso (* absolutives in -ya *)
  | Iic2 → transducers.iic2 (* idem in mode non gen *)
  | Iiif → transducers.iifc (* fake iic of ifc stems *)
  | Iiv → transducers.iiv (* in initio verbi nominal stems, perpft *)
  | Inv → transducers.inv (* invocations *)
  | Auxi → transducers.auxi (* k.r as and bhuu finite forms *)
  | Auxiinv → transducers.auxiinv (* k.r as and bhuu abs and inf forms *)
  | Auxik → transducers.auxik (* k.r as and bhuu kridanta forms *)
  | Auxiick → transducers.auxiick (* k.r as and bhuu kridanta bare forms *)
  | Peri → transducers.peri (* periphrastic perfect *)
  | Lopa → transducers.lopa (* e/o root forms *)
  | Lopak → transducers.lopak (* e/o kridanta forms *)
  | Ifcv → transducers.ifcv (* vowel-initial ifc forms *)
  | Ifcc → transducers.ifcc (* consonant-initial ifc forms *)
  | Ifc2 → transducers.ifc2 (* idem in mode non gen *)
  | Pv → transducers.prev (* preverbs *)
  | Pvkc | Pvc → transducers.pvc (* preverbs starting with consonant *)
  | Pvkv | Pvv → transducers.pvv (* preverbs starting with vowel *)
  | A | Ai → transducers.a (* privative a *)
  | An | Ani → transducers.an (* privative an *)
  | Iicv → transducers.iicv (* vowel-initial iic *)
```

```
  | Iicc → transducers.iicc (* consonant-initial iic *)
  | Iikv → transducers.iikv (* vowel-initial iic k.rdaanta *)
  | Iikc → transducers.iikc (* consonant-initial iic k.rdaanta *)
  | Iivv → transducers.iivv (* vowel-initial iiv (cvi) *)
  | Iivc → transducers.iivc (* consonant-initial iiv (cvi) *)
  | Kriv → transducers.kriv (* vowel-initial krid *)
  | Kric → transducers.kric (* consonant-initial krid *)
  | Vocv → transducers.vocv (* vowel-initial vocatives *)
  | Vocc → transducers.vocc (* consonant-initial vocatives *)
  | Iiy → transducers.iiy (* iic avyayiibhava *)
  | Avy → transducers.avya (* ifc avyayiibhava *)
  | Inftu → transducers.inftu (* infinitives in -tu iic. Renou HLS 72 *)
  | Kama → transducers.kama (* ifcs of kaama/manas: tyaktukaama dra.s.tumanas *)
  | Cache → transducers.cache (* cached forms *)
  | Cachei → transducers.cachei (* cached iic forms *)
  | Noun | Iic | Iik | Ifc | Voca | Krid | Vok
    → raise (Control.Anomaly "composite phase")
  | Unknown → raise (Control.Anomaly "transducer - Unknown")
  | _ → raise (Control.Anomaly "no transducer for Comp fake phase")
  ]
;
(* Tests whether a word starts with a phantom phoneme (precooked aa-prefixed finite or
participial or infinitive or abs-ya root form) *)
value phantomatic = fun
  [ [ c :: _ ] → c < (−2) ∨ c = 123
  | _ → False
  ]
(* Amuitic forms start with -2 = - which elides preceding -a or -aa from Pv *)
and amuitic = fun
  [ [ −2 :: _ ] → True
  | _ → False
  ]
;
```

(∗ Simplified description, not with all phases ∗)
(∗ We recognize $S = (Subst + Pron + Verb + Inde + Voca)^+$
with $Verb = (1 + Pv).Root + Pv.Abso + Iiv.Auxi$,
$Subst = Noun + Iic.Ifc + Iic.Subst + Iiv.Auxik$,
$Noun = Nounv + Nounc$ and $Iic = Iicv + Iicc$
NB. $Abso$ = absolutives in -ya, $Inde$ contains absolutives in -tvaa $Voca = Vocv + Vocc$ (vocatives), $Auxi$ = finite forms of bhuu and k.r.

The following is obtained from the above recursion equation by Brzozowski's derivatives like in Berry-Sethi's translator. ∗)

```
value cached = (* potentially cached lexicon acquisitions *)
  if Web.cache_active.val ="t" then [ Cache; Cachei ] else []
;
(* initial1, initial2: phases *)
value initial1 =
    (* All phases but Ifc, Abso, Auxi, Auxiinv, Auxik, Auxiick, Lopa, Lopak. *)
    [ Inde; Iicv; Iicc; Nouv; Nouc; Pron; A; An; Root; Kriv; Kric; Iikv; Iikc
    ; Peri; Pv; Pvkv; Pvkc; Iiv; Iivv; Iivc; Iiy; Inv; Ai; Ani
    ; Absv; Absc; Inftu; Vocv; Vocc; Vokv; Vokc ] @ cached
and initial2 = (* simplified segmenter with less phases, no generation *)
    [ Inde; Iic2; Noun2; Pron; Root; Pv; Iiv; Absv; Absc ]
;
value initial full = if full then initial1 else initial2
;
(* dispatch1: Word.word -¿ phase -¿ phases *)
value dispatch1 w = fun (* w is the current input word *)
  [ Nouv | Nouc | Pron | Inde | Abso | Auxi | Auxiinv | Auxik | Kama | Ifcv | Ifcc
  | Kriv | Kric | Absv | Absc | Avy | Lopak | Root | Lopa | Cache → initial1
  | A → if phantomatic w then [] else
          [ Iicc; Nouc; Iikc; Kric; Pvkc; Iivc; Vocc; Vokc ]
  | An → if phantomatic w then [] else
           [ Iicv; Nouv; Iikv; Kriv; Pvkv; Iivv; Vocv; Vokv
           ; A (* eg anak.sara anavadya *) ; An (* attested ? *) ]
  | Ai → [ Absc; Pvc ]
  | Ani → [ Absv; Pvv ]
    (* This assumes that privative prefixes cannot prefix Ifc forms justified by P{2,2,6} a-x
only if x is a subanta. *)
  | Iicv | Iicc | Iikv | Iikc | Iiif | Auxiick | Cachei → (* Compounding *)
        [ Iicv; Iicc; Nouv; Nouc; A; An; Ifcv; Ifcc; Iikv; Iikc; Kriv; Kric
        ; Pvkv; Pvkc; Iiif; Iivv; Iivc; Vocv; Vocc; Vokv; Vokc ] @ cached
  | Pv → if phantomatic w then [] else
           if amuitic w then [ Lopa ] else [ Root; Abso; Peri; Inftu ]
  | Pvc | Pvv → if phantomatic w then [] else [ Abso ]
  | Pvkc | Pvkv → if phantomatic w then [] else
           if amuitic w then [ Lopak ] else [ Iikv; Iikc; Kriv; Kric; Vokv; Vokc ]
  | Iiv → [ Auxi; Auxiinv ] (* as bhuu as and k.r finite, abs and inf forms *)
  | Iivv | Iivc → [ Auxik; Auxiick ] (* bhuu and k.r kridanta forms *)
  | Iiy → [ Avy ]
```

```
  | Peri → [ Auxi ] (* overgenerates, should be only perfect forms *)
  | Inftu → [ Kama ]
  | Vocc | Vocv | Vokv | Vokc → [ ]
      (* only chunk-final vocatives so no Iic overlap *)
  | Inv → [ Vocv; Vocc; Vokv; Vokc ] (* invocations before vocatives *)
(* Privative prefixes A and An are not allowed to prefix Ifc like a-dhii *)
  | Noun | Iic | Iik | Voca | Krid | Noun2 | Iic2 | Ifc2 | Vok
  | Unknown → failwith "Dispatcher␣anomaly"
  | ph → failwith ("Dispatcher␣fake␣phase:␣" ^ string_of_phase ph)
  ]
and dispatch2 w = fun (* simplified segmenter *)
  [ Noun2 | Pron | Inde | Abso | Absv | Absc | Auxi | Auxiinv | Ifc2 →
      if phantomatic w then [ Root; Abso ] else initial2
  | Root | Lopa →
      if phantomatic w then [ ] (* no consecutive verbs in chunk *)
      else [ Inde; Iic2; Noun2; Pron ]
  | Iic2 → [ Iic2; Noun2; Ifc2 ]
  | Pv → if phantomatic w then [ ] else
           if amuitic w then [ Lopa ] else [ Root; Abso ]
  | Iiv → [ Auxi; Auxiinv ]
  | _ → failwith "Dispatcher␣anomaly"
  ]
;
(* dispatch: bool -¿ Word.word -¿ phase -¿ phases *)
value dispatch full = if full then dispatch1 else dispatch2
;
value terminal = (* Accepting phases *)
   [ Nouv; Nouc; Noun2
   ; Pron
   ; Root
   ; Kriv
   ; Kric
   ; Inde
   ; Abso; Absv; Absc
   ; Ifcv; Ifcc; Ifc2
   ; Auxi; Auxiinv; Auxik
   ; Vocc; Vocv; Vokv; Vokc; Inv
   ; Lopa; Lopak
   ; Avy; Kama
   ; Cache
```

```
    ]
;
```

accepting: phase -¿ bool

```
value accepting phase = List.mem phase terminal
;
(* Segmenter control *)
type input = Word.word (* input sentence represented as a word *)
and transition = (* Reflexive relation *)
     [ Euphony of rule (* (w, rev u, v) such that u | v → w *)
     | Id (* identity or no sandhi *)
     ]
and segment = (phase × Word.word × transition)
and output = list segment
;
(* Now consistency check - we check that preverbs usage is consistent with root px declaration
in lexicon *)
```

```
value assoc_word word deco =
    let infos = Deco.assoc word deco in
    if infos = [] then failwith ("Unknown␣form:␣" ^ Canon.decode word)
    else infos
;
value autonomous root = (* root form allowed without preverb *)
  let infos = assoc_word root roots_usage in
  match infos with
     [ [ "" :: _ ] → True
     | _ → False
     ]
and attested prev root = (* prev is attested preverb sequence for root *)
  let pvs = assoc_word root roots_usage in
  List.mem prev pvs (* NB attested here means lexicalized entry *)
;
value gana_o = fun
  [ None → 0 (* arbitrary - beware this marks non-present forms in Pada *)
  | Some g → g (* only used for "tap" *)
  ]
and voice_o v = fun
  [ None → True
  | Some voice → voice = v
  ]
```

```
;
(* pvs is a list of preverb words *)
(* upasarga closest to the root form *)
value main_preverb pvs = List2.last pvs
;
value main_preverb_string pv =
  Canon.decode (main_preverb (assoc_word pv Naming.preverbs_structure))
;
value attested_verb (o_gana, o_voice) pv root = attested pv root ∧
  let gana = gana_o o_gana in
  let upasarga = main_preverb_string (Encode.code_string pv) in
  try let pada = Pada.voices_of_pv upasarga gana (Canon.decode root) in
      match pada with
      [ Pada.Ubha → True
      | _ → voice_o pada o_voice
      ]
  with [ Pada.Unattested → False ]
;
(* Similarly for root forms used without preverb *)
value autonomous_root (o_gana, o_voice) root = autonomous root ∧
  let gana = gana_o o_gana in
  try let pada = Pada.voices_of_pv "" gana (Canon.decode root) in
      match pada with
      [ Pada.Ubha → True
      | _ → voice_o pada o_voice
      ]
  with [ Pada.Unattested → False ]
;
value pada_of_voice = fun
  [ Active → Some Pada.Para
  | Middle → Some Pada.Atma
  | _ → None
  ]
;
exception Unvoiced
;
value extract_gana_pada = fun
  [ Verb_form (conj, paradigm) _ _ →
       let (o_gana, voice) = match paradigm with
            [ Presenta g _ → (Some g, Active)
```

```
              | Presentm g _ → (Some g, Middle)
              | Presentp _ → (None, Passive)
              | Conjug _ v | Perfut v → (None, v)
              ] in
          (conj, (o_gana, pada_of_voice voice))
  | Ind_verb _ _ → raise Unvoiced (* could be refined *)
  | _ → failwith "Unexpected root form"
  ]
and extract_gana_pada_k krit =
    let (o_gana, voice) = match krit with
        [ Ppp | Pprp | Pfutp _ → (None, Passive)
        | Pppa | Ppfta | Pfuta → (None, Active)
        | Ppftm | Pfutm → (None, Middle)
        | Ppra g → (Some g, Active)
        | Pprm g → (Some g, Middle)
        | _ → raise Unvoiced (* could be refined *)
        ] in
    (o_gana, pada_of_voice voice)
;
value fail_inconsistency form =
  raise (Control.Anomaly ("Unknown root form: " ^ Canon.decode form))
;
value valid_morph_pv pv root (morph : Morphology.inflexion_tag) = try
  let (conj, gana_pada) = extract_gana_pada morph in
  if conj = Primary then attested_verb gana_pada pv root else attested pv root
  with [ Unvoiced → attested pv root ]
and valid_morph_aut root (morph : Morphology.inflexion_tag) = try
  let (conj, gana_pada) = extract_gana_pada morph in
  if conj = Primary then autonomous_root gana_pada root
                    else autonomous root (* eg. kalpaya Para ca. while k.lp Atma *)
  with [ Unvoiced → autonomous root ]
;
value valid_morph_pv_k pv krit_stem morph = (* morph of form Part_form *)
  let (homo, bare_stem) = homo_undo krit_stem in
  let krit_infos = assoc_word bare_stem unique_kridantas in
  let ((conj, krit), root) = look_up_homo homo krit_infos in try
  (* Asymmetry of treatment: conj is deduced from krit_stem, not from morph *)
  let gana_pada = extract_gana_pada_k krit in
  if conj = Primary then attested_verb gana_pada pv root else attested pv root
  with [ Unvoiced → attested pv root ]
```

```
;
value validate_pv pv root_form =
  match Deco.assoc root_form morpho.roots with
    [ [] → fail_inconsistency root_form
    | tags → List.exists valid tags
                (* NB later on the lexer will refine in filtering validity *)
                where valid (delta, morphs) =
                  let root = Word.patch delta root_form in
                  List.exists (valid_morph_pv pv root) morphs
    ]
;
value validate_pv_tu pv root_form = (* special case infinitive forms in -tu *)
  match Deco.assoc root_form morpho.inftu with
    [ [] → fail_inconsistency root_form
    | tags → List.exists valid tags
                (* NB later on the lexer will refine in filtering validity *)
                where valid (delta, morphs) =
                  let root = Word.patch delta root_form in
                  List.exists (valid_morph_pv pv root) morphs
    ]
;
value validate_pv_k pv krit_form (delta, _) = (* see Morpho.print_inv_morpho *)
  let krit_stem = Word.patch delta krit_form in
  let (homo, bare_stem) = homo_undo krit_stem in
  let krit_infos = assoc_word bare_stem unique_kridantas in
  let ((conj, krit), root) = look_up_homo homo krit_infos in
  try let gana_pada = extract_gana_pada_k krit in
      if conj = Primary then attested_verb gana_pada pv root else attested pv root
  with [ Unvoiced → attested pv root ]
;
value autonomous_form root_form =
  match Deco.assoc root_form morpho.roots with
    [ [] → fail_inconsistency root_form
    | tags → List.exists valid tags (* Lexer will filter later on *)
      where valid (delta, morphs) =
        let root = Word.patch delta root_form in
        List.exists (valid_morph_aut root) morphs
    ]
;
(* This allows to rule out ifc only kridantas even when root autonomous *)
```

```
value filter_out_krit krit root = match Canon.decode root with
  [ "i" | "dagh" → krit = Ppp (* -ita -daghna *)
  | _ → False
  ]
;
value autonomous_form_k krid_form (delta, _) =
  let stem = Word.patch delta krid_form in
  let (homo, bare_stem) = homo_undo stem in
  let krid_infos = assoc_word bare_stem unique_kridantas in
  let ((conj, krit), root) = look_up_homo homo krid_infos in
  try let gana_pada = extract_gana_pada_k krit in
      if conj = Primary then if filter_out_krit krit root then False
                             else autonomous_root gana_pada root
      else True
  with [ Unvoiced → autonomous root ]
;
(* Checks whether a verbal or participial form is attested/validated *)
value valid_morpho gen =
  if gen then valid_morph_pv_k else valid_morph_pv
;
(* This inspects a multitag in order to filter out pv-inconsistent taggings. *)
(* It is used by Interface and Lexer for Reader and Parser *)
value trim_tags gen form pv tags = List.fold_right trim tags []
      where trim (delta, morphs) acc = (* tags : Morphology.multitag *)
        let stem = Word.patch delta form in (* root or kridanta *)
        let valid_pv = valid_morpho gen pv stem in
        let ok_morphs = List.filter valid_pv morphs in
        if ok_morphs = [] then acc else [ (delta, ok_morphs) :: acc ]
;
(* prune_sa checks that sa.h does not occur before consonants (should be sa) *)
(* NB called with last = (_, mirror form, _) and out = last :: next *)
value prune_sa out form last = fun (* next *)
  [ [ (Pron, [ 48; 1; 48 ], _) :: rest ] (* sas *) → match form with
      [ [ c :: _ ] when consonant c → []
      | _ → out
      ]
    (* Similar conditions for pronoun e.sa *)
  | [ (Pron, [ 48; 1; 47; 10 ], _) :: rest ] (* esas *) → match form with
      [ [ c :: _ ] when consonant c → []
      | _ → out
```

```
      ]
  | _ → out
  ]
;
(* Similar to List2.subtract but raises Anomaly exception *)
value rec chop word = fun
  [ [] → word
  | [ c :: r ] → match word with
     [ [ c' :: r' ] when c' = c → chop r' r
     | _ → raise (Control.Anomaly "Wrong transition between segments")
     ]
  ]
;
value iic_phase = fun
  [ Iicv | Iicc | Iikv | Iikc
  | Comp (_, Iikv) _ _ | Comp (_, Iikc) _ _ → True
  | _ → False ]
;
value apply_sandhi rleft right = fun
    [ Euphony (w, ru, v) →
        let rl = chop rleft ru
        and r = chop right v in List2.unstack rl (w @ r)
    | Id → List2.unstack rleft right
    ]
;
```

validate : *bool* → *output* → *output* dynamic consistency check in Segmenter. It refines the regular language of dispatch by contextual conditions expressing that preverbs are consistent with the following verbal form. The forms are then compounded, otherwise rejected. Things would be much simpler if we generated forms of verbs and kridantas with (only valid) preverbs attached, since this check would be unnecessary. On the other hand, we would have to solve the ihehi problem.

```
value validate out = match out with
  [ [] → []
  (* Preventing overgeneration of forms "sa" and "e.sa" P{6,1,132} Kale§50 *)
  | [ _ :: [ (Pron, [ 1; 48 ], s_) :: _ ] ] (* sa must be chunk-terminal *)
  | [ _ :: [ (Pron, [ 1; 47; 10 ], s_) :: _ ] ] → [] (* same for e.sa *)
  | [ (Root, rev_root_form, s) :: [ (Pv, prev, sv) :: r ] ] →
      let pv = Word.mirror prev in
      let pv_str = Canon.decode pv
```

```
      and root_form = Word.mirror rev_root_form in
      if validate_pv pv_str root_form then
         let form = apply_sandhi prev root_form sv in
         let verb_form = Word.mirror form in
         (* We glue the two segments with a composite tag keeping information *)
         [ (Comp (Pv, Root) pv root_form, verb_form, s) :: r ]
      else [ ]
  | [ ((Root, rev_root_form, s) as last) :: next ] →
      let root_form = Word.mirror rev_root_form in
      if autonomous_form root_form then prune_sa out root_form last next else [ ]
  | [ (Lopa, rev_lopa_form, s) :: [ (Pv, prev, sv) :: r ] ] →
      let pv = Word.mirror prev in
      let pv_str = Canon.decode pv
      and lopa_form = Word.mirror rev_lopa_form in
      let root_form = match lopa_form with
                      [ [ −2 :: rf ] → rf
                      | _ → failwith "Wrong lopa form"
                      ] in
      if validate_pv pv_str root_form then
        let form = apply_sandhi prev lopa_form sv in
        let verb_form = Word.mirror form in
        [ (Comp (Pv, Lopa) pv lopa_form, verb_form, s) :: r ]
      else [ ]
  | [ ((Lopa, rev_lopa_form, _) as last) :: next ] →
      let lopa_form = Word.mirror rev_lopa_form in
      let verb_form = match lopa_form with
                      [ [ −2 :: rf ] → rf | _ → failwith "Wrong lopa form" ] in
      if autonomous_form verb_form then prune_sa out verb_form last next else [ ]
  | (* infinitives in -tu with preverbs *)
    [ (Inftu, rev_root_form, s) :: [ (Pv, prev, sv) :: r ] ] →
      let pv = Word.mirror prev in
      let pv_str = Canon.decode pv
      and root_form = Word.mirror rev_root_form in
      if validate_pv_tu pv_str root_form then
         let form = apply_sandhi prev root_form sv in
         let verb_form = Word.mirror form in
         (* We glue the two segments with a composite tag keeping information *)
         [ (Comp (Pv, Inftu) pv root_form, verb_form, s) :: r ]
      else [ ]
   | (* kridanta forms with preverbs *)
```

```
    [ (phk, rev_krid_form, s) :: [ (ph, prev, sv) :: r ] ]
        when krid_phase phk ∧ preverb_phase ph →
     let pv = Word.mirror prev in
     let pv_str = Canon.decode pv
     and krid_form = Word.mirror rev_krid_form in
     match Deco.assoc krid_form morpho.krids with
     [ [] → failwith ("Unknown␣krid_form:␣" ^ Canon.decode krid_form)
     | tags → if List.exists (validate_pv_k pv_str krid_form) tags then
                   let form = apply_sandhi prev krid_form sv in
                   let cpd_form = Word.mirror form in
                   [ (Comp (ph, phk) pv krid_form, cpd_form, s) :: r ]
               (* else *)
     else []
     ]
  | [ (Kriv, rev_krid_form, s) :: next ] →
    let krid_form = Word.mirror rev_krid_form in
     match Deco.assoc krid_form morpho.krids with
     [ [] → failwith ("Unknown␣krid_form:␣" ^ Canon.decode krid_form)
     | tags → if List.exists (autonomous_form_k krid_form) tags
              then out else []
     ]
  | [ ((Kric, rev_krid_form, _) as last) :: next ] →
     let krid_form = Word.mirror rev_krid_form in
     match Deco.assoc krid_form morpho.krids with
     [ [] → failwith ("Unknown␣krid_form:␣" ^ (Canon.decode krid_form))
     | tags → if List.exists (autonomous_form_k krid_form) tags
              then prune_sa out krid_form last next else []
     ]
  | (* iic kridanta forms with preverbs *)
    [ (phk, rev_ikrid_form, s) :: [ (ph, prev, sv) :: r ] ]
        when ikrid_phase phk ∧ preverb_phase ph →
     let pv = Word.mirror prev in
     let pv_str = Canon.decode pv
     and ikrid_form = Word.mirror rev_ikrid_form in
     match Deco.assoc ikrid_form morpho.iiks with
     [ [] → failwith ("Unknown␣ikrid_form:␣" ^ Canon.decode ikrid_form)
     | tags → if List.exists (validate_pv_k pv_str ikrid_form) tags then
                   let form = apply_sandhi prev ikrid_form sv in (* Z *)
                   let cpd_form = Word.mirror form in
                   [ (Comp (ph, phk) pv ikrid_form, cpd_form, s) :: r ]
```

```
              else [ ]
    ]
| [ (Iikv, rev_krid_form, _) :: next ] →
    let krid_form = Word.mirror rev_krid_form in
    match Deco.assoc krid_form morpho.iiks with
    [ [ ] → failwith ("Unknown krid_form: " ^ Canon.decode krid_form)
    | tags → if List.exists (autonomous_form_k krid_form) tags
              then out else [ ]
    ]
| [ ((Iikc, rev_krid_form, _) as last) :: next ] →
    let krid_form = Word.mirror rev_krid_form in
    match Deco.assoc krid_form morpho.iiks with
    [ [ ] → failwith ("Unknown krid_form: " ^ Canon.decode krid_form)
    | tags → if List.exists (autonomous_form_k krid_form) tags
      then prune_sa out krid_form last next else [ ]
    ]
| (* vocative kridanta forms with preverbs *)
  [ (phk, rev_krid_form, s) :: [ (ph, prev, sv) :: r ] ]
       when vkrid_phase phk ∧ preverb_phase ph →
    let pv = Word.mirror prev in
    let pv_str = Canon.decode pv
    and krid_form = Word.mirror rev_krid_form in
    match Deco.assoc krid_form morpho.voks with
    [ [ ] → failwith ("Unknown krid_form: " ^ Canon.decode krid_form)
    | tags → if List.exists (validate_pv_k pv_str krid_form) tags then
                  let form = apply_sandhi prev krid_form sv in
                  let cpd_form = Word.mirror form in
                  [ (Comp (ph, phk) pv krid_form, cpd_form, s) :: r ]
              else [ ]
    ]
| [ (Vokv, rev_krid_form, _) :: next ] →
    let krid_form = Word.mirror rev_krid_form in
    match Deco.assoc krid_form morpho.voks with
    [ [ ] → failwith ("Unknown krid_form: " ^ Canon.decode krid_form)
    | tags → if List.exists (autonomous_form_k krid_form) tags
              then out else [ ]
    ]
| [ ((Vokc, rev_krid_form, _) as last) :: next ] →
    let krid_form = Word.mirror rev_krid_form in
    match Deco.assoc krid_form morpho.voks with
```

```
      [ [] → failwith ("Unknown␣krid_form:␣" ^ Canon.decode krid_form)
      | tags → if List.exists (autonomous_form_k krid_form) tags
                then prune_sa out krid_form last next else []
      ]
  | [ (Lopak, rev_lopak_form, s) :: [ (ph, prev, sv) :: r ] ]
        when preverb_phase ph →
      let pv = Word.mirror prev in
      let pv_str = Canon.decode pv
      and lopak_form = Word.mirror rev_lopak_form in
      let krid_form = match lopak_form with
                        [ [ -2 :: rf ] → rf | _ → failwith "Wrong␣lopa␣form" ] in
      match Deco.assoc krid_form morpho.lopaks with
      [ [] → failwith ("Unknown␣krid_form:␣" ^ Canon.decode krid_form)
      | tags → if List.exists (validate_pv_k pv_str krid_form) tags then
                  let form = apply_sandhi prev krid_form sv in
                  let cpd_form = Word.mirror form in
                  [ (Comp (ph, Lopak) pv krid_form, cpd_form, s) :: r ]
                else []
      ] | [ (Peri, rev_peri_form, s) :: [ (Pv, prev, sv) :: r ] ] →
      let pv = Word.mirror prev in
      let pv_str = Canon.decode pv
      and peri_form = Word.mirror rev_peri_form in
      match Deco.assoc peri_form morpho.peris with
      [ [] → failwith ("Unknown␣peri_form:␣" ^ Canon.decode peri_form)
      | tags → let valid (delta, morphs) =
                  let root = Word.patch delta peri_form in
                  attested pv_str root in
                if List.exists valid tags then
                  let form = apply_sandhi prev peri_form sv in
                  let cpd_form = Word.mirror form in
                  [ (Comp (Pv, Peri) pv peri_form, cpd_form, s) :: r ]
                else []
      ]
  | [ (Abso, rev_abso_form, s) :: [ (ph, prev, sv) :: r ] ]
          when preverb_phase ph →
      (* Takes care of absolutives in -ya *)
      let pv = Word.mirror prev in
      let pv_str = Canon.decode pv
      and abso_form = Word.mirror rev_abso_form in
      match Deco.assoc abso_form morpho.absya with
```

```
      [ [] → failwith ("Unknown␣abs_form:␣" ^ Canon.decode abso_form)
      | tags → let valid (delta, morphs) =
                   let root = Word.patch delta abso_form in
                   attested pv_str root in
               if List.exists valid tags then
                   let form = apply_sandhi prev abso_form sv in
                   let cpd_form = Word.mirror form in
                   [ (Comp (Pv, Abso) pv abso_form, cpd_form, s) :: r ]
               else [ ]
      ]
  | [ (Abso, rev_abso_form, s) :: next ] → (* impossible since Abso follows Pv *)
      raise (Control.Anomaly "Isolated␣Abso␣form") (* phase enforced *)
  | [ (_, w, _) :: _ ] when phantomatic (Word.mirror w) → (* should not happen *)
     let mess = "Bug␣phantomatic␣segment:␣" ^ Canon.rdecode w in
     raise (Control.Anomaly mess)
  | [ (phase, _, _) :: [ (pv, _, _) :: _ ] ] when preverb_phase pv →
      let m = "validate:␣" ^ string_of_phase pv ^ "␣" ^ string_of_phase phase in
      raise (Control.Anomaly m) (* all preverbs ought to have been processed *)
  | [ ((_, rform, _) as last) :: next ] → let form = Word.mirror rform in
                                           prune_sa out form last next
  ]
;
(* Inter-chunk sa/e.sa check : sa_check is True iff sa/e.sa segment is last in chunk and chunk
not last *)
value sanitize_sa sa_check chunk = match chunk with
  [ [ (Pron, [ 1; 47; 10 ], _) :: _ ] (* ... e.sa *)
  | [ (Pron, [ 1; 48 ], _) :: _ ] (* ... sa *) → if sa_check then chunk else [ ]
  | _ → chunk
  ]
;

end;
```

Module Segmenter

Sanskrit sentence segmenter - analyses (external) sandhi
Runs the segmenting transducer defined by parameter module *Eilenberg*.
Used by *Lexer*, and thus by *Reader* for segmenting, tagging and parsing.
Same logic as old *Segmenter1* but modular with multiple phases
Eilenberg is a finite Eilenberg machine, Control gives command parameters.

In the Sanskrit application, *Word.word* is (reverse of) inflected form.
Id means sandhi is optional. It is an optimisation, since it avoids listing all identity sandhi rules such as *con* | *voy* → *con.voy*. Such rules are nonetheless checked as legitimate.
NB. This segmenter is used by Reader and Parser, but not by Interface, that uses *Graph_segmenter* instead.

```
open Auto; (* Auto *)

module Segment
  (Phases : sig
        type phase
        and phases = list phase;
        value string_of_phase : phase → string;
        value aa_phase : phase → phase;
        value preverb_phase : phase → bool;
        value ii_phase : phase → bool;
        value un_lopa : phase → phase;
        end)
  (Eilenberg : sig
        value transducer : Phases.phase → Auto.auto;
        value initial : bool → Phases.phases;
        value dispatch : bool → Word.word → Phases.phase → Phases.phases;
        value accepting : Phases.phase → bool;
        type input = Word.word (* input sentence represented as a word *)
        and transition = (* junction relation *)
          [ Euphony of Auto.rule (* (w, rev u, v) such that u | v → w *)
          | Id (* identity or no sandhi *)
          ]
        and segment = (Phases.phase × Word.word × transition)
        and output = list segment;
        value validate : output → output; (* consistency check *)
        value sanitize_sa : bool → output → output;
        end)
  (Control : sig value star : ref bool; (* chunk= if star then word+ else word *)
                 value full : ref bool; (* all kridantas and nan cpds if full *)
             end)
  = struct
open Phases;
open Eilenberg;
open Control;
```

The summarizing structure sharing sub-solutions

It represents the union of all solutions

```
value max_input_length = 600
;
type phased_padas = (phase × list Word.word) (* padas of given phase *)
and segments = list phased_padas (* forgetting sandhi *)
;
(* Checkpoints structure (sparse subgraph with mandatory positioned padas) *)
type phased_pada = (phase × Word.word) (* for checkpoints *)
and check = (int × phased_pada × bool) (* checkpoint validation *)
;
value all_checks = ref ([] : list check) (* checkpoints in rest of input *)
and offset_chunk = ref 0
and segmentable_chunk = ref False
and sa_control = ref False
;
(* Used by Rank.segment_chunks_filter *)
value set_offset (offset, checkpoints) = do
  { offset_chunk.val := offset
  ; all_checks.val := checkpoints
  }
;
value set_sa_control b = sa_control.val := b
;
(* The offset permits to align the padas with the input string *)
value offset = fun
  [ Euphony (w, u, v) →
    let off = if w = [] then 1 (* amui/lopa from Lopa/Lopak *)
                        else Word.length w in
    off − (Word.length u + Word.length v)
  | Id → 0
  ]
;
value rec contains phase_w = fun
  [ [] → False
  | [ (phase, word, _) :: rest ] → phase_w = (phase, word) ∨ contains phase_w rest
  ]
;
(* This validation comes from the Summary mode with Interface, which sets checkpoints
that have to be verified for each solution. This is probably temporary, solution ought to be
checked progressively by react, with proper pruning of backtracking. *)
```

```
value check_chunk solution =
  let position = offset_chunk.val
  and checkpoints = all_checks.val in
  check_rec position (List.rev solution) checkpoints
    where rec check_rec index sol checks = match checks with
     [ [] → True (* all checkpoints verified *)
     | [ (pos, phase_word, select) :: more ] →
         (* select=True for check *)
         if index > pos then
           if select then False
           else check_rec index sol more (* checkpoint missed *)
         else match sol with
         [ [] → True (* checkpoint relevant for later chunks *)
         | [ (phase, word, sandhi) :: rest ] →
             let next_index = index + Word.length word + offset sandhi in
             if index < pos then check_rec next_index rest checks
             else let (nxt_ind, ind_sols, next_sols) = all_sol_seg_ind [] sol
                 where rec all_sol_seg_ind stack = fun
                 [ [] → (next_index, stack, [])
                 | [ ((phase2, word2, sandhi2) as seg2) :: rest2 ] →
                   let next_index = pos + Word.length word2 + offset sandhi2 in
                   if next_index = pos then all_sol_seg_ind [ seg2 :: stack ] rest2
                   else (next_index, [seg2 :: stack], rest2)
                 ]
             and (ind_check, next_check) = all_check_ind [] checks
             where rec all_check_ind stack = fun
               [ [] → (stack, [])
               | ([ (pos2, phase_word2, select2) :: more2 ] as orig) →
                   if pos2 = pos then
                       all_check_ind [ (pos2, phase_word2, select2) :: stack ] more2
                   else (stack, orig)
               ] in
             check_sols ind_sols ind_check
             where rec check_sols solspt = fun
               [ [] → check_rec nxt_ind next_sols next_check
               | [ (pos2, phase_word2, select2) :: more2 ] →
                   (select2 = contains phase_word2 solspt)
                   (* Boolean select2 should be consistent with the solutions *)
                   ∧ check_sols solspt more2
               ]
```

```
            ]
        ]
;
```

Checking for legitimate Id sandhi
Uses *sandhis_id* computed by *Compile_sandhi*
This is used to check legitimate Id sandhi.

```
value allowed_trans =
  (Gen.gobble Data.public_sandhis_id_file : Deco.deco Word.word)
;
value check_id_sandhi revl first =
  let match_right allowed = ¬ (List.mem [ first ] allowed) in
  try match revl with
        [ [] → True
        | [ last :: before ] →
            (Phonetics.n_or_f last ∧ Phonetics.vowel first) ∨
            (* we allow an-s transition with s vowel-initial, ignoring nn rules *)
            (* this is necessary not to block transitions from the An phase *)
            let allowed1 = Deco.assoc [ last ] allowed_trans in
            match before with
                [ [] → match_right allowed1
                | [ penu :: _ ] →
                   let allowed2 = Deco.assoc [ last :: [ penu ] ] allowed_trans in
                   match_right allowed2 ∧ match_right allowed1
                ]
        ]
  with [ Not_found → True ]
;
(* Examples: let st1 = Encode.code_revstring "raamas" and st2 = Encode.code_string "asti" in
check_id_sandhi st1 st2 = False ∧ let st1 = Encode.code_revstring "raamaa" and st2 = Encode.cod
check_id_sandhi st1 st2 = False ∧ let st1 = Encode.code_revstring "phalam" and st2 = Encode.cod
check_id_sandhi st1 st2 = True *)

value sandhi_aa = fun
  [ [ 48; 1 ] → [ 1; 2 ] (* a.h | aa → a_aa *)
  | [ 43; 1 ] → Encode.code_string "araa" (* ar | aa → araa *)
  | [ c ] → match c with
                [ 1 | 2 → [ 2 ]
                | 3 | 4 → Encode.code_string "yaa"
                | 5 | 6 → Encode.code_string "vaa"
                | 7 | 8 | 48 → Encode.code_string "raa"
```

```
                 | 9  →  Encode.code_string "laa"
                 | c  →  [ Phonetics.voiced c; 2 ]
                 ]
  | _ → failwith "sandhi_aa"
  ]
;
(* Expands phantom-initial or lopa-initial segments *)
(* phase (aa_phase ph) of "aa" is Pv for verbal ph, Pvkv for nominal ones *)
value accrue ((ph, revword, rule) as segment) previous_segments =
  match Word.mirror revword with
  [ (* First Lopa *)
    [ −2 (* - *) :: r ] → match previous_segments with
      [ [ (phase, pv, Euphony ([], u, [−2])) :: rest ] → (* phase=Pv,Pvkv,Pvkc *)
          let v = match r with [ [ 10 (* e *) :: _ ] → [ 10 ]
                               | [ 12 (* o *) :: _ ] → [ 12 ]
                               | _ → failwith "accrue␣anomaly"
                               ] in
          (* u is a or aa , v is e or o *)
          [ un_lopa_segment :: [ (phase, pv, Euphony (v, u, v )) :: rest ] ]
            where un_lopa_segment = (un_lopa ph, Word.mirror r, rule)
      | _ → failwith "accrue␣anomaly"
      ]
    (* Then phantom phonemes *)
  | [ −3 (* *a *) :: r ] → match previous_segments with
      [ [ (phase, rword, Euphony (_, u, [−3])) :: rest ] →
         let w = sandhi_aa u in
         [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 2 ], [ 2 ], [ 1 ] ))
                        :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
           where new_segment = (ph, Word.mirror [ 1 :: r ], rule)
      | _ → failwith "accrue␣anomaly"
      ]
  | [ −9 (* *A *) :: r ] → match previous_segments with
      [ [ (phase, rword, Euphony (_, u, [−9])) :: rest ] →
         let w = sandhi_aa u in
         [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 2 ], [ 2 ], [ 2 ] ))
                        :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
           where new_segment = (ph, Word.mirror [ 2 :: r ], rule)
      | _ → failwith "accrue␣anomaly"
      ]
  | [ −4 (* *i *) :: r ] → match previous_segments with
```

```
        [ [ (phase, rword, Euphony (_, u, [ -4 ])) :: rest ] →
          let w = sandhi_aa u in
          [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 10 ], [ 2 ], [ 3 ] ))
                          :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
            where new_segment = (ph, Word.mirror [ 3 :: r ], rule)
        | _ → failwith "accrue␣anomaly"
        ]
  | [ -7 (* *I *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [ -7 ])) :: rest ] →
          let w = sandhi_aa u in
          [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 10 ], [ 2 ], [ 4 ] ))
                          :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
            where new_segment = (ph, Word.mirror [ 4 :: r ], rule)
        | _ → failwith "accrue␣anomaly"
        ]
  | [ -5 (* *u *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [ -5 ])) :: rest ] →
          let w = sandhi_aa u in
          [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 12 ], [ 2 ], [ 5 ] ))
                          :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
            where new_segment = (ph, Word.mirror [ 5 :: r ], rule)
        | _ → failwith "accrue␣anomaly"
        ]
  | [ -8 (* *U *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [ -8 ])) :: rest ] →
          let w = sandhi_aa u in
          [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 12 ], [ 2 ], [ 6 ] ))
                          :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
            where new_segment = (ph, Word.mirror [ 6 :: r ], rule)
        | _ → failwith "accrue␣anomaly"
        ]
  | [ -6 (* *r *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [ -6 ])) :: rest ] →
          let w = sandhi_aa u in
          [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 2; 43 ], [ 2 ], [ 7 ] ))
                          :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
            where new_segment = (ph, Word.mirror [ 7 :: r ], rule)
        | _ → failwith "accrue␣anomaly"
        ]
  | [ 123 (* *C *) :: r ] → match previous_segments with
```

```
          [ [ (phase, rword, Euphony (_, u, [ 123 ])) :: rest ] →
            let w = sandhi_aa u in
            [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 2; 22; 23 ], [ 2 ], [ 23 ]))
                             :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
              where new_segment = (ph, Word.mirror [ 23 :: r ], rule)
          | _ → failwith "accrue␣anomaly"
          ]
  | _ → [ segment :: previous_segments ]
  ]
;
(* Now for the segmenter proper *)
type backtrack =
  [ Choose of phase and input and output and Word.word and Auto.choices
  | Advance of phase and input and output and Word.word
  ]
and resumption = list backtrack (* coroutine resumptions *)
;
value finished = ([] : resumption)
;
(* Service routines *)
access : phase → word → option (auto × word)
value access phase = acc (transducer phase) []
    where rec acc state w = fun
        [ [] → Some (state, w) (* w is reverse of access input word *)
        | [ c :: rest ] → match state with
              [ Auto.State (_, deter, _) → match List2.ass c deter with
                    [ Some next_state → acc next_state [ c :: w ] rest
                    | None → None
                    ]
              ]
        ]
;
(* The scheduler gets its phase transitions from Dispatcher.dispatch *)
value schedule phase input output w cont =
  let add phase cont = [ Advance phase input output w :: cont ] in
  let transitions =
    if accepting phase ∧ ¬ star.val then [] (* Word = Sanskrit pada *)
    else dispatch full.val w phase (* iterate Word+ = Sanskrit vaakya *) in
  List.fold_right add transitions cont
```

```
  (* respects dispatch order within a fair top-down search *)
;
(* The tagging transducer interpreter as a non deterministic reactive engine: phase is the
parsing phase input is the input tape represented as a word output is the current result of
type output back is the backtrack stack of type resumption occ is the current reverse access
path in the deterministic part the last argument is the current state of type auto. *)
value rec react phase input output back occ = fun
  [ Auto.State (accept, det, choices) →
    (* we try the deterministic space before the non deterministic one *)
    let deter cont = match input with
      [ [] → continue cont
      | [ letter :: rest ] → match List2.ass letter det with
          [ Some state → react phase rest output cont [ letter :: occ ] state
          | None → continue cont
          ]
      ] in
    let cont = if choices = [] then back (* non deterministic continuation *)
               else [ Choose phase input output occ choices :: back ] in
    (* now we look for - or + segmentation hint *)
    let (keep, cut, input') = match input with
        [ [ 0 :: rest ] → (* explicit "-" compound break hint *)
              (ii_phase phase, True, rest)
        | [ 100 :: rest ] → (* mandatory segmentation + *)
              (True, True, rest)
        | _ → (True, False, input) (* no hint in input *)
        ] in
    if accept ∧ keep then
        let segment = (phase, occ, Id) in
        let out = accrue segment output in
        match validate out with
        [ [] → if cut then continue cont else deter cont (* ZZZ was deter cont *)
        | contracted → match input' with
            [ [] → if accepting phase then (* potential solution found *)
                        emit contracted cont
                   else continue cont
            | [ first :: _ ] → (* we first try the longest matching word *)
                let cont' = schedule phase input' contracted [] cont in
                if cut then continue cont' else
                if check_id_sandhi occ first then (* legitimate Id *)
                    deter cont' else deter cont
```

```
            ]
        ]
    else if cut then continue cont else deter cont
  ]
and choose phase input output back occ = fun
  [ [] → continue back
  | [ ((w, u, v) as rule) :: others ] →
      let cont = if others = [] then back
                 else [ Choose phase input output occ others :: back ] in
      match List2.subtract input w with (* try to read w on input *)
        [ Some rest →
          let segment = (phase, u @ occ, Euphony rule) in
          let out = accrue segment output in
          match validate out with
          [ [] → continue cont
          | contracted →
              if v = [] (* final sandhi *) then
                if rest = [] ∧ accepting phase then (* potential solution found *)
                   emit contracted cont
                else continue cont
              else continue (schedule phase rest contracted v cont)
          ]
        | None → continue cont
        ]
  ]
and continue = fun
  [ [] → None
  | [ resume :: back ] → match resume with
      [ Choose phase input output occ choices →
          choose phase input output back occ choices
      | Advance phase input output occ → match access phase occ with
          [ None → continue back
          | Some (next_state, v) → react phase input output back v next_state
          ]
      ]
  ]
and emit solution cont =
  if check_chunk solution
     then match sanitize_sa sa_control.val solution with
          [ [] → continue cont
```

```
          | ok → Some (ok, cont) (* solution found *)
          ]
  else continue cont
;
value init_segment_initial initial_phases sentence =
  List.map (fun phase → Advance phase sentence [] []) initial_phases
;
value segment1_initial initial_phases sentence =
  continue (init_segment_initial initial_phases sentence)
;
(* Switch according to Complete/Simplify mode, governed by global ref full *)
value init_segment seg = (* do not eta reduce! *)
  init_segment_initial (initial full.val) seg
and segment1 seg =
  segment1_initial (initial full.val) seg
;
end;
```

Module Load_morphs

Load_morphs
Used for loading the (huge) morphology databanks.
Caution. This is an executable, that actually loads the lemmas at link time.

```
open Morphology; (* lemmas *)
```

Morph functor takes its prelude and control arguments as parameters

```
module Morphs
  (Prel : sig value prelude : unit → unit; end)
  (Phases : sig type phase = (* Phases.phase *)
  [ Noun | Noun2
  | Pron
  | Root
  | Inde
  | Absv | Absc | Abso
  | Voca
  | Inv
  | Iic | Iic2
  | Iiif
  | Iiv | Iivv | Iivc
```

```
  | Auxi | Auxiinv | Auxik | Auxiick
  | Ifc | Ifc2
  | Peri (* periphrastic perfect *)
  | Lopa (* e/o conjugated root forms with lopa *)
  | Lopak (* e/o kridantas forms with lopa *)
  | Pv (* Preverb optional before Root or Lopa or mandatory before Abso *)
  | Pvc | Pvv (* privative Abso *)
  | Pvkc | Pvkv (* Preverb optional before Krid or Iik or Lopak *)
  | A | An | Ai | Ani (* privative nan-compounds *)
  | Iicv | Iicc | Ifcv | Ifcc | Nouv | Nouc
  | Krid (* K.ridaantaas - used to be called Parts *)
  | Vok (* K.ridaanta vocatives *)
  | Iik (* K.ridaantaas as left component - used to be called Piic *)
  | Iikv | Iikc | Kriv | Kric | Vocv | Vocc | Vokv | Vokc
  | Iiy | Avy | Inftu | Kama
  | Cache (* Cached lexicon acquisitions *)
  | Cachei (* Cached iic lexicon acquisitions *)
  | Unknown (* Unrecognized chunk *)
  | Comp of (phase × phase) and (* pv *) Word.word and (* root form *) Word.word
  ]; end)
 = struct

open Phases (* phase *)
;
(* Somewhat weird classification of segments according to their construction by Dispatcher.
Preverbed segments may be finite verb forms or kridantas. *)
type tag_sort =
  [ Atomic of lemmas
  | Preverbed of (phase × phase) and (* pv *) Word.word and Word.word and lemmas
  ]
;
(* Fake tags of nan prefixes *)
value nan_prefix = Bare_stem
;
value a_tag = [ ((0, [ ]), [ nan_prefix ]) ]
and an_tag = [ ((0, [ 51 ]), [ nan_prefix ]) ] (* since lexicalized as an#1 *)
(* an_tag has delta = (0,51) since an#1 is the relevant entry. Such values ought to be
parameters of the specific lexicon used. *);
value ai_tag = a_tag (* special for privative abs-tvaa eg akritvaa *)
and ani_tag = an_tag
;
```

```
value unknown_tag = [ ((0, [ ]), [ Unanalysed ]) ]
;
value give_up cat =
  let mess = "Missing␣" ^ cat ^ "␣morphology␣bank" in do
  { Web.abort Html.default_language
                        "System␣error␣-␣please␣report␣-␣" mess
  ; Deco.empty
  }
;
value load_morpho file =
  try (Gen.gobble file : inflected_map)
  with [ _ → do { Prel.prelude (); give_up file } ]
and load_morpho_cache file =
  try (Gen.gobble file : inflected_map)
  with [ _ → Deco.empty ] (* dummy empty morpho lexmap *)
;
(* Loads all morphological databases; Used in Reader, Parser. *)
(* NB both Noun and Noun2 are loaded whether full or not - TODO improve *)
value load_morphs () =
  { nouns = load_morpho Data.public_nouns_file
  ; nouns2 = load_morpho Data.public_nouns2_file
  ; prons = load_morpho Data.public_pronouns_file
  ; roots = load_morpho Data.public_roots_file
  ; krids = load_morpho Data.public_parts_file
  ; voks = load_morpho Data.public_partvocs_file
  ; peris = load_morpho Data.public_peris_file
  ; lopas = load_morpho Data.public_lopas_file
  ; lopaks = load_morpho Data.public_lopaks_file
  ; indes = load_morpho Data.public_inde_file
  ; absya = load_morpho Data.public_absya_file
  ; abstvaa = load_morpho Data.public_abstvaa_file
  ; iics = load_morpho Data.public_iics_file
  ; iics2 = load_morpho Data.public_iics2_file
  ; iifs = load_morpho Data.public_iifcs_file
  ; iiks = load_morpho Data.public_piics_file
  ; iivs = load_morpho Data.public_iivs_file
  ; iiys = load_morpho Data.public_avyayais_file
  ; avys = load_morpho Data.public_avyayafs_file
  ; auxis = load_morpho Data.public_auxis_file
  ; auxiinvs = load_morpho Data.public_auxiinvs_file
```

```
  ; auxiks = load_morpho Data.public_auxiks_file
  ; auxiicks = load_morpho Data.public_auxiicks_file
  ; vocas = load_morpho Data.public_vocas_file
  ; invs = load_morpho Data.public_invs_file
  ; ifcs = load_morpho Data.public_ifcs_file
  ; ifcs2 = load_morpho Data.public_ifcs2_file
  ; inftu = load_morpho Data.public_inftu_file
  ; kama = load_morpho Data.public_kama_file
  ; caches = load_morpho_cache Data.public_cache_file
  ; cacheis = load_morpho_cache Data.public_cachei_file
  }
;
value morpho = load_morphs () (* costly *)
;
value morpho_tags = fun
    [ Noun | Nouv | Nouc → morpho.nouns
    | Pron → morpho.prons
    | Root → morpho.roots
    | Peri → morpho.peris
    | Lopa → morpho.lopas
    | Lopak → morpho.lopaks
    | Inde → morpho.indes
    | Absv | Absc → morpho.abstvaa
    | Abso → morpho.absya
    | Auxi → morpho.auxis
    | Auxiinv → morpho.auxiinvs
    | Auxik → morpho.auxiks
    | Auxiick → morpho.auxiicks
    | Voca | Vocv | Vocc → morpho.vocas
    | Inv → morpho.invs
    | Ifc | Ifcv | Ifcc → morpho.ifcs
    | Iic | Iicv | Iicc → morpho.iics
    | Iiv | Iivv | Iivc → morpho.iivs
    | Iiif → morpho.iifs
    | Iiy → morpho.iiys
    | Avy → morpho.avys
    | Krid | Kriv | Kric → morpho.krids
    | Vok | Vokv | Vokc → morpho.voks
    | Iik | Iikv | Iikc → morpho.iiks
    | Noun2 → morpho.nouns2
```

```
    | Iic2 → morpho.iics2
    | Ifc2 → morpho.ifcs2
    | Inftu → morpho.inftu
    | Kama → morpho.kama
    | Cache → morpho.caches
    | Cachei → morpho.cacheis
    | _ → raise (Control.Anomaly "morpho_tags")
    ]
;
```

Used in Lexer/Reader/Parser and Interface

```
value tags_of phase word =
  match phase with
  [ Pv | Pvkc | Pvkv → failwith "Preverb␣in␣tags_of"
    (* all preverbs ought to have been captured by Dispatcher.validate *)
  | A | Ai → Atomic a_tag
  | An | Ani → Atomic an_tag
  | Unknown → Atomic unknown_tag
  | Comp ((_, ph) as sort) pv form →
      let tag = Deco.assoc form (morpho_tags ph) in
      Preverbed sort pv form tag
(* NB Preverbed comprises tin verbal forms of verbs with preverbs as well as sup kridanta
forms with preverbs. The preverbs are packed in pv. *)
  | _ → Atomic (Deco.assoc word (morpho_tags phase))
    (* NB Atomic comprises tin verbal forms of roots as well as sup atomic forms and all
the pure stems collections Iic Iiv etc. *)
  ]
;

end;
```

Interface for module Lexer

Sanskrit Phrase Lexer

```
open Morphology; (* inflexions lemma morphology *)
open Phases;
open Dispatcher;
open Load_transducers; (* transducer_vect morpho *)
```

```
module Lexer : functor (* takes its prelude and iterator control as parameters *)
  (Prel : sig value prelude : unit → unit; end) → functor
    (Lexer_control : sig
        value star : ref bool; (* chunk = if star then word+ else word *)
        value full : ref bool; (* all kridantas and nan cpds if full *)
        value out_chan : ref out_channel; (* output channel *)
        value transducers_ref : ref Load_transducers.transducer_vect;
        end) → sig
  module Transducers : sig value mk_transducers : bool → transducer_vect; end;
  module Disp : sig
      value accepting : Phases.phase → bool;
      type input = Word.word
      and transition
      and segment = (Phases.phase × Word.word × transition)
      and output = list segment;
      end;
  module Viccheda : sig
      type resumption;
      value continue : resumption → option (Disp.output × resumption);
      value init_segment : Disp.input → resumption;
      value finished : resumption;
      type check = (int × (Phases.phase × Word.word) × bool);
      value all_checks : ref (list check);
      value set_offset : (int × list check) → unit;
      value set_sa_control : bool → unit;
      end;
  value extract_lemma : Phases.phase → Word.word → list lemma;
  value print_segment : int → Disp.segment → int;
(* Exported for Parser *)
  value process_kridanta : Word.word → int → Phases.phase → Word.word →
        Morphology.multitag → (Phases.phase × Word.word × Morphology.multitag);
  value table_morph_of : Phases.phase → string;
  value print_morph : Word.word → bool → int → bool → Word.word → int →
        Morphology.unitag → int;
  value trim_tags : bool →
        Word.word → string → Morphology.multitag → Morphology.multitag;
(* END Exported for Parser *)
  value all_checks : ref (list Viccheda.check);
  value un_analyzable : Word.word → (list Disp.segment × Viccheda.resumption);
```

```
  value set_offset : (int × list Viccheda.check) → unit;
  value print_scl_segment : int → (Phases.phase × Word.word) → int;
  value tags_of : Phases.phase → Word.word →
                    (Load_morphs.Morphs Prel Phases).tag_sort; (* ugly *)
end;
```

Module Lexer

Sanskrit Phrase Lexer - Used by Parser, and Rank for Reader/Regression. Uses Phases from Dispatcher to define phase. Loads the transducers, calls Dispatch to create module Disp. Calls Segment to build Viccheda, the Sanskrit lexer that undoes sandhi in order to produce a padapaa.tha. Exports various print functions for the various modes.

```
open Transduction;
open Canon;
open Skt_morph; (* verbal *)
open Auto.Auto; (* auto State *)
open Segmenter; (* Segment *)
open Dispatcher; (* Dispatch *)
open Word; (* word length mirror patch *)

module Lexer (* takes its prelude and control arguments as module parameters *)
  (Prel : sig value prelude : unit → unit; end)
  (Lexer_control : sig
        value star : ref bool; (* chunk = if star then word+ else word *)
        value full : ref bool; (* all kridantas and nan cpds if full *)
        value out_chan : ref out_channel; (* output channel *)
        value transducers_ref : ref Load_transducers.transducer_vect;
        end) = struct

open Html;
open Web; (* ps pl abort etc. *)
open Cgi;
open Phases; (* Phases *)
open Phases; (* phase generative *)

module Lemmas = Load_morphs.Morphs Prel Phases
;
open Lemmas; (* morpho tag_sort tags_of *)
open Load_transducers; (* transducer_vect Trans *)

module Transducers = Trans Prel;
```

```
module Disp = Dispatch Transducers Lemmas Lexer_control;
open Disp; (* color_of_phase transition trim_tags *)
module Viccheda = Segment Phases Disp Lexer_control;
                  (* init_segment continue set_offset *)
value all_checks = Viccheda.all_checks
and set_offset = Viccheda.set_offset
and set_sa_control = Viccheda.set_sa_control
;
value un_analyzable (chunk : word) =
  ([ (Unknown, mirror chunk, Disp.Id) ], Viccheda.finished)
;
```

Printing

```
value table_morph_of phase = table_begin (background (color_of_phase phase))
;
value print_morph pvs cached seg_num gen form n tag = do
(* n is the index in the list of tags of an ambiguous form *)
  { tr_begin |> ps
  ; th_begin |> ps
  ; span_begin Latin12 |> ps
  ; Morpho_html.print_inflected_link pvs cached form (seg_num, n) gen tag
  ; span_end |> ps
  ; th_end |> ps
  ; tr_end |> ps
  ; n + 1
  }
;
value print_tags pvs seg_num phase form tags =
  let ptag = print_morph pvs (is_cache phase) seg_num (generative phase) form in
  let _ = List.fold_left ptag 1 tags in ()
;
value rec scl_phase = fun
  [ Pv | Pvc | Pvv | Pvkc | Pvkv → "pv"
  | Noun | Noun2 | Nouc | Nouv | Krid | Kriv | Kric | Lopak | Pron | Auxik
        | Cache → "noun"
  | Root | Lopa | Auxi → "root"
  | Inde | Abso | Absv | Absc | Avy | Auxiinv → "inde"
  | Iic | Iic2 | A | An | Iicv | Iicc | Iik | Iikv | Iikc | Iiif | Auxiick
       | Ai | Ani | Cachei → "iic"
  | Iiv | Iivv | Iivc → "iiv"
```

```
  | Iiy → "iiy"
  | Peri → "peri"
  | Inftu → "inftu"
  | Kama → "kama"
  | Voca | Vocv | Vocc | Inv | Vok | Vokv | Vokc → "voca"
  | Ifc | Ifcv | Ifcc | Ifc2 → "ifc"
  | Unknown → "unknown"
  | Comp (_, ph) _ _ → "preverbed␣" ^ scl_phase ph
  ]
;
value print_scl_morph pvs gen form tag = do
  { xml_begin "tag" —> ps
  ; Morpho_scl.print_scl_inflected pvs form gen tag
  ; xml_end "tag" —> ps
  }
;
value print_scl_tags pvs phase form tags =
  let table phase =
      xml_begin_with_att "tags" [ ("phase",scl_phase phase) ] in do
  { table phase |> ps
  ; List.iter (print_scl_morph pvs (generative phase) form) tags
  ; xml_end "tags" —> ps
  }
;
(* These definitions are for export to Parser. They betray a difficuly in the modular organisation, since Parser sees Lexer, but not Load_morphs or Dispatcher. Modules ought to be revised. *)
value tags_of = Lemmas.tags_of
and trim_tags = Disp.trim_tags
;
(* Keeps only relevant tags with trim_tags *)value extract_lemma phase word =
  match tags_of phase word with
  [ Atomic tags → tags
  | Preverbed (_, phase) pvs form tags → (* tags to be trimmed to ok_tags *)
      if pvs = [] then tags
      else trim_tags (generative phase) form (Canon.decode pvs) tags
  ]
;
(* Returns the offset correction (used by SL interface) *)
value process_transition = fun
```

```
  [ Euphony (w, u, v) →
    let off = if w = [] then 1 (* amui/lopa from Lopa/Lopak *)
                         else length w in
    off − (length u + length v)
  | Id → 0
  ]
;
value print_transition = fun
  [ Euphony (w, u, v) → Morpho_html.print_sandhi u v w
  | Id → ()
  ]
;
value process_kridanta pvs seg_num phase form tags = do
  { th_begin |> ps
  ; table_morph_of phase |> pl (* table begin *)
  ; let ok_tags =
        if pvs = [] then tags
        else trim_tags (generative phase) form (Canon.decode pvs) tags in do
        (* NB Existence of the segment guarantees that ok_tags is not empty *)
  { print_tags pvs seg_num phase form ok_tags
  ; table_end |> ps (* table end *)
  ; th_end |> ps
  ; (phase, form, ok_tags)
  }}
;
(* Same structure as Interface.print_morpho *)
value print_morpho phase word = do
  { table_morph_of phase |> pl (* table begin *)
  ; tr_begin |> ps
  ; th_begin |> ps
  ; span_begin Latin12 |> ps
  ; let _ =
      match tags_of phase word with
      [ Atomic tags →
          process_kridanta [] 0 phase word tags
      | Preverbed (_, phase) pvs form tags →
          process_kridanta pvs 0 phase form tags
      ] in ()
  ; span_end |> ps
  ; th_end |> ps
```

```
  ; tr_end |> ps
  ; table_end |> ps (* table end *)
  }
;
(* Segment printing with phonetics without semantics for Reader *)
value print_segment offset (phase, rword, transition) = do
  { "[␣" -> ps
  ; Morpho_html.print_signifiant_off rword offset
  ; print_morpho phase (mirror rword)
  (* Now we print the sandhi transition *)
  ; "&lang;" -> ps (* ¡ *)
  ; let correction = process_transition transition in do
      { print_transition transition
      ; "&rang;]" -> pl (* ¿] *)
      ; html_break |> pl
      ; offset + correction + length rword
      }
  }
;
(* Similarly for scl_plugin mode (without offset and transitions) *)
(* Called from Scl_parser.print_scl_output *)
value print_scl_segment counter (phase, rword) =
  let word = Morpho_html.visargify rword in do
  { let solid = background (Phases.color_of_phase phase) in
    td_begin_class solid |> pl
  ; let ic = string_of_int counter in
    "<input␣type=\"hidden\"␣name=\"field" ^ ic ^ "\"␣value='<form␣wx=\""
         ^ Canon.decode_WX word ^ "\"/>" -> ps
  ; match tags_of phase (mirror rword) with
    [ Atomic tags ->
          print_scl_tags [] phase word tags
    | Preverbed (_, phase) pvs form tags ->
         let ok_tags =
           if pvs = [] then tags
           else trim_tags (generative phase) form (Canon.decode pvs) tags in
         print_scl_tags pvs phase form ok_tags
    ]
  ; "'>" -> ps (* closes ¡input *)
  ; Canon.unidevcode word |> ps
  ; td_end |> ps
```

```
  ; "\n" —> ps
  ; counter + 1
  }
;
end;
```

Module Rank

This library is used by Reader and Regression. It constructs a lexer Lex, indexed on parameters *iterate* and *complete*. Using the module *Constraints* for ranking, it computes a penalty for each solution, and returns all solutions with minimal penalties, with a further preference for the solutions having a minimum number of segments. It manages buckets of solutions ranked by penalties and lengths.

```
open Constraints;
(* roles_of sort_flatten eval_penalty *)

module Prel = struct
 value prelude () = Web.reader_prelude Web.reader_title;
 end (* Prel *)
;
(* Global parameters of the lexer *)
value iterate = ref True (* by default a chunk is a list of words *)
and complete = ref True (* by default we call the fuller segmenter *)
and output_channel = ref stdout (* by default cgi output on standard output *)
;
open Load_transducers; (* transducer_vect dummy_transducer_vect Trans *)

module Lexer_control = struct
 value star = iterate;
 value full = complete;
 value out_chan = output_channel;
 value transducers_ref =
 ref (dummy_transducer_vect : transducer_vect);
end (* Lexer_control *)
;
module Transducers = Trans Prel
;
(* Multi-phase lexer *)
module Lex = Lexer.Lexer Prel Lexer_control (* un_analyzable Disp Viccheda *)
;
```

```
(* Builds the penalty stack, grouping together equi-penalty items. *)
(* Beware, make_groups reverses the list of tags. *)
value make_groups tagger = comp_rec 1 []
  where rec comp_rec seg stack = fun (* going forward in time *)
  [ [] → stack (* result goes backward in time *)
  | [ (phase, rword, _) :: rest ] → (* we ignore euphony transition *)
      let word = Word.mirror rword in
      let lemma = tagger phase word in
      let keep = [ roles_of seg word lemma :: stack ] in
      comp_rec (seg + 1) keep rest
  ]
;
(* Compute minimum penalty in Parse mode *)
value minimum_penalty output =
  let tagger = Lex.extract_lemma
  and out = List.rev output in
  let groups = make_groups tagger out in
  if groups = [] then failwith "Empty penalty stack !!" else
  let sort_groups = sort_flatten groups in
  let min_pen =
    match sort_groups with
    [ [] → failwith "Empty penalty stack"
    | [ (pen, _) :: _ ] → pen
    ] in
  eval_penalty min_pen
;
(* Compound minimum path penalty with solution length *)
value process_output filter_mode ((_, output) as sol) =
  let length_penalty = if filter_mode then List.length output else 0 in
  (pen, sol) where pen =
           let min = if filter_mode ∧ iterate.val then minimum_penalty output
                     else 0 (* keep all *) in
           (min + length_penalty, min)
;
type tagging = (Phases.Phases.phase × Word.word × Lex.Disp.transition)
and solution = list tagging
and ranked_solution = (int (* rank *) × solution)
and bucket = (int (* length *) × list ranked_solution)
;
  (* Solutions None sols saved gives solutions sols within truncation limit; Solutions (Some n) sols saved
```

returns solutions sols within total n, saved is the list of solutions of penalty 0 and worse length penalty. ∗)

```
exception Solutions of option int and list ranked_solution and list bucket
;
(* What follows is absurd combinatorial code linearizing the set of solutions to chunk segmentation, exponential in the length of the chunk. This deprecated code is legacy from the naive parser. It is usable only in demos on small sentences. *)
```

Constructs a triple (p, sols, saved) where sols is the list of all (m,sol) such that ranked sol has minimal length penalty p and absolute penalty m and saved is the list of all ranked sols of length penalty ¿ p and absolute penalty 0, arranged in buckets by increasing length penalty

```
value insert ((pen, min), sol) ((min_pen, sols, saved) as current) =
  if sols = [] then (pen, [ (min, sol) ], [])
  else if pen > min_pen then if min > 0 then current (* sol is thrown away *)
                              else (min_pen, sols, List2.in_bucket pen sol saved)
  else if pen = min_pen then (min_pen, [ (min, sol) :: sols ], saved)
  else (pen, [ (min, sol) ], resc) where resc =
         let save (min, sol) rescued = if min = 0 then [ sol :: rescued ]
                                        else rescued in
         let rescue = List.fold_right save sols [] in
         if rescue = [] then saved else [ (min_pen, rescue) :: saved ]
;
(* Forget absolute penalties of solutions with minimal length penalty. *)
(* Also used to erase constraints - thus do not eta-reduce !!! *)
value trim x = List.map snd x
;
(* overflow is None or (Some n) when n solutions with n¡Web.truncation *)
value emit overflow (_, sols, saved) = (* really weird control structure *)
  raise (Solutions overflow (trim sols) saved)
;
(* Depth-first search in a stack of type list (output × resumption) *)
value dove_tail filter_mode init =
  let init_stack = trim init (* erasing constraints *) in
  dtrec 1 (0, [], []) init_stack (* exits raising exception Solutions *)
  where rec dtrec n kept stack = (* invariant: —stack—=—init—=number of chunks *)
  if n > Web.truncation then emit None kept
  else let full_output = List.fold_right conc stack []
                         where conc (o, _) oo = o @ oo in
       let pen_sol = process_output filter_mode (n, full_output) in
       let kept_sols = insert pen_sol kept in
```

```
      dtrec (n + 1) kept_sols (crank [] init stack)
        where rec crank acc ini = fun
        [ [ (_, c) :: cc ] → match ini with
            [ [ (constraints, i) :: ii ] → do
                { Lex.Viccheda.set_offset constraints
                ; match Lex.Viccheda.continue c with
                  [ Some next → List2.unstack acc [ next :: cc ]
                  | None → crank [ i :: acc ] ii cc
                  ]
                }
            | _ → raise (Control.Anomaly "dove_tail") (* imposs by invariant *)
            ]
        | [] → emit (Some n) kept_sols (* dove-tailing finished *)
        ]
;
(* From Graph_segmenter *)
(* Splitting checkpoints into current and future ones *)
value split_check limit = split_rec []
  where rec split_rec acc checkpts = match checkpts with
      [ [] → (List.rev acc, [])
      | [ ((index, _, _) as check) :: rest ] →
          if index > limit then (List.rev acc, checkpts)
          else split_rec [ check :: acc ] rest
      ]
;
value segment_chunk ((offset, checkpoints), stack) chunk sa_check = do
  { let ini_cont = Lex.Viccheda.init_segment chunk in
    let chunk_length = Word.length chunk in
    let extremity = offset + chunk_length in
    let (local, future) = split_check extremity checkpoints in
    let chunk_constraints = (offset, local) in
    ((succ extremity, future), do
        { Lex.Viccheda.set_offset chunk_constraints (* Sets local constraints *)
        ; Lex.Viccheda.set_sa_control sa_check (* inherit from chunks recursion *)
        ; let res = match Lex.Viccheda.continue ini_cont with
                [ Some c → c
                | None → Lex.un_analyzable chunk
                ] in
          [ (chunk_constraints, res) :: stack ]
        })
```

```
  }
;
value segment_all filter_mode chunks cpts =
  let (_, constrained_segs) = segment_chunks ((0, cpts), []) chunks
  where rec segment_chunks acc = fun
    [ [ (* last *) chunk ] → segment_chunk acc chunk False
    | [ chunk :: rest ] → let sa_check = Phonetics.consonant_starts rest in
                            segment_chunks (segment_chunk acc chunk sa_check) rest
    | [] → acc
    ] in
  dove_tail filter_mode constrained_segs (* infamous exponential dove-tailing *)
;
```

Module Scl_parser

Module *Scl_parser* used as interface with UoH dependency parser

```
open Html;
open Web; (* ps pl etc. *)
open Morphology; (* inflected lemma morphology *)
open Phases; (* Phases *)
open Dispatcher; (* Dispatch *)
open SCLpaths; (* scl_url scl_cgi default_output_font *)
```

```
module Prel = struct
 value prelude () = Web.reader_prelude Web.reader_title;
 end (* Prel *)
;
(* Global parameters of the lexer *)
value iterate = ref True (* by default a chunk is a list of words *)
and complete = ref True (* by default we call the fuller segmenter *)
and output_channel = ref stdout (* by default cgi output on standard output *)
;
module Lexer_control = struct
 value star = iterate;
 value full = complete;
 value out_chan = output_channel;
 value transducers_ref = ref Load_transducers.dummy_transducer_vect;
end (* Lexer_control *)
;
(* Multi-phase lexer *)
```

```
module Lex = Lexer.Lexer Prel Lexer_control (* print_scl_segment *)
;
value print_scl_output output =
  List.fold_left Lex.print_scl_segment 1 (List.rev output)
;
value print_scl_solutions s =
  let _ = print_scl_output s in ()
;
(* Invocation of UoH's CSL parser for dependency graph display *)
value print_scl1 (solutions : list (Phases.phase × Word.word)) =
  let svg_interface_url = scl_cgi ^ "SHMT/" in do
  { ps ("<script␣type=\"text/javascript\"␣src=\"" ^ scl_url ^ "js_files/dragtable.js\"></
  ; ps ("<form␣name=\"word-order\"␣method=\"POST\"␣action␣=␣\""
        ^ svg_interface_url ^ "prog/Word_order/call_heritage2anu.cgi\">\n")
  ; ps ("<table␣class=\"draggable\">")
  ; ps tr_begin
  ; print_scl_solutions solutions
  ; ps ("<td><input␣type=\"hidden\"␣name=\"DISPLAY\"␣value=\"" ^ default_output_font ^"\"/
  ; ps tr_end
  ; ps table_end
  ; ps (submit_input "Submit")
  }
;
(* We restrict to the first solution - TEMPORARY *)
value print_scl sols = match sols with
  [ [] → failwith "No␣sol"
  | [ s :: _ ] → print_scl1 s
  ]
;
(* end; *)
```

Module Reader

CGI-bin sktreader alias Reader for segmentation, tagging and parsing. Reads its input in shell variable *QUERY_STRING* URI-encoded. This CGI is triggered by page *reader_page* created by *sktreader*. It prints an HTML document giving segmentation/tagging of input on stdout.

It invokes Rank to construct the lexer Lex, compute penalties of its various solutions, and return all solutions with minimal penalties.

This is mostly legacy code, being superseded by sharing Interface module

```
open Encode; (* switch_code *)
open Canon;
open Html;
open Web; (* ps pl abort etc. remote_server_host *)
open Cgi; (* get decode_url *)
open Phases; (* Phases *)
open Rank; (* Prel Lex Lexer_control Transducers segment_all iterate Solutions *)
```

Reader interface
Mode parameter of the reader. Controled by service Reader for respectively tagging, shallow parsing, or dependency analysis with the UoH parser.
Note that Summary/Interface is not a Reader/Parser mode.

```
type mode = [ Tag | Parse | Analyse ]
;
value rpc = remote_server_host
and remote = ref False (* local invocation of cgi by default *)
;
value call_parser text sol =
  let cgi = parser_cgi ^ "?" ^ text ^ "p;n=" ^ sol in
              (* same remark as below: this assumes mode is last parameter *)
  let invocation = if remote.val then rpc ^ cgi else cgi in
  anchor Green_ invocation check_sign
;
value call_graph text =
  let cgi = graph_cgi ^ "?" ^ text ^ "g" in
  let invocation = if remote.val then rpc ^ cgi else cgi in
  anchor Green_ invocation check_sign
;
```

Prints n-th solution
ind is relative index within kept, n is absolute index within max

```
value print_solution text ind (n, output) = do
  { pl html_break
  ; pl hr
  ; ps (span_begin Blue_)
  ; ps "Solution␣"; print_int n; ps "␣:␣"
  ; ps (call_parser text (string_of_int n))
  ; ps span_end
  ; pl html_break
```

```
  ; let _ = List.fold_left Lex.print_segment 0 (List.rev output) in
    ind + 1
  }
;
```

General display of solutions, in the various modes

```
value print_sols text revsols = (* stats = (kept,max) *)
  let process_sol = print_solution text in
  let _ = List.fold_left process_sol 1 revsols in ()
;

value display limit mode text saved = fun
  (* saved is the list of all solutions of penalty 0 when filter_mode of process_input is True,
otherwise it lists all the solutions. *)
  [ [] -> do { pl (html_blue "No solution found"); pl html_break }
  | best_sols ->
    let kept = List.length best_sols
    and max = match limit with
              [ Some n -> n | None -> truncation ] in do
    { if mode = Analyse then ()
      else do
          { print_sols text (*kept,max*) best_sols
          ; pl html_break
          ; pl hr
          ; if limit = None then do
                { pl (html_blue "Output truncated at ")
                ; ps (span_begin Red_)
                ; print_int truncation
                ; ps span_end
                ; pl (html_blue " solutions")
                ; pl html_break
                } else ()
          }
    ; match mode with
      [ Parse -> do
          { ps (html_magenta (string_of_int kept))
          ; let mess = " solution" ^ (if kept = 1 then "" else "s")
                                   ^ " kept among " in
            ps (html_blue mess)
          ; ps (html_magenta (string_of_int max))
          ; pl html_break
```

```
        ; if kept < max then do
              { pl (html_blue "Filtering␣efficiency:␣")
              ; let eff = (max − kept) × 100/(max − 1) in
                pl (html_magenta (string_of_int eff ^ "%"))
              } else ()
        ; pl html_break
        ; match saved with
          [ [] → ()
          | [ (_, min_buck) :: _ ] → do
              (* we print only the upper layer of saved *)
            { pl html_break
            ; ps (html_red "Additional␣candidate␣solutions")
            ; let min_sols = List.rev min_buck in
              print_sols text (*kept,max*) min_sols
            ; pl html_break
            }
          ]
        }
    | Analyse → (* best_sols : list (int × list Rank.Lex.Disp.segment) *)
        let solutions = match saved with
            [ [] → best_sols
            | [ (_, min_buck) :: _ ] → List.append best_sols (List.rev min_buck)
            ] in
        let forget_transitions (phase, word, _) = (phase, word) in
        let forget_index (_, segments) = List.map forget_transitions segments in
        let segmentations = List.map forget_index solutions in
        Scl_parser.print_scl segmentations
    | _ → ()
    ]
  }
 ]
;
```

NB This reader is parameterized by an encoding function, that parses the input as a list of words, according to various transliteration schemes. However, the use of "decode" below to compute the romanisation and devanagari renderings does a conversion through VH transliteration which may not be faithful to encodings which represent eg the sequence of phonemes t and h.

```
value process_input text us mode topic (input :string) encode cpts =
  let pieces = Sanskrit.read_raw_sanskrit encode input in
```

```
  let romapieces = List.map Canon.uniromcode pieces in
  let romainput = String.concat "␣" romapieces in
  let chunker = if us (* sandhi undone *) then Sanskrit.read_raw_sanskrit
                else (* blanks non-significant *) Sanskrit.read_sanskrit in
  let chunks = chunker encode input (* normalisation here *) in
  let deva_chunks = List.map Canon.unidevcode chunks in
  let deva_input = String.concat "␣" deva_chunks in do
  { pl (xml_begin_with_att "p" [ ("align","center") ])
  ; ps (div_begin Latin16)
  ; pl (call_graph text ^ "␣Show␣Summary␣of␣Solutions")
  ; pl (xml_end "p")
  ; pl "Input:"
  ; ps (roma16_red_sl romainput) (* romanisation *)
  ; pl hr
  ; pl html_break
  ; pl "Sentence:␣"
  ; ps (deva16_blue deva_input) (* devanagari *)
  ; pl html_break
  ; if mode = Analyse then () else ps "may␣be␣analysed␣as:"
  ; ps div_end (* Latin16 *)
  ; let all_chunks = match topic with
        [ Some topic → chunks @ [ code_string topic ]
        | None → chunks
        ] in
    let filter_mode = mode = Parse ∨ mode = Analyse in
    try segment_all filter_mode all_chunks cpts with
        [ Solutions limit revsols saved →
            let sols = List.rev revsols in
            display limit mode text saved sols
        ]
  }
;
value sort_check cpts =
  let compare_index (a, _, _) (b, _, _) = compare a b in
  List.sort compare_index cpts
;
```

Standard format of cgi arguments

```
value arguments translit lex cache st us cp input topic abs cpts =
    "t=" ^ translit
```

```
  ^ ";lex=" ^ lex
  ^ ";cache=" ^ cache
  ^ ";st=" ^ st
  ^ ";us=" ^ us
  ^ ";cp=" ^ cp
  ^ ";text=" ^ input
  ^ ";topic=" ^ topic
  ^ ";abs=" ^ abs
  ^ ";cpts=" ^ Checkpoints.string_points cpts
  ^ ";mode=" (* mode to be filled later *)
;
(* Faster if only segmenting: no loading of nouns_file, roots_file, ... *)
value reader_engine () = do
  { Prel.prelude ()
  ; let query = try Sys.getenv "QUERY_STRING" with
                    [ Not_found → failwith "Environment␣required" ] in
    let env = create_env query in
    let url_encoded_input = get "text" env ""
    and url_encoded_mode = get "mode" env "p"
    and url_encoded_topic = get "topic" env ""
    and st = get "st" env "t" (* default vaakya rather than isolated pada *)
    and cp = get "cp" env "t" (* default Complete mode *)
    and us = get "us" env "f" (* default input sandhied *)
    and translit = get "t" env Paths.default_transliteration
    and lex = get "lex" env Paths.default_lexicon
    and cache = get "cache" env "f" in
    let () = cache_active.val := cache
    and abs = get "abs" env "f" (* default local paths *) in
    let lang = Html.language_of lex
    and input = decode_url url_encoded_input (* unnormalized string *)
    and uns = us ="t" (* unsandhied vs sandhied corpus *)
    and encode = switch_code translit (* encoding as a normalized word *)
    and () = Html.toggle_lexicon lex
    and () = if abs ="t" then remote.val := True else () (* Web service mode *)
    and () = if st ="f" then iterate.val := False else () (* word stemmer *)
    and () = let full = (cp ="t") in do
          { Lexer_control.full.val := full
          ; Lexer_control.transducers_ref.val := Transducers.mk_transducers full
          }
    and mode = match decode_url url_encoded_mode with
```

```
        [ "t" → Tag
        | "p" → Parse
        | "o" → Analyse (* Analyse mode of UoH parser *)
        | s → raise (Failure ("Unknown␣mode␣" ^ s))
        ]
  (* Contextual information from past discourse *)
  and topic_mark = decode_url url_encoded_topic in
  let topic = match topic_mark with
        [ "m" → Some "sa.h"
        | "f" → Some "saa"
        | "n" → Some "tat"
        | _ → None
        ]
  and abortl = abort lang
  and checkpoints = (* checkpoints for graph *)
     try let url_encoded_cpts = List.assoc "cpts" env in (* do not use get *)
           Checkpoints.parse_cpts (decode_url url_encoded_cpts)
     with [ Not_found → [] ] in
  let cpts = sort_check checkpoints in
  try let text = arguments translit lex cache st us cp url_encoded_input
                              url_encoded_topic abs checkpoints in do
        { (* Now we call the lexer *)
           process_input text uns mode topic input encode cpts
        ; pl hr
        ; pl html_break
        ; close_page_with_margin ()
        ; page_end lang True
        }
  with
  [ Sys_error s → abortl Control.sys_err_mess s (* file pb *)
  | Stream.Error s → abortl Control.stream_err_mess s (* file pb *)
  | Encode.In_error s → abortl "Wrong␣input␣" s
  | Exit (* Sanskrit *) → abortl "Wrong␣character␣in␣input" ""
  | Invalid_argument s → abortl Control.fatal_err_mess s (* sub *)
  | Failure s → abortl Control.fatal_err_mess s
  | End_of_file → abortl Control.fatal_err_mess "EOF" (* EOF *)
  | Not_found → let s = "You␣must␣choose␣a␣parsing␣option" in
                              abortl "Unset␣button␣in␣form␣-␣" s
  | Control.Fatal s → abortl Control.fatal_err_mess s (* fatal *)
  | Control.Anomaly s → abortl Control.anomaly_err_mess ("Anomaly:␣" ^ s)
```

```
    | _ → abortl Control.anomaly_err_mess "Unexpected␣anomaly"
    ]
  }
;
value safe_engine lang =
(* In case of error, we lose the current language of the session *)
try reader_engine () with
  [ Failure s → abort lang Control.fatal_err_mess s
  | _ → abort lang Control.anomaly_err_mess "Unexpected␣anomaly"
  ]
;
(* Should always produce a compliant xhtml page *)
safe_engine Html.default_language
;
```

Module Parser

CGI-bin callback for shallow syntax analysis
Parser is similar to Reader, but it is invoked from the green hearts in the output of the reader, in order to give the semantic analysis of a specific solution. It basically replays reading until this specific solution

```
open Encode;
open Canon;
open Html;
open Web; (* ps pl abort truncation etc. remote_server_host *)
open Cgi; (* get *)
open Checkpoints;
open Phases.Phases; (* generative *)
open Scl_parser; (* Interface with UoH dependency parser *)
```

```
module Prel = struct (* Parser's lexer prelude *)
```

prelude is executed by Lexer when initialisation of transducers fails

```
value prelude () = do
  { pl http_header
  ; page_begin parser_meta_title
  ; pl (body_begin Chamois_back)
  ; if scl_toggle then (* external call SCL (experimental) *)
        pl (javascript (SCLpaths.scl_url ^ javascript_tooltip))
    else ()
```

```
  ; pl parser_title
  ; open_page_with_margin 15
  }
;
end (* Prel *)
;
value iterate = ref True (* by default we read a sentence (list of words) *)
and complete = ref True (* by default we call the fuller segmenter *)
and output_channel = ref stdout (* by default cgi output *)
;
open Load_transducers; (* transducer_vect dummy_transducer_vect Trans *)

module Lexer_control = struct
 value star = iterate;
 value full = complete;
 value out_chan = output_channel;
 value transducers_ref =
 ref (dummy_transducer_vect : transducer_vect);
end (* Lexer_control *)
;
module Transducers = Trans Prel
;
module Lex = Lexer.Lexer Prel Lexer_control
(* print_proj print_segment_roles print_ext_segment extract_lemma *)
;
(* Printing functions *)

value table_labels = table_begin (background Pink)
;
value print_labels tags seg_num = do
    { ps th_begin (* begin labels *)
    ; pl table_labels
    ; let print_label n _ = do
        { ps (cell (html_red (string_of_int seg_num ^ "." ^ string_of_int n)))
        ; n + 1
        } in
      let _ = List.fold_left print_label 1 tags in ()
    ; ps table_end
    ; ps th_end (* end labels *)
    }
;
```

```
value rec color_of_role = fun (* Semantic role of lexical category *)
  [ Pv | Pvkc | Pvkv | Iic | Iic2 | Iik | Voca | Inv | Iicv | Iicc
  | Iikv | Iikc | Iiif | A | An | Vok | Vokv | Vokc | Vocv | Vocc | Iiy
  | Iiv | Iivv | Iivc | Peri | Auxiick | Pvv | Pvc → Grey
  | Noun | Noun2 | Nouv | Nouc | Krid | Kriv | Kric | Ifc | Ifcv | Ifcc | Ifc2
  | Pron | Kama | Lopak | Auxik → Cyan (* Actor or Predicate *)
  | Root | Lopa | Auxi → Pink (* abs-tvaa in Inde *) (* Process *)
  | Abso | Absv | Absc | Inde | Avy | Ai | Ani | Inftu | Auxiinv (* Circumstance
*)
    → Lavender
  | Unknown | Cache | Cachei → Grey
  | Comp (_, ph) _ _ → color_of_role ph
  ]
and table_role_of phase = table_begin (background (color_of_role phase))
;
(* syntactico/semantical roles analysis, function of declension *)
value print_roles pr_sem phase tags form = do
    { ps th_begin
    ; pl (table_role_of phase)
    ; let pr_roles (delta, sems) = do
        { ps tr_begin
        ; ps th_begin
        ; let word = Word.patch delta form in
          pr_sem word sems
        ; ps th_end
        ; ps tr_end
        } in
      List.iter pr_roles tags
    ; ps table_end
    ; ps th_end
    }
;
(* Segment printing without phonetics with semantics for Parser *)
value print_segment_roles print_sems seg_num (phase, rword, _) =
  let word = Word.mirror rword in do
  { Morpho_html.print_signifiant_yellow rword
  ; let (decl_phase, form, decl_tags) = match Lex.tags_of phase word with
        [ Atomic tags →
            Lex.process_kridanta [] seg_num phase word tags
        | Preverbed (_, phase) pvs form tags →
```

```
          Lex.process_kridanta pvs seg_num phase form tags
        ] in do
      { print_labels decl_tags seg_num
      ; print_roles print_sems decl_phase decl_tags form
      }
  }
;
value project n list = List.nth list (n − 1) (* Ocaml's nth starts at 0 *)
;
value print_uni_kridanta pvs phase word multitags (n, m) =
  let (delta, polytag) = project n multitags in
  let unitag = [ project m polytag ] in do
      { th_begin |> ps
      ; Lex.table_morph_of phase |> pl (* table of color of phase begins *)
      ; let _ = (* print unique tagging *)
         Lex.print_morph pvs False 0 (generative phase) word 0 (delta, unitag) in ()
      ; table_end |> ps (* table of color of phase ends *)
      ; th_end |> ps
      }
;
value print_projection phase rword index = do
  { tr_begin |> ps (* tr begins *)
  ; Morpho_html.print_signifiant_yellow rword
  ; let word = Word.mirror rword in
     match Lex.tags_of phase word with
     [ Atomic tags → print_uni_kridanta [] phase word tags index
     | Preverbed (_, phase) pvs form tags →
          (* we trim out lemmas inconsistent with preverb assignment to form *)
          let trim = Lex.trim_tags (generative phase) form (Canon.decode pvs) in
          print_uni_kridanta pvs phase form (trim tags) index
     ]
  ; tr_end |> ps (* tr ends *)
  }
;
value print_proj phase rword = fun
   [ [] → failwith "Projection␣missing"
   | [ n_m :: rest ] → do
         { print_projection phase rword n_m
         ; rest (* returns the rest of projections stream *)
         }
```

```
    ]
;
(* End Printing functions *)
value rpc = remote_server_host
and remote = ref False (* local invocation of cgi by default *)
;
open Skt_morph;
open Inflected;
open Constraints; (* roles_of sort_flatten extract *)
open Paraphrase; (* display_penalties print_sem print_role *)
value query = ref "" (* ugly - stores the query string *)
;
value set_query q = query.val := q (* Parser.parser_engine *)
;
(* Duplicated from Rank *)
value make_groups tagger = comp_rec 1 []
  where rec comp_rec seg stack = fun (* going forward in time *)
  [ [] → stack (* result goes backward in time *)
  | [ (phase, rword, _) :: rest ] → (* we ignore euphony transition *)
       let word = Word.mirror rword (* segment is mirror word *) in
       let lemma = tagger phase word in
       let keep = [ roles_of seg word lemma :: stack ] in
       comp_rec (seg + 1) keep rest
  ]
;
value print_sols sol =
  let xmlify_call sol = (* sol in reverse order *)
    let projections = List.fold_left extract "" sol in
    let invoke = parser_cgi ^ "?" ^ query.val ^ ";p=" ^ projections in
    anchor Green_ invoke heart_sign in do
  { ps html_break
  ; List.iter print_role (List.rev sol)
  ; ps (xmlify_call sol)
  ; ps html_break
  }
;
value monitoring = True (* We show explicitly the penalty vector by default *)
;
value display_penalty p = "Penalty␣" ^
```

```
  if monitoring then Constraints.show_penalty p
  else string_of_int (Constraints.eval_penalty p)
;
value print_bucket (p, b_p) = do
  { ps html_break
  ; ps (html_green (display_penalty p))
  ; ps html_break
  ; List.iter print_sols b_p
  }
;
value analyse query output =
  let tagger = Lex.extract_lemma in
  let groups = make_groups tagger output in
  let sorted_groups = sort_flatten groups in
  let (top_groups, threshold) = truncate_groups sorted_groups in do
  { pl (xml_empty "p")
  ; let find_len = fun
        [ [ (_,[ a :: _ ]) :: _ ] → List.length a
        | _ → 0
        ] in
    pl (xml_empty_with_att "input" (* Final call to Parser for display *)
            [ ("type","submit"); ("value","Submit");
              ("onclick","unique('" ^ parser_cgi ^ "?" ^ query
              ^ ";p=','" ^ string_of_int (find_len top_groups) ^ "')" )
            ] ^ html_break)
  ; pl (xml_empty "p")
  ; if scl_toggle then (* Call SCL parser *)
        let segments = List.map (fun (ph, w, _) → (ph, w)) output in
        Scl_parser.print_scl [ List.rev segments ]
        else ()
    ; List.iter print_bucket top_groups
  ; match threshold with
    [ None → ()
    | Some p → do
        { html_break |> ps
        ; html_red ("Truncated␣penalty␣" ^ string_of_int p ^ "␣or␣more") |> ps
        ; html_break |> ps
        }
    ]
  }
```

```
;
value print_sems word morphs = do
  { span_begin Latin12 |> ps
  ; "{␣" —> ps
  ; let bar () = "␣|␣" —> ps
    and sem = Canon.decode word in
    List2.process_list_sep (print_sem sem) bar morphs
  ; "␣}" —> ps
  ; span_end |> ps
  }
;
value print_out seg_num segment = do
  (* Contrarily to Reader, we discard phonetic information. *)
  { tr_begin |> ps
  ; print_segment_roles print_sems seg_num segment
  ; tr_end |> ps
  ; seg_num + 1
  }
;
value rec print_project proj = fun
    [ [] → match proj with
             [ [] → () (* finished, projections exhausted *)
             | _ → failwith "Too␣many␣projections"
             ]
    | [ (phase, rword, _) :: rest ] → (* sandhi ignored *)
      let new_proj = print_proj phase rword proj in
      print_project new_proj rest
    ]
;
exception Truncation (* raised if more solutions than Web.truncation *)
;
(* Replay reader until solution index - quick and dirty way to recreate it. *)
(* Follows the infamous exponential Rank.dove_tail. *)
value dove_tail_until sol_index init =
  let init_stack = List.map (fun (_, s) → s) init (* erasing constraints *) in
  dtrec 1 (0, [], []) init_stack
  where rec dtrec n kept stack = (* invariant: —stack—=—init—=number of chunks *)
  if n = Web.truncation then raise Truncation
  else if n = sol_index then (* return total output *)
          List.fold_right conc stack []
```

```
          where conc (o, _) oo = o @ oo
  else dtrec (n + 1) kept (crank [] init stack)
         where rec crank acc ini = fun
         [ [ (_, c) :: cc ] → match ini with
            [ [ (constraints, i) :: ii ] → do
               { Lex.Viccheda.set_offset constraints
               ; match Lex.Viccheda.continue c with
                 [ Some next → List2.unstack acc [ next :: cc ]
                 | None → crank [ i :: acc ] ii cc
                 ]
               }
            | _ → raise (Control.Anomaly "dove_tail_until")
            ]
         | [] → raise Truncation
         ]
;
(* Following two functions are same as in Rank *)
(* Splitting checkpoints into current and future ones *)
value split_check limit = split_rec []
   where rec split_rec acc checkpts = match checkpts with
      [ [] → (List.rev acc, [])
      | [ ((index, _, _) as check) :: rest ] →
          if index > limit then (List.rev acc, checkpts)
          else split_rec [ check :: acc ] rest
      ]
;
value segment_chunk ((offset, checkpoints), stack) chunk sa_check = do
  { let ini_cont = Lex.Viccheda.init_segment chunk in
    let chunk_length = Word.length chunk in
    let extremity = offset + chunk_length in
    let (local, future) = split_check extremity checkpoints in
    let chunk_constraints = (offset, local) in
    ((succ extremity, future), do
       { Lex.Viccheda.set_offset chunk_constraints (* Sets local constraints *)
       ; Lex.Viccheda.set_sa_control sa_check (* inherit from chunks recursion *)
       ; let res = match Lex.Viccheda.continue ini_cont with
              [ Some c → c
              | None → Lex.un_analyzable chunk
              ] in
         [ (chunk_constraints, res) :: stack ]
```

```
      })
  }
;
(* Follows logic of Rank.segment_all until solution reached *)
value segment_until sol_index chunks cpts =
  let (_, constrained_segs) = segment_chunks ((0, cpts), []) chunks
  where rec segment_chunks acc = fun
    [ [ (* last *) chunk ] → segment_chunk acc chunk False
    | [ chunk :: rest ] → let sa_check = Phonetics.consonant_starts rest in
                          segment_chunks (segment_chunk acc chunk sa_check) rest
    | [] → acc
    ] in
  dove_tail_until sol_index constrained_segs
;
```

Printing stuff

```
value stamp =
  "Heritage" ^ "␣" ^ Date.version
;
value print_validate_button query =
  let cgi = parser_cgi ^ "?" ^ query ^ ";validate=t" in
  let invocation = if remote.val then rpc ^ cgi else cgi in
  anchor Green_ invocation check_sign
;
(* Follows Reader.process_input *)
value process_until sol_index query topic mode_sent translit sentence
                    cpts us encode proj sol_num query do_validate =
  let pieces = Sanskrit.read_raw_sanskrit encode sentence in
  let romapieces = List.map Canon.uniromcode pieces in
  let romasentence = String.concat "␣" romapieces in
  let chunker = if us then Sanskrit.read_raw_sanskrit
                else Sanskrit.read_sanskrit in
  let chunks = chunker encode sentence in
  let devachunks = List.map Canon.unidevcode chunks in
  let devasentence = String.concat "␣" devachunks in do
  { pl html_break
  ; let lex_stamp = "Lexicon:␣" ^ stamp in
    ps (html_green lex_stamp) (* in order to keep relation corpus/lexicon *)
  ; pl html_break
  ; pl hr
```

```
  ; pl html_break
  ; ps (roma16_red_sl romasentence) (* romanisation *)
  ; pl html_break
  ; ps (deva16_blue devasentence) (* devanagari *)
  ; pl html_break
  ; let all_chunks = match topic with
        [ Some topic → chunks @ [ code_string topic ]
        | None → chunks
        ] in
    try let output = segment_until sol_index all_chunks cpts in
        let solution = List.rev output in do
        { pl html_break
        ; pl (xml_begin_with_att "table" [ noborder; padding10; spacing5 ])
        ; match proj with
          [ None → let _ = List.fold_left print_out 1 solution in ()
          | Some triples → print_project triples solution
          ]
        ; ps table_end
        ; match proj with
          [ None → analyse query solution
          | Some p → ()
          ]
        }
    with [ Truncation → pl (html_red "Solution␣not␣found" ^ html_break) ]
  }
;
value sort_check cpts =
  let compare_index (a, _, _) (b, _, _) = compare a b in
  List.sort compare_index cpts
;
value parser_engine () = do
(* Replays Reader until given solution - dumb but reliable *)
  { Prel.prelude ()
  ; let query = Sys.getenv "QUERY_STRING" in
    let alist = create_env query in
    let url_encoded_input = get "text" alist ""
    and url_encoded_sol_index = get "n" alist "1"
    and url_encoded_topic = get "topic" alist ""
    and st = get "st" alist "t"
```

```
and cp = get "cp" alist "t"
and us = get "us" alist "f"
and translit = get "t" alist Paths.default_transliteration
and lex = get "lex" alist Paths.default_lexicon
and abs = get "abs" alist "f" (* default local paths *) in
let lang = language_of lex
and input = decode_url url_encoded_input (* unnormalized string *)
and uns = us ="t" (* unsandhied vs sandhied corpus *)
and mode_sent = st ="t" (* default sentence mode *)
and encode = Encode.switch_code translit (* encoding as a normalized word *)
and () = toggle_lexicon lex
and () = if abs ="t" then remote.val := True else () (* Web service mode *)
and () = if st ="f" then iterate.val := False else () (* word stemmer *)
and () = let full = (cp ="t") in do
      { Lexer_control.full.val := full
      ; Lexer_control.transducers_ref.val := Transducers.mk_transducers full
      }
and sol_index = int_of_string (decode_url url_encoded_sol_index)
(* For Validate mode, register number of solutions *)
and sol_num = int_of_string (get "allSol" alist "0")
(* Only register this solution if validate is true *)
and do_validate = get "validate" alist "f"
(* Contextual information from past discourse *)
and topic_mark = decode_url url_encoded_topic in
let topic = match topic_mark with
    [ "m" → Some "sa.h"
    | "f" → Some "saa"
    | "n" → Some "tat"
    | _ → None
    ] in
 (* Corpus interaction disabled (* File where to store locally the taggings - only for Station
platform *) let corpus_file = (× optionally transmitted by argument "out_file" ×) try let file_name =
) in   Some file_name  with [ Not_found  →  Some regression_file_name ] in *)
(* Regression disabled let () = if Paths.platform = "Station" then match corpus_file with [ Some file_
 let regression_file = var_dir ^ file_name ^ ".txt" in  output_channel.val := open_out_gen [ Open_wr
 None → () ] else () in *)
let proj = (* checks for parsing mode or final unique tags listing *)
    try let url_encoded_proj = List.assoc "p" alist in (* do not use get *)
        Some (parse_proj (decode_url url_encoded_proj))
    with [ Not_found → do
```

```
                    { set_query query (* query regurgitated - horror *)
                    ; None
                    }
                ]
   and checkpoints = (* checkpoints for graph *)
     try let url_encoded_cpts = List.assoc "cpts" alist in (* do not use get *)
         parse_cpts (decode_url url_encoded_cpts)
     with [ Not_found → [] ] in
   let cpts = sort_check checkpoints in
   try do
     { process_until sol_index query topic mode_sent translit input
                    cpts uns encode proj sol_num query do_validate
     ; close_page_with_margin ()
     ; let bandeau = ¬ (Gen.active proj) in
       page_end lang bandeau
     }
   with [ Stream.Error _ → abort lang "Illegal␣transliteration␣" input ]
 }
;
value safe_engine () =
  let abor = abort default_language in
  try parser_engine () with
  [ Sys_error s → abor Control.sys_err_mess s (* file pb *)
  | Stream.Error s → abor Control.stream_err_mess s (* file pb *)
  | Encode.In_error s → abor "Wrong␣input␣" s
  | Exit (* Sanskrit *) → abor "Wrong␣character␣in␣input␣-␣" "use␣ASCII"
  | Invalid_argument s → abor Control.fatal_err_mess s (* sub *)
  | Failure s → abor Control.fatal_err_mess s (* anomaly *)
  | End_of_file → abor Control.fatal_err_mess "EOF" (* EOF *)
  | Not_found (* assoc *) → let s = "You␣must␣choose␣a␣parsing␣option" in
                            abor "Unset␣button␣in␣form␣-␣" s
  | Control.Fatal s → abor Control.fatal_err_mess s (* anomaly *)
  | Control.Anomaly s → abor Control.fatal_err_mess ("Anomaly:␣" ^ s)
  | _ → abor Control.fatal_err_mess "Unexpected␣anomaly"
  ]
;
safe_engine () (* Should always produce a valid HTML page *)
;
```

Interface for module Constraints

Constraints machinery

```
open Skt_morph;
open Morphology; (* inflexions *)
type noun_role =
  [ Subject of person and number (* agent of active or patient of passive *)
  | Object (* patient or goal of active or adverb of manner *)
  | Instrument (* agent of passive or instrument of active or adverb of manner *)
  | Destination (* receiver or goal *)
  | Origin (* origin of action or adverb of manner *)
  | Possessor (* dual role as verb complement or noun attribution *)
  | Circumstance (* adverb of time or location *)
  ]
and demand = list noun_role
;
type mood =
  [ Indicative
  | Imper of bool (* True: Imperative False: Injunctive *)
  ]
;
(* Part of speech *)
type pos =
  [ Process of demand and mood (* roles governed by a verb form *)
  | Subprocess of demand (* verbal subphrase *)
  | Actor of noun_role and gender and number (* noun form with morphology *)
  | Addressee (* vocative *)
  | Tool of tool (* grammatical word *)
  | Compound (* iic *)
  | Number of gender and case and number (* number (gender for eka) *)
  | Ignored (* indeclinable not known as tool *)
  ]
(* Combinatorial tools *)
and tool =
  [ Coordination (* ca *)
  | Post_instrument (* sahaa1 vinaa prep *)
  | Not_Post_instrument (* sahaa1 adv *)
  | Prohibition (* maa *)
  | Post_genitive (* varam *)
  | Todo (* to avoid warning *)
```

```
  ]
;
type aspect =
  [ Imperfectif (* active or middle indicative *)
  | Impersonal (* intransitive passive *)
  | Perfectif (* transitive passive *)
  | Statif (* factitive *)
  ]
;
type regime =
  [ Transitive (* transitive verbs in active and middle *)
  | Intransitive (* intransitive verbs in active and middle *)
  | Factitive (* impersonal - no subject *)
  | Quotative (* aahur - it is said *)
(*— Bitransitive - use of transitive with 2 accusatives *)
(*— Regime of (list case * list case) - specific regime - unused so far *)
  ]
;
value root_regime : string → regime
;
(* compute aspect, demand and mood of a verbal finite form *)
value regime : string → (conjugation × paradigm) → (aspect × demand × mood)
;
type label = (int × int × int) (* (segment number, homonym index, tag index) *)
and roles = list (label × pos)
;
value roles_of : int → list int → list ((int × list int) × inflexions) → roles
;
type penalty =
  [ Sentence of (int × int × int × int)
  | Copula of (int × int × int × int × int)
  | NP of penalty
  ]
;
value eval_penalty : penalty → int
;
value show_penalty : penalty → string
;
type flattening = list (penalty × list roles)
;
```

```
value sort_flatten : list roles → flattening
;
value truncate_groups : flattening → (flattening × option int)
;
value extract : string → (label × pos) → string
;
```

Module Constraints

Syntactico/semantic analysis and penalty computations.
This is the 2005 design of a constraint machinery working on some kind of linear logic graph matching of semantic roles. Verbs are assigned arities of needed complements, seen as roles with a negative polarity. It does not really use the karaka theory, the role of a nominative is mediated through the voice. This is very primitive, and works only for toy examples. It merely gives a proof of feasability. A more serious machinery should work on discourse, deal with ellipses, and possibly use optimality theory with matrix computations.
We need to enrich this parser with kridantas which have their own aaka.mk.saa , eg participles. Then we must recognize that certain passive constructs, such ppp, may be use in the active sense to indicate past e.g. with verbs of mouvement

```
open Skt_morph;
open Morphology; (* inflexion_tag *)
open Html;
```

Constraints analysis
Nouns

```
type noun_role = (* not karaka *)
  [ Subject of person and number (* agent of active or patient of passive *)
  | Object (* patient or goal of active or adverb of manner *)
  | Instrument (* agent of passive or instrument of active or adverb of manner *)
  | Destination (* receiver or goal *)
  | Origin (* origin of action or adverb of manner *)
  | Possessor (* dual role as verb complement or noun attribution *)
  | Circumstance (* adverb of time or location *)
  ]
and demand = list noun_role
;
value person_of_subst = fun
  [ "aham" → First | "tvad" → Second | _ → Third ]
;
```

```
value gram_role num entry = fun
  [ Nom → Subject (person_of_subst entry) num
  | Acc → Object (* Patient or adverb of manner *)
  | Ins → Instrument (* Agent or adverb of instrument *)
  | Dat → Destination
  | Abl → Origin
  | Gen → Possessor
  | Loc → Circumstance
  | Voc → failwith "Unexpected vocative (gram_role)"
  ]
and case_of = fun (* inverse of gram_role *)
  [ Subject _ _ → Nom
  | Object → Acc
  | Instrument → Ins
  | Destination → Dat
  | Origin → Abl
  | Possessor → Gen
  | Circumstance → Loc
  ]
;
type mood =
  [ Indicative
  | Imper of bool (* True: Imperative False: Injunctive *)
  ]
;
(* mood processing - pertains to maa management *)
value ini_mood = (0,0)
and add_mood m moods = match m with
   [ Imper b → let (imp, inj) = moods in if b then (imp + 1, inj) else (imp, inj + 1)
   | _ → moods
   ]
;
```

Part of speech

```
type pos =
  [ Process of demand and mood (* roles governed by a verb form *)
  | Subprocess of demand (* verbal subphrase *)
  | Actor of noun_role and gender and number (* noun form with morphology *)
  | Addressee (* vocative *)
  | Tool of tool (* grammatical word *)
```

```
  | Compound (* iic *)
  | Number of gender and case and number (* number (gender for eka) *)
  | Ignored (* indeclinable not known as tool *)
  ]
(* Combinatorial tools *)
and tool =
  [ Coordination (* ca *)
  | Post_instrument (* saha vinaa prep *)
  | Not_Post_instrument (* saha adv *)
  | Prohibition (* maa *)
  | Post_genitive (* varam TODO *)
  | Todo (* to avoid warning *)
  ]
;
(* Verb valencies - Very experimental. *)
(* The serious version will have to make computations with preverbs *)
(* and will accommodate several sememes with different valencies for a given lexeme - e.g. "dhaav#1.1" intransitive, "dhaav#1.2" transitive. The paraphrase will be associated with sememes and not just lexemes. *)
type regime =
  [ Transitive (* transitive verbs in active and middle *)
  | Intransitive (* intransitive verbs in active and middle *)
  | Factitive (* impersonal - no subject *)
  | Quotative (* aahur - it is said *)
(*— Bitransitive - use of transitive with 2 accusatives *)
(*— Regime of (list case * list case) - specific regime - unused so far *)
(* Actually a root should have a valency list like 0; 1; 2 for "bhaa.s" *)
  ]
;
```

We simplify by assuming equal valency of atmanepade and parasmaipade.
Also we assume (to be revised) that valency is independent of preverb.

```
value root_regime = fun
  [ (* akarmaka roots, checked by Pawan Goyal *)
    (* more exactly, these are the roots that may be used akarmaka *)
    "an#2" | "as#1" | "as#2" | "aas#2" | "iih" | "uc" | "uurj#1" | ".rdh"
  | "edh" | "kamp" | "kaaz" | "kuc" | "ku.t" | "kup" | "kul" | "kuuj" | "k.lp"
  | "krii.d" | "krudh#1" | "klid" | "kvath" | "k.sar" | "k.si" | "k.su"
  | "k.sudh#1" | "k.subh" | "khel" | "gaj" | "garj" | "gard" | "galbh" | "gu~nj"
  | "gur" | "g.rr#2" | "glai" | "gha.t" | "gha.t.t" | "ghuur.n" | "cakaas"
```

```
  | "ca~nc" | "cal" | "cit#1" | "ce.s.t" | "jan" | "jaag.r" | "jiiv" | "j.rmbh"
  | "j.rr" | "jyaa#1" | "jvar" | "jval" | ".dii" | "tan#2" | "tam" | "tu.s"
  | "t.r.s#1" | "trap" | "tras" | "tvar" | "tsar" | "dak.s" | "dal" | "das"
  | "dah#1" | "dih" | "diik.s" | "diip" | "du.s" | "d.rh" | "dev#1" | "dyut#1"
  | "draa#1" | "draa#2" | "dhaav#1" | "dhru" | "dhvan" | "dhv.r" | "na.t"
  | "nand" | "nard" | "naz#1" | "nah" | "nii#1" | "n.rt" | "pat#1" | "pii"
  | "puuy" | "p.r#2" | "pyaa" | "prath" | "phal" | "ba.mh" | "bal" | "bha.n.d"
  | "bhand" | "bha.s" | "bhaa#1" | "bhaas#1" | "bhii#1" | "bhuj#1" | "bhuu#1"
  | "bhra.mz" | "bhram" | "bhraaj" | "ma.mh" | "majj" | "mad#1" | "mud#1" | "muh"
  | "muurch" | "m.r" | "m.rdh" | "mre.d" | "mlaa" | "yabh" | "yas" | "yu#2"
  | "yudh#1" | "ra~nj" | "ra.n" | "ram" | "raaj#1" | "ru" | "ruc#1" | "rud#1"
  | "ru.s#1" | "ruh#1" | "lag" | "lamb" | "lal" | "las" | "vak.s" | "vas#1"
  | "vah#1" (* nadii vahati *) | "vaa#2" | "vaaz" | "vij" | "vip" | "viz#1"
  | "v.rt#1" | "v.rdh#1" | "vyath" | "zak" | "zad" | "zam#1" | "zii#1" | "ziil"
  | "zuc#1" | "zudh" | "zubh#1" | "zu.s" | "zuu" | "zram" | "zrambh" | "zvas#1"
  | "zvit#1" | "sap#1" | "saa#1" | "sidh#1" | "sur" | "skhal" | "stan" | "stu"
  | "stubh" | "sthaa#1" | "snih#1" | "snu" | "spand" | "spardh" | "sphaa"
  | "sphu.t" | "sphur" | "smi" | "syand" | "sra.ms" | "svap" | "svar#1"
  | "svar#2" | "had" | "has" | "hikk" | "h.r.s" | "hras" | "hraad" | "hrii#1"
  | "hlaad" | "hval" → Intransitive
  | "baa.sp" | "zyaam" (* nominal verbs *) → Intransitive
  | "v.r.s" → Factitive
  | "ah" → Quotative
  | _ → (* sakarmaka in all usages *) Transitive
(* But "bhaa.s" is Transitive, even though he may be used with 0 or 2 objects *)
(* Thus a penalty should not occur if he has no object or 2 objects *)
  ]
;
(* But valency may depend on gana for the present system *)
value root_regime_gana k = fun
  [ "i" → match k with [ 2 | 4 → Intransitive | _ → Transitive ]
  | "daa#1" → match k with [ 3 → Intransitive | _ → Transitive ]
  | "b.rh#1" → match k with [ 1 → Intransitive | _ → Transitive ]
  | "maa#1" → match k with [ 2 → Intransitive | _ → Transitive ]
  | "tap" | "pac" | "raadh" | "svid#2" → match k with
      [ 4 → Intransitive | _ → Transitive ]
  | root → root_regime root
  ]
;
(* Certain roots marked as Transitive are in fact Intransitive for some of their meanings:
```

"kruz" | "gh.r" | "jak.s" | "ji" | "t.rp#1" | "d.rp" | "dhva.ms" | "pi~nj" | "bhas" | "mand#1" | "radh" | "lafgh" | "lu.n.th" | "vii#1" | "zumbh" | "sad#1" | "su#2" | "svan" | "ha.th" | "hi#2" | "hu.n.d" | "huu" When used intransitively, the parser will look for a missing object and may penalize correct sentences. For roots marked as Intransitive, but nonetheless used transitively in a sentence, the parser will consider their accusative object, in the active voice, as an adverb, but no penalty will incur. NB. dvikarmaka roots are just treated as Transitive in this version. *)

```
value agent_of_passive = fun
  [ "vid#2" → [] (* ellipsed impersonal agent "it is known that" *)
  | _ → [ Instrument ] (* Agent at instrumental in passive voice *)
  ]
;
(* The following type actually combines aspect, voice and mood *)
type aspect =
  [ Imperfectif (* active or middle indicative *)
  | Impersonal (* intransitive passive *)
  | Perfectif (* transitive passive *)
  | Statif (* factitive *)
  ]
;
(* Computes aspect valency and mood of a verbal finite form as a triple *)
value regime entry (cj, t) =
  (* conjugation cj and possible preverb sequence ignored in first version *)
  let regime = root_regime entry in (* TODO dependency on k *)
  match t with
    [ Conjug _ Passive | Presentp _ →
        let aspect = match regime with
                       [ Intransitive → Impersonal
                       | Factitive → Statif
                       | _ → Perfectif
                       ]
        and valency = agent_of_passive entry in
        (aspect, valency, Indicative)
    | Conjug t _ →
        let aspect = if regime = Factitive ∨ regime = Quotative then Statif
                     else Imperfectif
        and valency = match regime with
                        [ Transitive → [ Object ] | _ → [] ]
        and mood = match t with [ Injunctive _ → Imper False
                                | _ → Indicative
```

```
                          ] in
     (aspect, valency, mood)
   | Presenta k m | Presentm k m → (* on affine le regime gana et mode *)
     let regime = root_regime_gana k entry in
     let aspect = if regime = Factitive ∨ regime = Quotative then Statif
                  else (* if m=Optative then Statif (* NEW bruyaat *) else *) Imperfectif
     and valency = match regime with
                   [ Transitive → [ Object ] | _ → [] ]
     and mood = match m with
                [ Imperative → Imper True
                | _ → Indicative (* now, only Imperative for Present *)
                ] in
     (aspect, valency, mood)
   | Perfut _ → (if regime = Factitive then Statif else Imperfectif,
                 match regime with [ Transitive → [ Object ] | _ → [] ],
                 Indicative)
   ]
;
value get_fin_roles entry f n p =
  let (aspect, valency, mood) = regime entry f in
  let demand = match aspect with
      [ Statif | Impersonal → valency
      | _ → [ Subject p n :: valency ] (* anaphoric subject reference *)
    (* — Imperfectif -¿ Subject p n :: valency (* subject is agent *) — Perfectif -¿
Subject p n :: valency (* subject is goal/patient *) *)
      ] in
  Process demand mood
and get_abs_roles entry =
  let demand = match root_regime entry with
      [ Intransitive | Factitive → []
      | _ → [ Object ]
      ] in
  Subprocess demand
;
(* Present participle active defines an auxiliary clause, like Absolutive *)
(* It denotes simultaneity rather than sequentiality/causality *)
value is_ppra (_, v) = match v with (* TEMP *)
  [ Ppra _ → True | _ → False ]
;
(* get_roles assigns roles to morphological items. *)
```

```
(* Some tool words are processed here and numbers are recognized. *)
value get_roles entry = fun
  [ Part_form v g n c
               → if c = Voc then Addressee
                 else if is_ppra v then Subprocess [] (* should lookup root *)
                 else Actor (gram_role n entry c) g n (* beware n duplication *)
  | Noun_form g n c
               → if c = Voc then Addressee
                 else if g = Deictic Numeral ∨ entry ="eka" then Number g c n
                 else Actor (gram_role n entry c) g n (* beware n duplication *)
  | Verb_form f n p → get_fin_roles entry f n p
  | Abs_root _ → get_abs_roles entry
  | Ind_form Conj → match entry with
                    [ "ca" → Tool Coordination
                    | _ → Ignored (* TODO vaa etc *)
                    ]
  | Ind_form Prep → if entry = "saha" ∨ entry = "vinaa" ∨ entry = "satraa"
                       then Tool Post_instrument
                    else Ignored
  | Ind_form Adv → if entry = "saha" then Tool Not_Post_instrument
                   else Ignored
  | Ind_form Abs → get_abs_roles entry
  | Ind_form Part → match entry with
                    [ "maa#2" → Tool Prohibition
                    | _ → Ignored
                    ]
  | Bare_stem → Compound
  | _ → Ignored
  ]
;
(* Used in Parser, Reader *)
value roles_of seg word tags =
  let distrib (sub, res) (delta, morphs) =
    let entry = Canon.decode (Word.patch delta word) in
    let roles = List.map (get_roles entry) morphs in
    let (_, r) = List.fold_left label (1, res) roles
            where label (i, l) role = (i + 1, [ ((seg, sub, i), role) :: l ]) in
    (sub + 1, r) in
  let (_, rls) = List.fold_left distrib (1, []) tags in
  rls
```

```
;
(* We flatten the role matrix into a list of sequences. *)
(* This is potentially exponential, since we multiply choices. *)
type label = (int × int × int) (* (segment number, homonymy index, tag index) *)
and roles = list (label × pos)
;
(* Combinator flatten_add is for the brave. Do not attempt to understand this code if you
have not already mastered flatten and flatteni above. *)
(* flatten_add : list roles → list roles *)
value rec flatten_add = fun (* arg goes backward in time *)
  [ [] → [ [] ]
  | [ l :: r ] → (* l: roles *)
    let flatr = flatten_add r
    and distr res f = (* f: roles *)
        let prefix acc x = [ [ x :: f ] :: acc ] in
        let result = List.fold_left prefix [] l in
        result @ res in
    List.fold_left distr [] flatr
  ]
;
(* Tool words as semantic combinators - reverse role stream transducers *)
```

Coordination tool

```
exception No_coord (* Coordination failure *)
;
(* future deictic gender context, here assumed all male *)
value context d = Mas
;
(* abstract interpretation of coordination *)
value merge = fun (* persons priorities *)
  [ First → fun _ → First
  | Second → fun [ First → First | _ → Second ]
  | Third → fun [ First → First | Second → Second | _ → Third ]
  ]
and add = fun (* numbers additions *)
  [ Plural | Dual → fun _ → Plural
  | Singular → fun [ Plural | Dual → Plural | Singular → Dual ]
  ]
;
```

```
value rec dom = fun (* male dominance *)
  [ Mas → fun _ → Mas
  | Fem → fun [ Mas → Mas | Deictic d → dom Fem (context d) | _ → Fem ]
  | Neu → fun [ Deictic d → context d | g → g ]
  | Deictic d → dom (context d)
(* Unsatisfactory - numbers ought to be treated as Neu. *)
(* The gender is used only for possible adjective agreement, not for verb government *)
  ]
;
```

(∗ Coordination recognizes noun phrases (N = IIc*.Noun@nom) N1 N2 ca ... Np ca N1 ca N2 ca ... Np ca with N = C* S C = iic, S = Subst NB negation not yet accounted for (naca etc); also is missing N1 N2 ... Np ca avec Ni homogène en nb - adjectival cascade. We synthesize a multiple homogeneous substantive in the output stream ∗)

```
value coord_penalty = 1
;
(* removing possible compound prefixes *)
value end_coord kar acc p g n = rem_iic
  where rec rem_iic cur = match cur with
   [ [ Compound :: rest ] → rem_iic rest
   | _ → match kar with (* Synthesis of compound kar *)
       [ Subject _ _ → ([ Actor (Subject p n) g n :: acc ], cur)
       | kar → ([ Actor kar g n :: acc ], cur)
       ]
   ]
;
value agree_deictic g = fun
   [ Deictic _ → True
   | g1 → g = g1
   ]
;
(* Remove compound formation and possible adjectival number word. *)
value skim c g n context = skim_rec context
  where rec skim_rec con = match con with
   [ [ Compound :: rest ] → skim_rec rest (* skip possible iic - compounding *)
   | [ Number g1 c1 n1 :: rest ] →
       if agree_deictic g g1 ∧ c = c1 ∧ n = n1 (* agreement of Number *)
          then rest
          else raise No_coord
   | _ → con
   ]
```

```
;
value rec coord1 kar acc p g n = fun
  (* searching for closest noun phrase *)
  [ [] → raise No_coord
  | [ np :: rest ] → match np with
     [ Actor (Subject p1 _) g1 n1 → match kar with
        [ Subject _ _ →
          coord2 kar acc (merge p p1) (dom g g1) (add n n1) rest
        | _ → raise No_coord
        ]
     | Actor k g1 n1 when k = kar →
        coord2 kar acc Third (dom g g1) (add n n1) rest
     | _ → raise No_coord
     ]
  ]
and coord2 kar acc p g n cur = match cur with
     (* searching for previous noun phrases *)
  [ [] → raise No_coord
  | [ np :: rest ] → match np with
     [ Actor (Subject p1 _) g1 n1 → match kar with
        [ Subject _ _ →
          let before = skim Nom g1 n1 rest in
          end_coord kar acc (merge p p1) (dom g g1) (add n n1) before
        | _ → raise No_coord
        ]
     | Actor k g1 n1 →
        if k = kar then let before = skim (case_of k) g1 n1 rest in
                        (* additive interpretation of ca *)
                        end_coord kar acc Third (dom g g1) (add n n1) before
                   else raise No_coord
     | Tool Coordination → coord1 kar acc p g n rest (* iterate the tool *)
     | _ → raise No_coord
     ]
  ]
;
(* Coordination: the ca tool constructs a composite tag from its predecessors *)
value coordinate acc = fun
  (* searching for first noun phrase *)
  [ [] → raise No_coord
  | [ np :: rest ] → match np with
```

```
    [ Actor (Subject p1 _ as kar) g1 n1 →
         let before = skim Nom g1 n1 rest in
         coord2 kar acc p1 g1 n1 before
    | Actor kar g1 n1 →
         let before = skim Nom g1 n1 rest in
         coord2 kar acc Third g1 n1 before
    | _ → raise No_coord
    ]
   ]
;
(* Bumping the current penalty by a given malus *)
value penalize malus (roles, pen) = (roles, pen + malus)
;
```

Ugly experimental management of "maa" negative particle - temporary

```
value maa_counter = ref 0
;
value reset_maa () = maa_counter.val := 0
;
(* apply tools on the list of roles, read from right to left *)
(* tools are piped as role streams transducers - res is accumulated output of the form (list
role,penalty). *)
value rec use_tools res = fun
  [ [] → res
  | [ r :: iroles ] → match r with
      [ Tool Coordination → (* ca *)
        try let (oroles, penalty) = res in
             let (result, left) = coordinate oroles iroles in
             use_tools (result, penalty) left with
        [ No_coord → use_tools (penalize coord_penalty res) iroles ]
      | Tool Post_instrument → (* saha vinaa prep *)
        match iroles with
        [ [] → penalize 1 res
        | [ r :: previous ] → match r with
             [ Actor Instrument _ _ → use_tools res previous (* ¡i.-saha¿ *)
             | _ → use_tools (penalize 1 res) iroles
             ]
        ]
      | Tool Not_Post_instrument → (* saha adv *)
        match iroles with
```

```
        [ [] → res
        | [ r :: _ ] → match r with
              [ Actor Instrument _ _ → use_tools (penalize 1 res) iroles
              | _ → use_tools res iroles
              ]
        ]
      | Tool Prohibition → (* maa *) do
        { maa_counter.val := maa_counter.val + 1
        ; res
        }
      | Tool _ (* not yet implemented *)
      | Ignored (* noop *)
      | Compound → use_tools res iroles (* compounds are skipped *)
        (* ordinary roles are processed as Identity tools *)
      | _ → let (oroles, p) = res in (* otherwise we take role as is *)
              use_tools ([ r :: oroles ], p) iroles
      ]
  ]
;
```

We construct a list *neg* of expected *noun_roles*, a list *pos* of available ones, a counter *pro* of processes, a boolean *subpro* indicating the need of a finite verb form, a mood integrator *moo*

```
value process_role (neg, pos, pro, subpro, moo) role =
  match role with
    [ Process noun_roles m → (noun_roles @ neg, pos, pro + 1, subpro, add_mood m moo)
      (* pro+1 is problematic, it does not account for relative clauses *)
    | Subprocess noun_roles → (noun_roles @ neg, pos, pro, True, moo)
    | Actor noun_role gender number →
          (neg, [ (noun_role, number, gender) :: pos ], pro, subpro, moo)
    | _ → (neg, pos, pro, subpro, moo)
    ]
;
exception Missing
;
type triple = (noun_role × number × gender)
      (* NB there is redundancy in the case (Subject p n,n',g) since n'=n *)
;
value subject_agreement (noun_role, _, _) p n =
  noun_role = Subject p n
;
```

```
(* Tries to find a matching agent: looks into the list of leftover given roles for an expected
agent with person p and number n, returns it paired with the rest of given roles if found,
raises exception Missing otherwise *)
value remove_subj p n = remrec []
  where rec remrec acc = fun
    [ [] → raise Missing
    | [ triple :: rest ] →
      if subject_agreement triple p n then (triple, List2.unstack acc rest)
      else remrec [ triple :: acc ] rest
    ]
;
(* Tries to find a matching role for a non-agent noun_role *)
value remove_matching kar = remrec []
  where rec remrec acc = fun
    [ [] → raise Missing
    | [ ((k, _, _) as triple) :: rest ] →
      if k = kar then
          (triple, List2.unstack acc rest) (* we choose latest matching *)
      else remrec [ triple :: acc ] rest
    ]
;
(* missing is the list of missing expectancies noun_roles taken is the list of found expectancies
noun_roles left is the list of found unexpected noun_roles *)
value process_exp (missing, taken, left) = fun
  (* for each expected noun_role we look for a matching given one *)
  [ Subject p n → (* verb subject has p and n *)
    try let (found, remain) = remove_subj p n left in
        (missing, [ found :: taken ], remain)
    with [ Missing → (missing, taken, left) ] (* subject is optional *)
  | kar → try let (found, remain) = remove_matching kar left in
                (missing, [ found :: taken ], remain)
             with [ Missing → ([ kar :: missing ], taken, left) ] (* mandatory *)
  ]
;
(* Contraction corresponding to agreement between phrase-forming chunks. *)
(* Items agreeing with an already taken item are removed from leftovers. *)
value contract taken = List.fold_left filter []
  where filter left triple = if List.mem triple taken then left
                                                else [ triple :: left ]
;
```

```
(* Penalty parameters in need of tuning by training *)
value missing_role_penalty _ = 1
and excess_subject_penalty = 1
and np_penalty = 2
and absol_penalty = 2 (* absolutive without finite verb *)
;
(* remaining extra nominatives give penalty *)
value count_excess pen = fun
  [ (Subject p n, _, _) → pen + excess_subject_penalty
  | triple → pen (* taken as adverbs or genitive noun phrases *)
  ]
;
(* We count all persons with same person and number *)
value count_subj persons = fun
  [ Subject p n → List2.union1 (p, n) persons
  | _ → persons
  ]
;
value count_missing pen k = pen + missing_role_penalty k
;
value missing_penalty = List.fold_left count_missing 0
and excess_penalty = List.fold_left count_excess 0
;
type penalty =
  [ Sentence of (int × int × int × int)
  | Copula of (int × int × int × int × int)
  | NP of penalty
  ]
;
value rec show_penalty = fun (* explicit vector for debug *)
  [ Sentence (p1, p2, p3, p4) →
      "S(" ^ string_of_int p1 ^ "," ^ string_of_int p2 ^ ","
           ^ string_of_int p3 ^ "," ^ string_of_int p4 ^ ")"
  | Copula (p1, p2, p3, p4, p5) →
      "C(" ^ string_of_int p1 ^ "," ^ string_of_int p2 ^ ","
           ^ string_of_int p3 ^ "," ^ string_of_int p4 ^ ","
           ^ string_of_int p5 ^ ")"
  | NP p → string_of_int np_penalty ^ "+" ^ show_penalty p
  ]
;
```

```
(* Ad-hoc linear penalty function - to be optimized by corpus training *)
value rec eval_penalty = fun
  [ Sentence (pen1, pen2, pen3, pen4) → pen1 + pen2 + pen3 + pen4
  | Copula (pen1, pen2, pen3, pen4, pen5) → pen1 + pen2 + pen3 + pen4 + pen5
  | NP pen → np_penalty + eval_penalty pen
  ]
;
value balance_process pro subpro =
  if pro > 1 then pro − 1 (* TEMP, to be adjusted with relative clauses *)
  else if pro = 0 then if subpro then absol_penalty else 0
  else 0
;
(* Delay dealing with nominatives in order to favor Acc over Nom for neuters *)
value sort_kar = sort_rec [] [] 0
  where rec sort_rec nomins others n = fun
  [ [] → (List2.unstack others nomins, n)
  | [ (Subject _ _ as kar) :: rest ] →
       sort_rec [ kar :: nomins ] others (n + 1) rest
  | [ kar :: rest ] → sort_rec nomins [ kar :: others ] n rest
  ]
;
value check_sentence pen1 neg pos pro subpro =
  let (missing, taken, left) = List.fold_left process_exp ([], [], pos) neg in
  let contracted = contract taken left in
  let pen2 = missing_penalty missing
  and pen3 = excess_penalty contracted
  and pen4 = balance_process pro subpro in
  Sentence (pen1, pen2, pen3, pen4)
;
(* Given a list of remaining roles, tries to find a matching Subject; returns (missing,taken,rest)
where either taken is the singleton found, rest is the list of remaining roles, and missing is
empty, or else taken is empty, missing is the singleton not found, and rest is all roles *)
value process_exp_g p n roles =
  let remove_matching = remrec []
  where rec remrec acc = fun
    [ [] → raise Missing
    | [ triple :: rest ] → match triple with
        [ (Subject p n′, _, _) when n′ = n → (triple, List2.unstack acc rest)
                                  (* NB there is no mandatory concord of genders *)
        | _ → remrec [ triple :: acc ] rest
```

```
        ]
    ] in
(* we look for a matching nominative *)
    try let (found, remain) = remove_matching roles in
        ([], [ found ], remain)
    with [ Missing → (* First and Second persons Subjects are optional *)
           if p = First ∨ p = Second then ([], [], roles)
           else ([ Subject p n ], [], roles) ]
;
value check_copula_sentence pen1 p n pos subpro =
  let (missing, taken, left) = process_exp_g p n pos in
  let contracted = contract taken left in
  let pen2 = missing_penalty missing
  and pen3 = excess_penalty contracted
  and pen4 = if subpro then absol_penalty else 0 in
  Copula (pen1, pen2, pen3, pen4, 0)
;
(* get_predicate returns the first available Subject (backward from the end) if there is one,
else raises Missing *)
value get_predicate = search_subject []
  where rec search_subject acc = fun
  [ [] → raise Missing
  | [ ((kar, _, _) as triple) :: rest ] → match kar with
      [ Subject p n → (p, n, List2.unstack acc rest)
      | _ → search_subject [ triple :: acc ] rest
      ]
  ]
(* NB adding a topic amounts to replacing get_predicate pos by (Third,Singular,pos) below
*)
;
(* We enforce that maa must correspond to an injunctive or an imperative and that in-
junctives occur only with maa. TODO: allow also optative, subjunctive and augmentless
imperfect with maa. UGLY *)
value rec mood_correction (imp, inj) pen =
  let maa_tokens = maa_counter.val in (* counted by Prohibition tool *)
  let maa_pen = if maa_tokens > imp + inj then maa_tokens − (imp + inj)
                else if inj > maa_tokens then inj − maa_tokens
                else 0 in match pen with
  [ Sentence (p1, p2, p3, p4) → Sentence (p1, p2, p3, p4 + maa_pen) (* p4=0 *)
  | Copula (p1, p2, p3, p4, p5) → Copula (p1, p2, p3, p4, p5 + maa_pen) (* p5=0 *)
```

```
  | NP pen → mood_correction (imp, inj) pen (* weird *)
  ]
;
value inspect pen (neg, pos, pro, subpro, md) = mood_correction md pens
  where pens =
  if neg = [] (* no overt verb, we conjecture copula (pro=0) *) then
      try let (p, n, rest) = get_predicate pos in
            check_copula_sentence pen p n rest subpro
      with [ Missing → (* maybe noun phrase *)
                 NP (check_sentence pen [] pos pro subpro) ] (* 2+ *)
  else check_sentence pen neg pos pro subpro (* verbal predicate exists *)
;
(* We compute a path penalty by applying use_tools from right to left to the given path,
then iterating process_role on the resulting roles, then inspecting and weighting the resulting
constraints *)
value penalty rev_path =
  let right_left_roles = List.map snd rev_path in do
  { reset_maa () (* horreur *)
  ; let (roles, pen_tools) = use_tools ([], 0) right_left_roles in
    let constraints =
          List.fold_left process_role ([], [], 0, False, ini_mood) roles in
    inspect pen_tools constraints
  }
;
type flattening = list (penalty × list roles)
;
(* We flatten all choices in the chunked solution *)
(* sort_flatten : list roles → flattening *)
value sort_flatten groups = (* groups goes backward in time *)
  let parses = flatten_add groups in (* each parse goes backward in time *)
  let insert_in sorted_buckets rev_path =
        let p = penalty rev_path in
        let ep = eval_penalty p in
        ins_rec [] sorted_buckets
        where rec ins_rec acc = fun
        [ [] → List2.unstack acc [ (p, [ rev_path ]) ]
        | ([ ((pk, b_k) as b) :: r ] as buckets) →
          let ek = eval_penalty pk in (* recomputation to avoid *)
          if ek = ep then List2.unstack acc [ (p, [ rev_path :: b_k ]) :: r ]
          else if ek < ep then ins_rec [ b :: acc ] r
```

```
      else List2.unstack acc [ (p, [ rev_path ]) :: buckets ]
    ] in
  let sort_penalty = List.fold_left insert_in [] in
  sort_penalty parses
;
```

Output truncated to avoid choking on immense web page. Returns penalty threshold if truncation. Used in Reader and Parser

```
value truncate_groups buckets = match buckets with
  [ [ best :: [ next :: rest ] ] →
    let top = [ best; next ] in (* top 2 buckets *)
    let threshold =
      match rest with
      [ [] → None
      | [ (p, _) :: _ ] → Some (eval_penalty p)
      ] in
    (top, threshold)
  | _ → (buckets, None)
  ]
;
value extract str ((seg, sub, ind), _) = (* construct tag projections *)
  let m = string_of_int sub (* segment number seg is redundant *)
  and n = string_of_int ind in
  let proj = m ^ "," ^ n in
  if str ="" then proj else proj ^ "|" ^ str
;
end;
```

Module Multilingual

This module gives headers of grammar engines Declension and Conjugation both in roman font (English at present) and devanagarii font (Sanskrit)

```
open Skt_morph;
open Html;
```

```
type font = [ Deva | Roma ]
;
value font_of_string = fun
  [ "deva" → Deva
  | "roma" → Roma
  | f → failwith ("Unknown␣font␣" ^ f)
  ]
and string_of_font = fun
  [ Deva → "deva"
  | Roma → "roma"
  ]
;
value gender_caption gender = fun
  [ Roma → span3_center (match gender with
      [ Mas → "Masculine"
      | Fem → "Feminine"
      | Neu → "Neuter"
      | Deictic _ → "All"
      ])
  | Deva → deva12_blue_center (Encode.skt_raw_to_deva (match gender with
      [ Mas → "pumaan"
      | Fem → "strii"
      | Neu → "napu.msakam"
      | Deictic _ → "sarvam"
      ]))
  ]
and number_caption number = fun
  [ Roma → span3_center (match number with
      [ Singular → "Singular"
      | Dual → "Dual"
      | Plural → "Plural"
      ])
  | Deva → deva12_blue_center (Encode.skt_raw_to_deva (match number with
      [ Singular → "eka"
      | Dual → "dvi"
      | Plural → "bahu"
      ]))
  ]
and case_caption case = fun
  [ Roma → span3_center (match case with
```

```
      [ Nom → "Nominative"
      | Acc → "Accusative"
      | Ins → "Instrumental"
      | Dat → "Dative"
      | Abl → "Ablative"
      | Gen → "Genitive"
      | Loc → "Locative"
      | Voc → "Vocative"
      ])
  | Deva → deva12_blue_center (Encode.skt_raw_to_deva (match case with
      [ Nom → "prathamaa"
      | Acc → "dvitiiyaa"
      | Ins → "t.rtiiyaa"
      | Dat → "caturthii"
      | Abl → "pa~ncamii"
      | Gen → ".sa.s.thii"
      | Loc → "saptamii"
      | Voc → "sambodhanam"
      ]))
  ]
;
value compound_name = fun
  [ Roma → span3_center "Compound"
  | Deva → deva12_blue_center (Encode.skt_raw_to_deva "samaasa")
  ]
and avyaya_name = fun
  [ Roma → span3_center "Adverb"
  | Deva → deva12_blue_center (Encode.skt_raw_to_deva "avyaya")
  ]
;
value western_pr = fun
  [ Present → "Present"
  | Imperative → "Imperative"
  | Optative → "Optative"
  | Imperfect → "Imperfect"
  ]
and indian_pr = fun
  [ Present → "la.t"
  | Imperative → "lo.t"
  | Optative → "vidhilif"
```

```
  | Imperfect → "laf"
  ]
;
value western_tense = fun
  [ Future → "Future"
  | Perfect → "Perfect"
  | Aorist _ → "Aorist"
  | Injunctive _ → "Injunctive"
  | Conditional → "Conditional"
  | Benedictive → "Benedictive"
  | Subjunctive → "Subjunctive"
  ]
and indian_tense = fun
  [ Future → "l.r.t"
  | Perfect → "li.t"
  | Aorist _ → "luf"
  | Injunctive _ → "aagamaabhaavayuktaluf"
  | Conditional → "l.rf"
  | Benedictive → "aaziirlif"
  | Subjunctive → "le.t"
  ]
;
type gentense =
  [ Present_tense of pr_mode
  | Other_tense of tense
  ]
;
value tense_name gentense = fun
  [ Deva → deva16_blue_center (Encode.skt_raw_to_deva s)
             where s = match gentense with
    [ Present_tense pr → indian_pr pr
    | Other_tense t → indian_tense t
    ]
  | Roma → span2_center s where s = match gentense with
    [ Present_tense pr → western_pr pr
    | Other_tense t → western_tense t
    ]
  ]
and perfut_name = fun
  [ Deva → deva16_blue_center (Encode.skt_raw_to_deva "lu.t")
```

```
  | Roma → span2_center "Periphrastic␣Future"
  ]
;
value person_name person = fun
  [ Deva → let deva_person = match person with
                  [ First → "uttama"
                  | Second → "madhyama"
                  | Third → "prathama"
                  ] in
             deva12_blue_center
                  (Encode.skt_raw_to_deva deva_person)
  | Roma → let roma_person = match person with
                  [ First → "First"
                  | Second → "Second"
                  | Third → "Third"
                  ] in
             span3_center roma_person
  ]
;
value conjugation_name conj = fun
  [ Deva → let indian_conj = match conj with
                  [ Primary → "apratyayaantadhaatu"
                  | Causative → ".nic"
                  | Intensive → "yaf"
                  | Desiderative → "san"
                  ] in
             deva16_blue_center (Encode.skt_raw_to_deva indian_conj)
  | Roma → let western_conj = match conj with
                  [ Primary → "Primary"
                  | Causative → "Causative"
                  | Intensive → "Intensive"
                  | Desiderative → "Desiderative"
                  ] in
             span2_center (western_conj ^ "␣Conjugation")
  ]
;
value conjugation_title narrow = fun
  [ Deva → Encode.skt_to_deva "dhaatuvibhakti"
  | Roma → if narrow then "Conjugation"
             else "The␣Sanskrit␣Grammarian:␣Conjugation"
```

```
  ]
and declension_title narrow = fun
  [ Deva → Encode.skt_to_deva "praatipadikavibhakti"
  | Roma → if narrow then "Declension"
           else "The␣Sanskrit␣Grammarian:␣Declension"
  ]
and conjugation_caption = fun
  [ Deva → Encode.skt_to_deva "tifantaavalii"
  | Roma → "Conjugation␣tables␣of"
  ]
and declension_caption = fun
  [ Deva → Encode.skt_to_deva "subantaavalii"
  | Roma → "Declension␣table␣of"
  ]
and participles_caption = fun
  [ Deva → deva16_blue_center (Encode.skt_raw_to_deva "k.rdanta")
  | Roma → span2_center "Participles"
  ]
and indeclinables_caption = fun
  [ Deva → deva16_blue_center (Encode.skt_raw_to_deva "avyaya")
  | Roma → span2_center "Indeclinable␣forms"
  ]
and infinitive_caption = fun
  [ Deva → Encode.skt_to_deva "tumun"
  | Roma → "Infinitive"
  ]
and absolutive_caption is_root = fun
  [ Deva → Encode.skt_to_deva (if is_root then "ktvaa" else "lyap")
(* PB: absolutives in -aam should rather be labeled ".namul" *)
  | Roma → "Absolutive"
  ]
and peripft_caption = fun
  [ Deva → Encode.skt_to_deva "li.t"
  | Roma → "Periphrastic␣Perfect"
  ]
;
value voice_mark = fun
  [ Active → "para"
  | Middle → "aatma"
  | Passive → "karma.ni"
```

```
  ]
;
value participle_name part = fun
  [ Deva → let indian_part = match part with
      [ Ppp → [ "kta" ]
      | Pppa → [ "ktavatu" ]
      | Ppra _ → [ "zat.r" ]
      | Pprm _ → [ "zaanac" ]
      | Pprp → [ "zaanac"; "karma.ni" ]
      | Ppfta → [ "li.daadeza"; voice_mark Active ]
      | Ppftm → [ "li.daadeza"; voice_mark Middle ]
      | Pfuta → [ "lu.daadeza"; voice_mark Active ]
      | Pfutm → [ "lu.daadeza"; voice_mark Middle ]
      | Pfutp k → match k with
                  [ 1 → [ "yat" ]
                  | 2 → [ "aniiyar" ]
                  | 3 → [ "tavya" ]
                  | _ → []
                  ]
      | Action_noun → [ "krit" ] (* "gha~n" for -a "lyu.t" for -ana *)
      ] in
  let cat s x = s ^ "␣" ^ (Encode.skt_raw_to_deva x) in
  List.fold_left cat "" indian_part (* no skt punctuation so far *)
  | Roma → let western_part = match part with
      [ Ppp → "Past␣Passive␣Participle"
      | Pppa → "Past␣Active␣Participle"
      | Ppra _ → "Present␣Active␣Participle"
      | Pprm _ → "Present␣Middle␣Participle"
      | Pprp → "Present␣Passive␣Participle"
      | Ppfta → "Perfect␣Active␣Participle"
      | Ppftm → "Perfect␣Middle␣Participle"
      | Pfuta → "Future␣Active␣Participle"
      | Pfutm → "Future␣Middle␣Participle"
      | Pfutp _ → "Future␣Passive␣Participle"
      | Action_noun → "Action␣Noun"
      ] in western_part
  ]
;
value voice_name voice = fun
  [ Deva → let ivoice = match voice with
```

```
                [ Active → "parasmaipade"
                | Middle → "aatmanepade"
                | Passive → "karma.ni"
                ] in
            deva12_blue_center (Encode.skt_raw_to_deva ivoice)
  | Roma → let wvoice = match voice with
                [ Active → "Active"
                | Middle → "Middle"
                | Passive → "Passive"
                ] in
            span3_center wvoice
  ]
;
```

Interface for module Paraphrase

English paraphrase of semantic analysis

```
value print_sem : string → Morphology.inflexion_tag → unit;
value print_role : ((int × int × int) × Constraints.pos) → unit;
```

Module Paraphrase

English paraphrase of semantic analysis
A bit ad-hoc admittedly

```
open Skt_morph;
open Constraints; (* val_of_voice regime root_regime *)
open Html;
open Web; (* ps pl etc. *)
open Morphology; (* inflexions *)
```

```
value imperative_paraphrase pers num =
  match pers with
  [ First → match num with
        [ Singular → "Let␣me␣"
        | Dual → "Let␣us␣two␣"
        | Plural → "Let␣us␣"
        ]
  | Second → match num with
        [ Singular → "Thou␣"
```

```
          | Dual → "You␣two␣"
          | Plural → "You␣"
          ]
  | Third → match num with
          [ Singular → "Let␣it␣"
          | Dual → "Let␣them␣two␣"
          | Plural → "Let␣them␣"
          ]
  ]
;
value subject_paraphrase pers num =
  match pers with
  [ First → match num with
          [ Singular → "I␣"
          | Dual → "Both␣of␣us␣"
          | Plural → "We␣"
          ]
  | Second → match num with
          [ Singular → "Thou␣"
          | Dual → "Both␣of␣you␣"
          | Plural → "You␣"
          ]
  | Third → match num with
          [ Singular → "It␣"
          | Dual → "Both␣of␣them␣"
          | Plural → "All␣of␣them␣"
          ]
  ]
;
exception Unknown
;
value reg_stem = fun (* regular english verbs paraphrase *)
  [ "k.lp" → "effect"
  | "krii.d" → "play"
  | "tan#1" → "stretch"
  | "tap" → "suffer"
  | "tyaj" → "abandon"
  | "dhaav#2" → "clean"
  | "nind" → "blame"
  | "pac" → "cook"
```

```
  | "pa.th" → "learn"
  | "paa#2" → "protect"
  | "pii" → "increase"
  | "praz" → "ask"
  | "bhaa.s" → "talk"
  | "tarj"
  | "bharts" → "threaten"
  | "bruu" → "say"
  | "bhii#1" → "fear"
  | "ruc#1" → "please"
  | "labh" → "obtain"
  | "lal" → "fondle"
  | "v.rt#1" → "exist"
  | "v.r.s" → "rain"
  | "zubh#1" → "shine"
  | "sp.rz#1" → "touch"
  | "svid#2" → "sweat"
  | "ha#1" → "kill"
  | _ → raise Unknown
  ]
;
value paraphrase = fun (* returns pair (present stem, past participle) *)
  [ "at" | "i" | "gam" | "gaa#1" | "car" → ("go","gone") (* irregular verbs *)
  | "as#1" → ("i","")
  | "aas#2"
  | "viz#1" → ("sit","seated")
  | "kath" → ("tell","told")
  | "j~naa#1" → ("know","known")
  | "ta.d" → ("beat","beaten")
  | "daa#1" → ("give","given")
  | "dhaav#1" → ("run","chased")
  | "dh.r" → ("hold","held")
  | "nii#1" → ("lead","led")
  | "paz" → ("see","seen")
  | "paa#1" → ("drink","drunk")
  | "bhuj#2" → ("eat","eaten")
  | "bhuu#1" → ("become","become")
  | "m.r" → ("die","dead")
  | "likh" → ("write","written")
  | "vac" → ("speak","spoken")
```

```
  | "vah#1" → ("carry","carried")
  | "vid#1" → ("know","known")
  | "vid#2" → ("find","found")
  | "v.rdh#1" → ("grow","grown")
  | "vyadh"
  | "han" → ("hit","hurt")
  | "zru" → ("hear","heard")
  | "suu#1" → ("impel","impelled")
  | "sthaa#1" → ("stand","stood")
  | "svap" → ("sleep","asleep")
  | e → try let regular = reg_stem e in
              (regular, regular ^ "ed")
          with [ Unknown → ("do","done") (* default *) ]
  ]
;
value print_gender = fun
  [ Mas → ps "[M]"
  | Neu → ps "[N]"
  | Fem → ps "[F]"
  | Deictic d → match d with
      [ Speaker → ps "[Speaker]" (* First person *)
      | Listener → ps "[Listener]" (* Second person *)
      | Self → ps "[Self]" (* reflexive subject *)
      | Numeral → ps "[Num]" (* number *)
      ]
  ]
and print_number = fun
    [ Singular → () | Dual → ps "(2)" | Plural → ps "s" ]
and print_case = fun
    [ Nom → ps "Subject" (* Actor/Agent *)
    | Acc → ps "Object" (* Goal *)
    | Voc → ps "O" (* Invocation *)
    | Ins → ps "by" (* Agent/Instrument *)
    | Dat → ps "to" (* Destination *)
    | Abl → ps "from" (* Origin *)
    | Gen → ps "of" (* Possessor *)
    | Loc → ps "in" (* Circumstance *)
    ]
and print_person = fun
    [ First → ps "I"
```

```
      | Second → ps "You"
      | Third → ()
      ]
   ;
value genitive = fun
      [ Singular → "'s"
      | Dual → "␣pair's"
      | Plural → "s'"
      ]
;
value print_noun c n g =
   match c with
   [ Nom | Acc | Voc → do (* direct *)
         { print_case c
         ; print_number n
         ; sp ()
         ; print_gender g
         }
   | Gen → do
         { print_gender g
         ; ps (genitive n)
         }
   | _ → do (* oblique *)
         { print_case c
         ; sp ()
         ; print_gender g
         ; print_number n
         }
   ]
                ;
value third_sg act =
   if act = "do" ∨ act = "go" then "es" else "s"
;
value print_role = fun
   [ Subject _ _ → ps "Subject" (* Actor/Agent *)
   | Object → ps "Object" (* Goal/Patient *)
   | Instrument → ps "Agent" (* Agent/Instrument *)
   | _ → ()
   ]
;
```

```
value copula n = fun
  [ First → if n = Singular then "am" else "are"
  | Second → "are"
  | Third → if n = Singular then "is" else "are"
  ]
;
value print_verb w f n p =
  let (aspect, demand, _) = regime w f
  and (act, pas) = paraphrase w in
  match aspect with
  [ Imperfectif → do
      { ps (subject_paraphrase p n)
      ; if w = "as#1" then ps (copula n p)
        else do
            { if act = "carry" then ps "carrie" else ps act
            ; match p with
              [ First | Second → ()
              | Third → if n = Singular then ps (third_sg act) else ()
              ]
            }
      ; ps "␣"
      ; List.iter print_role demand
      }
  | Perfectif → do
      { ps (subject_paraphrase p n)
      ; ps (copula n p)
      ; ps "␣"
      ; ps pas
      }
  | Impersonal → do
      { ps act
      ; ps (third_sg act)
      }
  | Statif → do
    { ps "It␣"
    ; ps act
    ; ps (third_sg act)
    }
  ]
;
```

```
value print_abs entry =
  match root_regime entry with
  [ Intransitive | Factitive → ()
  | _ → ps "Object"
  ] (* conjugation c ignored at this stage *)
;
(* Translation Sanskrit -¿ English of tool words *)
value translate_tool = fun
  [ "ca" → "and"
  | "vaa" → "or"
  | "saha" → "with"
  | "iva" → "indeed"
  | "iti" → "even"
  | "eva" → "so"
  | "naaman" → "by␣name"
  | "yathaa" → "if"
  | "tathaa" → "then"
  | x → x (* keep stem *)
  ]
;
value print_verbal _ = ps "(Participial)␣" (* TODO *)
;
(* Adapted from Morpho.print_morph with extra string argument w for lexeme. Called
from Parser.print_roles. *)
value print_sem w = fun
  [ Noun_form g n c → print_noun c n g
  | Part_form v g n c → do { print_verbal v; print_noun c n g }
  | Verb_form f n p → print_verb w f n p
  | Abs_root _ | Ind_form Abs → print_abs w
  | Ind_form Adv → ps "Adverb"
  | Ind_form _ → ps (translate_tool w)
  | Bare_stem → ps "Compound"
  | Gati → ps "Composed"
  | _ → ()
  ]
;
value subj_of p n = match p with
  [ First → match n with
              [ Singular → "I" | Dual → "Us␣two" | Plural → "Us" ]
  | Second → match n with
```

```
              [ Singular → "Thou" | Dual → "You␣two" | Plural → "You" ]
  | Third → match n with
              [ Singular → "It" | Dual → "Both" | Plural → "They" ]
  ]
;
value print_noun_role = fun
  [ Subject p n → ps (subj_of p n)
  | Object → ps "Obj"
  | Instrument → ps "Agt"
  | Destination → ps "Dst"
  | Origin → ps "Org"
  | Possessor → ps "Pos"
  | Circumstance → ps "Cir"
  ]
;
value print_neg_noun_role k = do
  { ps "-"; print_noun_role k; ps "␣" }
;
value print_role ((seg, sub, ind), role) = do
  { ps (html_red (string_of_int seg ^ "." ^ string_of_int sub
                       ^ "." ^ string_of_int ind))
  ; ps (html_green "␣[")
  ; ps (span_begin Latin12)
  ; match role with
    [ Process noun_roles _
    | Subprocess noun_roles → List.iter print_neg_noun_role noun_roles
    | Actor noun_role gender number → do
        { print_noun_role noun_role
        ; match noun_role with [ Subject _ _ → () | _ → print_number number ]
        ; print_gender gender
        }
    | Tool Coordination → ps "␣&␣"
    | Tool _ → ps "␣T␣"
    | Number _ _ _ → ps "␣N␣"
    | Addressee → ps "␣V␣"
    | Compound → ps "␣C␣"
    | Ignored → ps "␣-␣"
    ]
  ; ps span_end
  ; ps (html_green "]␣")
```

```
  }
;
```

Module Bank_lexer

A simple lexer recognizing idents formed from ASCII letters and integers and skipping spaces and comments between Used by *Parse_tree* and *Reader*.

```
module Bank_lexer = struct

open Camlp4.PreCast;
open Format;

module Loc = Loc (* Using the PreCast Loc *)
;
module Error = struct
  type t = string
  ;
  exception E of t
  ;
  value to_string x = x
  ;
  value print = Format.pp_print_string
  ;
  end
;
module Token = struct
  module Loc = Loc
  ;
  type t =
    [ KEYWORD of string
    | IDENT of string
    | TEXT of string
    | INT of int
    | INTS of int
    | EOI
    ]
  ;
  module Error = Error
  ;
  module Filter = struct
```

```
    type token_filter = Camlp4.Sig.stream_filter t Loc.t
    ;
    type t = string → bool
    ;
    value mk is_kwd = is_kwd
    ;
    value rec filter is_kwd = parser
        [ [: ‘((KEYWORD s, loc) as p); strm :] → [: ‘p; filter is_kwd strm :]
(* PB if is_kwd s then [: ‘p; filter is_kwd strm :] else failwith ("Undefined␣token:␣" ^ s)
*)
        | [: ‘x; s :] → [: ‘x; filter is_kwd s :]
        | [: :] → [: :]
        ]
    ;
    value define_filter _ _ = ()
    ;
    value keyword_added _ _ _ = ()
    ;
    value keyword_removed _ _ = ()
    ;
    end
  ;
  value to_string = fun
    [ KEYWORD s → sprintf "KEYWORD␣%S" s
    | IDENT s → sprintf "IDENT␣%S" s
    | TEXT s → sprintf "TEXT␣%S" s
    | INT i → sprintf "INT␣%d" i
    | INTS i → sprintf "INTS␣%d" i
    | EOI → "EOI"
    ]
  ;
  value print ppf x = pp_print_string ppf (to_string x)
  ;
  value match_keyword kwd = fun
    [ KEYWORD kwd′ → kwd′ = kwd
    | _ → False
    ]
  ;
  value extract_string = fun
    [ INT i → string_of_int i
```

```
    | INTS i → string_of_int i
    | IDENT s | KEYWORD s | TEXT s → s
    | EOI → ""
    ]
  ;
end
;

open Token
;
```

The string buffering machinery - ddr + np

```
value store buf c = do { Buffer.add_char buf c; buf }
;
value rec base_number len =
  parser
  [ [: a = number len :] → a ]
and number buf =
  parser
  [ [: `('0'..'9' as c); s :] → number (store buf c) s
  | [: :] → Buffer.contents buf
  ]
  ;
value rec skip_to_eol =
  parser
  [ [: `'\n' | '\026' | '\012'; s :] → ()
  | [: `c ; s :] → skip_to_eol s
  ]
;
value ident_char =
  parser
  [ [: `('a'..'z' | 'A'..'Z' | '.' | ':' | '"' | '~' | '\'' as c) :]
     → c ]
;
value rec ident2 buff =
  parser
  [ [: c = ident_char; s :] → ident2 (store buff c) s
  | [: `('0'..'9' as c); s :] → ident2 (store buff c) s
  | [: :] → Buffer.contents buff
  ]
;
```

```
value rec text buff =
  parser
  [ [: ''}' :] → Buffer.contents buff
  | [: ''{'; buff = text_buff (store buff '{'); s :] →
                    text (store buff '}') s
  | [: 'c; s :] → text (store buff c) s
  ]
and text_buff buff =
  parser
  [ [: ''}' :] → buff
  | [: ''{'; buff = text_buff (store buff '{'); s :] →
                    text_buff (store buff '}') s
  | [: 'c; s :] → text_buff (store buff c) s
  ]
;
value next_token_fun =
  let rec next_token buff =
    parser _bp
    [ [: ''{'; t = text buff :] → TEXT t
    | [: '('1'..'9' as c); s = number (store buff c) :] → INT (int_of_string s)
    | [: ''0'; s = base_number (store buff '0') :] → INT (int_of_string s)
    | [: c = ident_char; s = ident2 (store buff c) :] →
      if s = "Comment" then KEYWORD "Comment" else
      if s = "Example" then KEYWORD "Example" else
      if s = "Continue" then KEYWORD "Continue" else
      if s = "Source" then KEYWORD "Source" else
      if s = "Parse" then KEYWORD "Parse" else
      if s = "Gloss" then KEYWORD "Gloss" else IDENT s
    | [: 'c :] _ep → KEYWORD (String.make 1 c)
    ] in
  let rec next_token_loc =
      parser bp
      [ [: ''%' ; _ = skip_to_eol; s :] → next_token_loc s
      | [: '' ' | '\n' | '\r' | '\t' | '\026' | '\012'; s :] → next_token_loc s
      | [: ''^' ; s :] → let (tok, loc) = next_token_loc s in
                         match tok with [ INT n → (INTS n, loc)
                                        | _ → raise (Token.Error.E "+n")
                                        ] (* for Gillon's dislocated phrases *)
      | [: ''!' ; s :] → let (tok, loc) = next_token_loc s in
                         match tok with [ INT n → (INTS (−n), loc)
```

```
                                          | _ → raise (Token.Error.E "-n")
                                          ] (* for Gillon's dislocation context *)
        | [: tok = next_token (Buffer.create 80) :] ep → (tok, (bp, ep))
        | [: _ = Stream.empty :] → (EOI, (bp, succ bp))
        ] in
  next_token_loc
  ;
value mk () =
  let err loc msg = Loc.raise loc (Token.Error.E msg) in
  fun init_loc cstrm → Stream.from lexer
  where lexer _ =
    try let (tok, (bp, ep)) = next_token_fun cstrm in
        let loc = Loc.move `start bp (Loc.move `stop ep init_loc) in
        Some (tok, loc)
    with [ Stream.Error str →
           let bp = Stream.count cstrm in
           let loc = Loc.move `start bp (Loc.move `stop (bp + 1) init_loc) in
           err loc str ]
;
end;
```

Module Checkpoints

Checkpoints management

```
open Phases.Phases; (* string_of_phase phase_of_string *)
```

string encoding of a phase, used to transmit checkpoints in URLs

```
value rec phase_encode = fun
  [ Comp (ph, ph') prev form →
        "<{" ^ string_of_phase ph ^ "}{" ^
               string_of_phase ph' ^ "}{" ^
               Canon.decode prev ^ "}{" ^ Canon.decode form ^ "}>"
  | phase → "{" ^ string_of_phase phase ^ "}"
  ]
and bool_encode b = if b then "t" else "f"
;
value string_point (k, (phase, rword), select) =
  let segment = Canon.rdecode rword in
  string_of_int k ^ "," ^ phase_encode phase ^ ",{" ^ segment ^
```

```
                    "},{" ^ bool_encode select ^ "}"
;
value rec string_points = fun
  [ [] → ""
  | [ last ] → string_point last
  | [ first :: rest ] → string_point first ^ "|" ^ string_points rest
  ]
;
open Bank_lexer;
module Gram = Camlp4.PreCast.MakeGram Bank_lexer
;
open Bank_lexer.Token
;
value cpts = Gram.Entry.mk "cpts"
and lcpt = Gram.Entry.mk "lcpt"
and phase_rword = Gram.Entry.mk "phase_rword"
and cpt = Gram.Entry.mk "cpt"
and phase = Gram.Entry.mk "phase"
and guess_morph = Gram.Entry.mk "guess_morph" (* for interface *)
;
EXTEND Gram
  cpts :
    [ [ l = lcpt; `EOI → l
      | lcpt → failwith "Wrong checkpoints parsing\n"
    ] ] ;
  lcpt :
    [ [ l = LIST0 cpt SEP "|" → l ] ] ;
  phase :
    [ [ "<"; p = TEXT; p' = TEXT (* Preverbed *)
          ; pre = TEXT; form = TEXT ; ">" →
        Comp (phase_of_string p, phase_of_string p')
                  (Encode.code_string pre) (Encode.code_string form)
      | p = TEXT → phase_of_string p
    ] ] ;
  phase_rword :
    [ [ s = phase; ","; o = TEXT → (s, Encode.rev_code_string o) ] ] ;
  cpt :
    [ [ m = INT; ","; p = phase_rword; ","; s = TEXT →
        (int_of_string m, p, s ="t") ] ] ;
  guess_morph :
```

```
    [ [ n = TEXT; ","; o = TEXT; `EOI → (n, o) ] ] ;
END
;
value parse_cpts s =
  try Gram.parse_string cpts Loc.ghost s with
  [ _ → raise (Control.Anomaly "parse_cpts") ]
;
value parse_guess s =
  try Gram.parse_string guess_morph Loc.ghost s with
  [ _ → raise (Control.Anomaly "parse_guess") ]
;
```

Parsing projections stream (Parser, Regression)

```
value projs = Gram.Entry.mk "projs"
and lproj = Gram.Entry.mk "lproj"
and proj = Gram.Entry.mk "proj"
;
(* A stream of projections is encoded under the form 1,2| 2,3|... *)
EXTEND Gram
  projs :
    [ [ l = lproj; `EOI → l
      | lproj → failwith "Wrong␣projections␣parsing\n"
    ] ] ;
  lproj :
    [ [ l = LIST0 proj SEP "|" → l ] ] ;
  proj :
    [ [ n = INT; ","; m = INT → (int_of_string n, int_of_string m) ] ] ;
END
;
value parse_proj s =
  try Gram.parse_string projs Loc.ghost s with
  [ _ → raise (Control.Anomaly "parse_proj") ]
;
```

Module Graph_segmenter

This segmenter is inspired from old module Segmenter, but uses a graph structure for the sharing of phased segments given with their offset.

```
open List2; (* unstack ass subtract *)
open Auto.Auto; (* auto rule choices State *)
```

used by Interface : *Viccheda* = *Segment Phases Machine Segment_control* where *Machine* = *Dispatch*
where *Lemmas* = *Load_morphs.Morphs Prel Phases*

```
module Segment
  (Phases : sig
        type phase
        and phases = list phase;
        value unknown : phase;
        value aa_phase : phase → phase;
        value preverb_phase : phase → bool;
        value ii_phase : phase → bool;
        value un_lopa : phase → phase;
        end)
  (Eilenberg : sig (* To be instanciated by Dispatcher *)
        value transducer : Phases.phase → auto;
        value initial : bool → Phases.phases;
        value dispatch : bool → Word.word → Phases.phase → Phases.phases;
        value accepting : Phases.phase → bool;
        type input = Word.word (* input sentence represented as a word *)
        and transition = (* junction relation *)
            [ Euphony of rule (* (w, rev u, v) such that u | v → w *)
            | Id (* identity or no sandhi *)
            ]
        and segment = (Phases.phase × Word.word × transition)
        and output = list segment;
        value validate : output → output; (* consistency check *)
        value sanitize_sa : bool → output → output;
        end)
  (Control : sig value star : ref bool; (* chunk= if star then word+ else word *)
                 value full : ref bool; (* all kridantas and nan cpds if full *)
            end)
  = struct

open Phases;
open Eilenberg;
open Control; (* star full *)
```

The summarizing structure sharing sub-solutions
It represents the union of all solutions
859 attested as last sentence in Pancatantra

```
value max_input_length = 1000
and max_seg_rows = 1000
```

```
;
exception Overflow (* length of sentence exceeding array size *)
;
(* segments of a given phase *)
type phased_segment = (phase × list (Word.word × list Word.word))
                       (* (segment, mandatory prefixes of following segment)
*)
and segments = list phased_segment (* partially forgetting sandhi *)
;
value null = ([] : segments) (* initialisation of graph entry *)
and null_visual = ([] : list (Word.word × list Word.word × phase × int))
                   (* (word, v's of next segment, phase, offset) *)
and null_visual_conf = ([] : list (Word.word × phase × int × bool))
                   (* (word, phase, offset, is_conflicting) *)
;
(* This is the graph on padas of the union of all solutions *)
(* We guarantee that every arc of the graph belongs to at least one bona fide segmentation.
But every path in this graph is not a valid segmentation. A path must pass global sandhi
verification to qualify as valid. *)
(* NB. Valid segmentations may contain unrecognized segments. *)
value graph = Array.make max_input_length null (* global over chunks *)
and visual = Array.make max_seg_rows null_visual
and visual_conf = Array.make max_seg_rows null_visual_conf
and visual_width = Array.make max_seg_rows 0
;
(* Checkpoints structure (sparse subgraph with mandatory positioned padas) *)
type phased_pada = (phase × Word.word) (* for checkpoints *)
and check = (int × phased_pada × bool) (* checkpoint validation *)
;
type checks =
  { all_checks : mutable (list check) (* checkpoints in valid solution *)
  ; segment_checks : mutable (list check) (* checkpoints in local segment *)
  }
;
value chkpts = { all_checks = []; segment_checks = []}
;
(* Accessing graph entry with phase *)
value split phase = split_rec []
  where rec split_rec acc = fun
  [ ([ ((ph, _) as fst) :: rst ] as l) →
```

```
    if ph = phase then (acc, l) else split_rec [ fst :: acc ] rst
  | [] → (acc, [])
  ]
;
value insert_right right pada = ins_rec
  where rec ins_rec acc = fun
  [ [] → failwith "insert_right"
  | [ (p, tr) :: rst ] →
        if p = pada then let tr' = [ right :: tr ] in
                             unstack [ (p, tr') :: acc ] rst
        else ins_rec [ (p, tr) :: acc ] rst
  ]
;
value get_pada pada = getrec where rec getrec = fun
  [ [] → None
  | [ (p, tr) :: rest ] → if p = pada then (Some tr) else getrec rest
  ]
;
value register_pada index (phase, pada, sandhi) =
  (* We search for bucket of given phase in graph *)
  let (al, ar) = split phase graph.(index)
  and allowed_right = match sandhi with
        [ Id → []
        | Euphony (w, _, v) → if w = v then [] else v
        ] in
  let pada_right = (pada, [ allowed_right ]) in
  let update_graph ar' = graph.(index) := unstack al ar' in
  match ar with
  [ [ (_, padas) :: rest ] → (* bucket found *)
    match get_pada pada padas with
    [ Some tr →
          if List.mem allowed_right tr then () (* already registered *)
          else let updated_sandhi = insert_right allowed_right pada [] padas in
                 update_graph [ (phase, updated_sandhi) :: rest ]
    | None → update_graph [ (phase, [ pada_right :: padas ]) :: rest ]
    ]
  | [] → update_graph [ (phase, [ pada_right ]) ] (* new bucket *)
  ]
;
```

To avoid heavy functional transmission of chunk global parameters, we define a record of chunk parameters. Attributes offset and *sa_control* are inherited, segmentable is synthesized. *sa_control* is True iff chunk is followed by chunk starting with consonant, and it then authorizes its last segment to be sa or e.sa pronouns

```
type chunk_params = { offset : mutable int
                    ; segmentable : mutable bool
                    ; sa_control : mutable bool (* for inter-chunk sa check *)
                    }
;
value cur_chunk = { offset = 0; segmentable = False ; sa_control = False }
;
value set_cur_offset n = cur_chunk.offset := n
and set_segmentable b = cur_chunk.segmentable := b
and set_sa_control b = cur_chunk.sa_control := b
;
value set_offset (offset, checkpoints) = do
  { set_cur_offset offset
  ; chkpts.all_checks := checkpoints
  }
;
value reset_graph () = for i = 0 to max_input_length − 1 do
  { graph.(i) := null }
;
value reset_visual () = for i = 0 to max_seg_rows − 1 do
  { visual.(i) := null_visual
  ; visual_conf.(i) := null_visual_conf
  ; visual_width.(i) := 0
  }
;
(* The offset permits to align each segment with the input string *)
value offset = fun
  [ Euphony (w, u, v) →
      let off = if w = [] then 1 (* amui/lopa from Lopa/Lopak *)
                else Word.length w in
      off − (Word.length u + Word.length v)
  | Id → 0
  ]
;
value rec contains phase_w = fun
  [ [] → False
  | [ (phase, word, _) :: rest ] → phase_w = (phase, word) ∨ contains phase_w rest
```

```
  ]
;
value check_chunk position solution checkpoints =
    check_rec position solution checkpoints
    where rec check_rec index sol checks = match checks with
      [ [] → True (* all checkpoints verified *)
      | [ (pos, phase_word, select) :: more ] →
          (* select=True for check *)
          if index > pos then
            if select then False
            else check_rec index sol more (* checkpoint missed *)
          else match sol with
          [ [] → True (* checkpoint relevant for later chunks *)
          | [ (phase, word, sandhi) :: rest ] →
              let next_index = index + Word.length word + offset sandhi in
              if index < pos then check_rec next_index rest checks
              else let (nxt_ind, ind_sols, next_sols) = all_sol_seg_ind [] sol
                  where rec all_sol_seg_ind stack = fun
                  [ [] → (next_index, stack, [])
                  | [ ((phase2, word2, sandhi2) as seg2) :: rest2 ] →
                    let next_index = pos + Word.length word2 + offset sandhi2 in
                    if next_index = pos then all_sol_seg_ind [ seg2 :: stack ] rest2
                    else (next_index, [seg2 :: stack], rest2)
                  ]
              and (ind_check, next_check) = all_check_ind [] checks
              where rec all_check_ind stack = fun
                [ [] → (stack, [])
                | ([ (pos2, phase_word2, select2) :: more2 ] as orig) →
                    if pos2 = pos then
                        all_check_ind [ (pos2, phase_word2, select2) :: stack ] more2
                    else (stack, orig)
                ] in
              check_sols ind_sols ind_check
              where rec check_sols solspt = fun
                [ [] → check_rec nxt_ind next_sols next_check
                | [ (pos2, phase_word2, select2) :: more2 ] →
                    (select2 = contains phase_word2 solspt)
                    (* Boolean select2 should be consistent with the solutions *)
                    ∧ check_sols solspt more2
                ]
```

```
            ]
        ]
;
(* counts the number of segmentation solutions of a chunk *)
value solutions_counter = ref 0
;
value bump_counter () = solutions_counter.val := solutions_counter.val + 1
and get_counter () = solutions_counter.val
and reset_counter () = solutions_counter.val := 0
;
value log_chunk revsol =
  let solution = List.rev revsol
  and position = cur_chunk.offset in
  if position ≥ max_input_length then raise Overflow else
  let check = check_chunk position solution chkpts.segment_checks in
    if check then (* log solution consistent with checkpoints *) do
        { log_rec position solution
          where rec log_rec index = fun
          [ [] → ()
          | [ ((phase, word, sandhi) as triple) :: rest ] → do
                { register_pada index triple
                ; log_rec (index + Word.length word + offset sandhi) rest
                }
          ]
        ; set_segmentable True
        ; bump_counter ()
        }
    else ()
;
```

Rest duplicated from Segmenter
Checking for legitimate Id sandhi
Uses *sandhis_id* computed by *Compile_sandhi*

```
value allowed_trans =
  (Gen.gobble Data.public_sandhis_id_file : Deco.deco Word.word)
;
value check_id_sandhi revl first =
  let match_right allowed = ¬ (List.mem [ first ] allowed) in
  try match revl with
      [ [] → True
```

```
    | [ last :: before ] →
        (Phonetics.n_or_f last ∧ Phonetics.vowel first) ∨
        (* we allow an-s transition with s vowel-initial, ignoring nn rules *)
        (* this is necessary not to block transitions from the An phase *)
        let allowed1 = Deco.assoc [ last ] allowed_trans in
        match before with
          [ [] → match_right allowed1
          | [ penu :: _ ] →
            let allowed2 = Deco.assoc [ last :: [ penu ] ] allowed_trans in
            match_right allowed2 ∧ match_right allowed1
          ]
    ]
  with [ Not_found → True ]
;
value sandhi_aa = fun
  [ [ 48; 1 ] → [ 1; 2 ] (* a.h | aa → a_aa *)
  | [ 43; 1 ] → Encode.code_string "araa" (* ar | aa → araa *)
  | [ c ] → match c with
              [ 1 | 2 → [ 2 ]
              | 3 | 4 → Encode.code_string "yaa"
              | 5 | 6 → Encode.code_string "vaa"
              | 7 | 8 | 48 → Encode.code_string "raa"
              | 9 → Encode.code_string "laa"
              | c → [ Phonetics.voiced c; 2 ]
              ]
  | _ → failwith "sandhi_aa"
  ]
;
(* Expands phantom-initial or lopa-initial segments *)
(* NB phase (aa_phase ph) of "aa" is Pv for verbal ph, Pvkv for nominal ones *)
value accrue ((ph, revword, rule) as segment) previous_segments =
  match Word.mirror revword with
    [ [ −2 (* - *) :: r ] → match previous_segments with (* First Lopa *)
      [ [ (phase, pv, Euphony ([], u, [−2])) :: rest ] → (* phase=Pv,Pvkv,Pvkc *)
        let v = match r with [ [ 10 (* e *) :: _ ] → [ 10 ]
                             | [ 12 (* o *) :: _ ] → [ 12 ]
                             | _ → failwith "accrue␣anomaly␣1"
                             ] in
        (* u is a or aa , v is e or o *)
        [ un_lopa_segment :: [ (phase, pv, Euphony (v, u, v)) :: rest ] ]
```

```
              where un_lopa_segment = (un_lopa ph, Word.mirror r, rule)
        | _ → failwith "accrue␣anomaly␣2"
        ]
         (* Then phantom phonemes *)
  | [ −3 (* *a *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [−3])) :: rest ] →
           let w = sandhi_aa u in
           [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 2 ], [ 2 ], [ 1 ]))
                             :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
             where new_segment = (ph, Word.mirror [ 1 :: r ], rule)
        | _ → failwith "accrue␣anomaly␣3"
        ]
  | [ −9 (* *A *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [−9])) :: rest ] →
           let w = sandhi_aa u in
           [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 2 ], [ 2 ], [ 2 ]))
                             :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
             where new_segment = (ph, Word.mirror [ 2 :: r ], rule)
        | _ → failwith "accrue␣anomaly␣4"
        ]
  | [ −4 (* *i *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [−4])) :: rest ] →
           let w = sandhi_aa u in
           [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 10 ], [ 2 ], [ 3 ]))
                             :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
             where new_segment = (ph, Word.mirror [ 3 :: r ], rule)
        | _ → failwith "accrue␣anomaly␣5"
        ]
  | [ −7 (* *I *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [−7])) :: rest ] →
           let w = sandhi_aa u in
           [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 10 ], [ 2 ], [ 4 ]))
                             :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
             where new_segment = (ph, Word.mirror [ 4 :: r ], rule)
        | _ → failwith "accrue␣anomaly␣6"
        ]
  | [ −5 (* *u *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [−5])) :: rest ] →
           let w = sandhi_aa u in
           [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 12 ], [ 2 ], [ 5 ]))
```

```
                        :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
              where new_segment = (ph, Word.mirror [ 5 :: r ], rule)
        | _ → failwith "accrue anomaly 7"
        ]
  | [ −8 (* *U *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [−8])) :: rest ] →
            let w = sandhi_aa u in
            [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 12 ], [ 2 ], [ 6 ]))
                        :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
              where new_segment = (ph, Word.mirror [ 6 :: r ], rule)
        | _ → failwith "accrue anomaly 8"
        ]
  | [ −6 (* *r *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [−6])) :: rest ] →
            let w = sandhi_aa u in
            [ new_segment :: [ (aa_phase ph, [ 2 ], Euphony ([ 2; 43 ], [ 2 ], [ 7 ]))
                        :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
              where new_segment = (ph, Word.mirror [ 7 :: r ], rule)
        | _ → failwith "accrue anomaly 9"
        ]
  | [ 123 (* *C *) :: r ] → match previous_segments with
        [ [ (phase, rword, Euphony (_, u, [ 123 ])) :: rest ] →
            if preverb_phase phase then failwith "accrue C with aa" else
            let w = sandhi_aa u in
            [ new_seg :: [ (aa_phase ph, [ 2 ], Euphony ([ 2; 22; 23 ], [ 2 ], [ 23 ]))
                    :: [ (phase, rword, Euphony (w, u, [ 2 ])) :: rest ] ] ]
              where new_seg = (ph, Word.mirror [ 23 :: r ], rule)
        | _ → failwith "accrue anomaly 10"
        ]
  | _ → [ segment :: previous_segments ]
  ]
;
type backtrack =
  [ Choose of phase and input and output and Word.word and choices
  | Advance of phase and input and output and Word.word
  ]
and resumption = list backtrack (* coroutine resumptions *)
;
```

Service routines of the segmenter

```
access : phase → word → option (auto × word)

value access phase = acc (transducer phase) []
  where rec acc state w = fun
    [ [] → Some (state, w) (* w is reverse of access input word *)
    | [ c :: rest ] → match state with
        [ State (_, deter, _) → match ass c deter with
            [ Some next_state → acc next_state [ c :: w ] rest
            | None → None
            ]
        ]
    ]
;
(* The scheduler gets its phase transitions from Dispatcher.dispatch *)
value schedule phase input output w cont =
  let add phase cont = [ Advance phase input output w :: cont ] in
  let transitions =
    if accepting phase ∧ ¬ star.val then [] (* Word = Sanskrit padas *)
    else dispatch full.val w phase (* iterate Word+ *) in
  List.fold_right add transitions cont
  (* respects dispatch order within a fair top-down search *)
;
(* The graph segmenter as a non deterministic reactive engine: phase is the parsing phase
input is the input tape represented as a word output is the current result of type output
back is the backtrack stack of type resumption occ is the current reverse access path in the
deterministic part the last anonymous argument is the current state of type auto. *)
(* Instead of functioning in coroutine with the Reader, one solution at a time, it computes
all solutions, populating the graph structure for later display *)
value rec react phase input output back occ = fun
  [ State (accept, det, choices) →
    (* we try the deterministic space before the non deterministic one *)
    let deter cont = match input with
      [ [] → continue cont
      | [ letter :: rest ] → match ass letter det with
          [ Some state → react phase rest output cont [ letter :: occ ] state
          | None → continue cont
          ]
      ] in
    let cont = if choices = [] then back (* non deterministic continuation *)
               else [ Choose phase input output occ choices :: back ] in
    (* now we look for - or + segmentation hint *)
```

```
    let (keep, cut, input') = match input with
      [ [ 0 :: rest ] → (* explicit "-" compound break hint *)
              (ii_phase phase, True, rest)
      | [ 100 :: rest ] → (* mandatory segmentation "+" *)
              (True, True, rest)
      | _ → (True, False, input) (* no hint in input *)
      ] in
    if accept ∧ keep then
      let segment = (phase, occ, Id) in
      let out = accrue segment output in
      match validate out (* validate and compact partial output *) with
      [ [] → if cut then continue cont else deter cont
      | contracted → match input' with
              [ [] → if accepting phase (* solution found *) then
                          register contracted cont
                     else continue cont
              | [ first :: _ ] → (* we first try the longest matching word *)
                     let cont' = schedule phase input' contracted [] cont in
                     if cut then continue cont' else
                     if check_id_sandhi occ first then (* legitimate Id *)
                          deter cont' else deter cont
              ]
      ]
    else if cut then continue cont else deter cont
  ]
and choose phase input output back occ = fun
  [ [] → continue back
  | [ ((w, u, v) as rule) :: others ] →
      let cont = if others = [] then back
                 else [ Choose phase input output occ others :: back ] in
      match subtract input w with (* try to read w on input *)
        [ Some rest →
          let segment = (phase, u @ occ, Euphony rule) in
          let out = accrue segment output in
          match validate out with
          [ [] → continue cont
          | contracted →
              if v = [] (* final sandhi *) then
                  if rest = [] ∧ accepting phase (* solution found *) then
                      register contracted cont
```

```
              else continue cont
            else continue (schedule phase rest contracted v cont)
          ]
        | None → continue cont
        ]
  ]
and continue = fun
  [ [] → () (* Exploration finished *)
  | [ resume :: back ] → match resume with
      [ Choose phase input output occ choices →
          choose phase input output back occ choices
      | Advance phase input output occ → match access phase occ with
          [ None → continue back
          | Some (state, v) → react phase input output back v state
          ]
      ]
  ]
(* CAUTION - This continue is completely different from the old continue from Segmenter.
It does not return one solution at a time in coroutine manner, but sweeps the whole solution
space. In particular, it returns () rather than an optional solution. *)
and register solution cont = (* Last check for sa/e.sa inter-chunk consistency *)
  match sanitize_sa cur_chunk.sa_control solution with
  [ [] → continue cont
  | chunk_sol → do { log_chunk chunk_sol; continue cont }
  ]
;
value init_segment_initial entries sentence =
  List.map (fun phase → Advance phase sentence [] []) entries
;
(* Works for Complete as well as Simplified mode *)
value segment1 chunk = continue (init_segment_initial (initial full.val) chunk)
;
value segment chunk = do
  { segment1 chunk (* does not assume Complete mode *)
  ; cur_chunk.segmentable ∨ do
    { graph.(cur_chunk.offset) := [ (unknown, [ (Word.mirror chunk, []) ]) ]
    ; False
    }
  }
;
```

```
(* Splitting checkpoints into current and future ones *)
value split_check limit = split_rec []
  where rec split_rec acc checkpts = match checkpts with
       [ [] → (Word.mirror acc, [])
       | [ ((index, _, _) as check) :: rest ] →
            if index > limit then (Word.mirror acc, checkpts)
            else split_rec [ check :: acc ] rest
       ]
;
(* We do not need to dove_tail like in Rank, since chunks are independent. *)
(* Returns a pair (b,n) where b is True if all chunks are segmentable so far, and n is the
number of potential solutions *)
value segment_chunk (full, count) chunk sa_check =
     let extremity = cur_chunk.offset + Word.length chunk in
     let (local, future) = split_check extremity chkpts.all_checks in do
     { chkpts.segment_checks := local
     ; set_sa_control sa_check (* inherited from chunks recursion *)
     ; let segmentable = segment chunk
        and local_count = get_counter () in do
        { set_segmentable False
        ; set_offset (succ extremity, future)
        ; if segmentable then do
               { reset_counter ()
               ; (full, count × local_count) (* overflow may compute modulo *)
                  (* we have local_count segmentations of the local chunk, and, chunks being
independent, the total number of solutions multiply *)
               }
          else (False, count) (* unsegmentable chunk *)
        }
     }
;
value segment_iter chunks = segment_chunks (True, 1) chunks
  where rec segment_chunks acc = fun (* speedy terminal recursion *)
     [ [ (* last *) chunk ] → segment_chunk acc chunk False
     | [ chunk :: rest ] → let sa_check = Phonetics.consonant_starts rest in
                                 segment_chunks (segment_chunk acc chunk sa_check) rest
     | [] → acc
     ]
;

end; (* Segment *)
```

Interface for module Interface

Sanskrit Reader Summarizing interface.
Similar design to Segmenter and Lexer, but records recognized segments represented in a shared graph with their offset with respect to the input sentence.

```
module Interface : sig
value safe_engine : unit → unit;
end;
```

Module Interface

Sanskrit Reader Summarizing interface. Yields sktgraph.cgi
We construct a CGI Interface displaying the segmentation graph in which the user may indicate segments as mandatory checkpoints. At any point he may call the standard displaying of all, or of preferred solutions consistent with the current checkpoints. An undo button allows backtracking.

```
module Interface = struct

open Graph_segmenter; (* Segment cur_chunk set_cur_offset graph visual *)
open Phases; (* Phases *)
open Phases; (* phase is_cache generative *)
open Dispatcher; (* transducer_vect Dispatch transition trim_tags *)
open Html; (* html constructors *)
open Web; (* ps pl abort reader_cgi scl_toggle etc. *)
open Cgi; (* url get decode_url *)

module Prel = struct (* Interface's lexer prelude *)

 value prelude () = do
  { pl http_header
  ; page_begin graph_meta_title
  ; pl (body_begin Chamois_back)
  ; pl interface_title
  ; pl (h3_begin C3 ^ "Click on " ^ html_green check_sign
                    ^ " to select segment, click on " ^ html_red x_sign
                    ^ " to rule out segment" ^ h3_end)
  ; pl (h3_begin C3 ^ mouse_action_help
                    ^ " on segment to get its lemma" ^ h3_end)
  ; open_page_with_margin 15
  }
;
```

```
 end (* Prel *)
;
(* Global parameters of the lexer *)
value iterate = ref True (* by default a chunk is a list of words *)
and complete = ref True (* by default we call the complete segmenter *)
and output_channel = ref stdout (* by default cgi output on standard output *)
;
(* Service routines for morphological query, loading the morphology banks *)
module Lemmas = Load_morphs.Morphs Prel Phases
;
open Lemmas (* tags_of morpho *)
;
open Load_transducers (* Trans mk_transducers dummy_transducer_vect *)
;
module Lexer_control = struct
 value star = iterate; (* vaakya vs pada *)
 value full = complete; (* complete vs simplified *)
 value out_chan = output_channel;
 value transducers_ref = ref (dummy_transducer_vect : transducer_vect);
 end (* Lexer_control *)
;
module Transducers = Trans Prel
;
module Machine = Dispatch Transducers Lemmas Lexer_control
;
open Machine
;
(* At this point we have a Finite Eilenberg machine ready to instantiate *)
(* the Eilenberg component of the Segment module. *)
```

Viccheda sandhi splitting

```
module Viccheda = Segment Phases Machine Lexer_control
;
open Viccheda (* segment_iter visual_width etc. *)
;
(* At this point we have the sandhi inverser segmenting engine *)
```

Separates tags of homophonous segments vertically

```
value fold_vert f = fold 1 where rec fold n = fun
  [ [] → ()
```

```
  | [ x ] → f n x
  | [ x :: l ] → do { f n x; ps html_break; fold (n + 1) l }
  ]
;
value print_morph pvs seg_num cached gen form n tag =
  Morpho_html.print_graph_link pvs cached form (seg_num, n) gen tag
;
(* tags : Morphology.multitag is the multi-tag of the form of a given phase *)
value print_tags pvs seg_num phase form tags =
  let gen = generative phase
  and cached = is_cache phase in
  let ok_tags = if pvs = [] then tags
                else trim_tags (generative phase) form (Canon.decode pvs) tags
  (* NB Existence of the segment warrants that ok_tags is not empty *)
  and ptag = print_morph pvs seg_num cached gen form in
  fold_vert ptag ok_tags
;
(* This is called "printing␣morphology␣interface␣style". *)
value print_morpho phase word =
    match tags_of phase word with
      [ Atomic tags → print_tags [] 0 phase word tags
      | Preverbed (_, phase) pvs form tags → print_tags pvs 0 phase form tags
      ]
;
```

Parsing mandatory checkpoints

```
open Checkpoints; (* string_points *)
```

```
value rpc = Paths.remote_server_host
and remote = ref False (* local invocation of cgi by default (switched on to True by "abs"
cgi parameter) *)
;
value invoke cgi = if remote.val then rpc ˆ cgi else cgi
;
value mem_cpts ind phase_pada = memrec where rec memrec = fun
  [ [] → False
  | [ (k, pw, _) :: rest ] → (k = ind ∧ pw = phase_pada) ∨ memrec rest
  ]
;
value unanalysed (phase, _) = (phase = Phases.unknown)
;
```

```
value already_checked = html_blue check_sign
;
value call_back text cpts (k, seg) conflict =
  if mem_cpts k seg cpts then already_checked
  else if ¬ conflict ∧ ¬ (unanalysed seg) then already_checked
  else let choices b = string_points [ (k, seg, b) :: cpts ]
       and (out_cgi, sign, color) =
           if unanalysed seg then (user_aid_cgi, spade_sign, Red_)
                             else (graph_cgi, check_sign, Green_) in
       let call_back flag = out_cgi ^ "?" ^ text ^ ";cpts=" ^ choices flag in
       let cgi_select = call_back True
       and cgi_reject = call_back False in
       anchor color (invoke cgi_select) sign ^
          if unanalysed seg then "" else anchor Red_ (invoke cgi_reject) x_sign
;
value call_reader text cpts mode = (* mode = "o", "p", "n" or "t" *)
  let cgi = reader_cgi ^ "?" ^ text ^ ";mode=" ^ mode ^
            ";cpts=" ^ string_points cpts in
  anchor Green_ (invoke cgi) check_sign
;
value call_parser text cpts =
  let cgi = parser_cgi ^ "?" ^ text ^ ";mode=p" ^
            ";cpts=" ^ string_points cpts ^ ";n=1" in
  anchor Green_ (invoke cgi) check_sign
;
value sort_check cpts =
  let compare_index (a, _, _) (b, _, _) = compare a b in
  List.sort compare_index cpts
;
value seg_length = fun
 [ [ -2 :: rest ] → Word.length rest (* lopa does not count *)
 | w → Word.length w
 ]
;
value rec merge_rec lpw = fun
  [ [] → lpw
  | [ (p, lw) :: rest ] → merge_rec (fill p lpw lw) rest
      where rec fill p lpw = fun
        [ [] → lpw
        | [ wh :: rest1 ] → fill p [ (p, wh) :: lpw ] rest1
```

```
         ]
  ]
;
value build_visual k segments =
  if segments = [] then () else
  let phw = merge_rec [] segments in
  let comp_length (_,(a,_)) (_,(b,_)) = compare (seg_length a) (seg_length b) in
  let sorted_seg = List.rev (List.sort comp_length phw) in
  ass_rec sorted_seg
    where rec ass_rec seg =
      let start_ind = find_ind_rec 0
          where rec find_ind_rec n =
          if k < visual_width.(n) then find_ind_rec (n + 1) else n in
      match seg with
      [ [] → ()
      | [ (phase,(w1,tr)) :: rest ] →
            if preverb_phase phase then failwith "Preverb␣in␣build_visual"
            else do
             { visual.(start_ind) := visual.(start_ind) @ [ (w1,tr,phase,k) ]
             ; visual_width.(start_ind) := (seg_length w1) + k
             ; ass_rec rest
             }
      ]
;
(* We check whether the current segment (w,tr,phase,k) is conflicting with others at previous offset n; if not it is mandatory and marked blue. *)
(* Returns True for blue mandatory segments, False for green/red optional ones *)
(* Warning: very hairy code, do not change without understanding the theory. *)
value is_conflicting ((w,tr,ph,k) as segment) =
 let l = seg_length w in is_conflicting_rec 0
 where rec is_conflicting_rec n = (* n is position in input string *)
 match visual.(n) with
 [ [] → False (* will exit here when n is length of input *)
 | segs → does_conflict segs (* we search for conflicting segments *)
     where rec does_conflict = fun
       [ [] → is_conflicting_rec (n + 1) (* go to next input position *)
       | [ ((w',tr',ph',k') as segment') :: rest ] →
           if segment' = segment then (* skip itself *) does_conflict rest
           else let l' = seg_length w' in
                if (k' ≤ k ∧ k' + l' − 1 > k) (* w inside w' *)
```

```
                    ∨ (k′ ≤ k ∧ k′ + l′ − 1 ≥ k ∧ l = 1) (* w is a or aa *)
                      (* This condition is necessary for the overlapping case *)
                    ∨ (k ≤ k′ ∧ k + l − 1 > k′) then
                      if k + l − 1 = k′ then let r′ = Word.mirror w′ in match_tr tr
```
(∗ This is to check for the overlapping case, occurs when $k = k'$, $l = 1$. We need to check the sandhi conditions to decide whether this is a case of overlap or conflict. ∗)
```
                           where rec match_tr = fun
                             [ [] → True
                             | [ v :: rst ] → match v with
                                 [ [] → match_tr rst
                                 | _ → if Word.prefix v r′
                                           then does_conflict rest
                                       else match_tr rst
                                 ]
                             ]
                      else if (k′ ≤ k ∧ k′ + l′ − 1 = k ∧ l = 1) then match_tr′ tr′
```
(∗ For the case with $l = 1$, this is to check whether w is the only possible v for w', in which case it is an overlap returning a blue sign. If w' has any other possible v's, there is a conflict. ∗)

(∗ This may only occur if w=1 (a) and w' ends in a or aa ∗)

(∗ NB. In naabhaava.h caakiirti.h a should be marked blue but in mahaajana.h after checking mahaa a should not be marked blue ∗)
```
                              where match_tr′ = fun
                                 [ [ v ] → ¬ (v = w) ∨ does_conflict rest
                                 | _ → True
                                 ]
                           else True
                      else does_conflict rest
          ]
  ]
;
value rec find_conflict_seg acc l = fun
  [ [] → List.rev acc
  | [ (w1, tr, phase, k) :: rest ] →
      let conflict = is_conflicting (w1, tr, phase, k) in
      let seg_here = (w1, phase, k, conflict) in
      find_conflict_seg [ seg_here :: acc ] l rest
  ]
;
value rec find_conflict l = match visual.(l) with
```

```
  [ [] → ()
  | segs → do
    { visual_conf.(l) := find_conflict_seg [] l segs
    ; find_conflict (succ l)
    }
  ]
;
value make_visual n = vrec 0
  where rec vrec k = do
    { build_visual k graph.(k)
    ; if k = n − 1 then () else vrec (succ k)
    }
;
value rec print_extra = fun
  [ 0 → ()
  | l → do { ps (td_wrap ""); print_extra (l − 1) }
  ]
and fixed_space = td_wrap " "
;
value rec print_first_server chunk =
  match Word.length chunk with
  [ 0 → ps fixed_space
  | l → match chunk with
        [ [] → ps fixed_space
        | [ st :: rest ] → let to_print = Canon.uniromcode [ st ] in do
            { ps (td_wrap to_print)
            ; print_first_server rest
            }
        ]
  ]
;
value call_back_pseudo text cpts ph newpt =
  if List.mem newpt cpts then already_checked
  else let list_points = [ newpt :: cpts ] in
       let out_cgi = user_aid_cgi in
       let cgi = out_cgi ^ "?" ^ text ^ ";cpts=" ^ (string_points list_points) in
       anchor_pseudo (invoke cgi) ph
;
value un_analyzable (chunk : Word.word) = (Phases.Unknown, Word.mirror chunk)
;
```

```
value rec print_first text cpts chunk_orig chunk chunk_ind =
  match Word.length chunk with
  [ 0 → ps fixed_space
  | l → match chunk with
        [ [] → ps fixed_space
        | [ st :: rest ] → let to_print = Canon.uniromcode [ st ] in do
            { let unknown_chunk = (chunk_ind, un_analyzable chunk_orig, True) in
              ps (td_wrap (call_back_pseudo text cpts to_print unknown_chunk))
            ; print_first text cpts chunk_orig rest chunk_ind
            }
        ]
  ]
 ;
(* Making use of the index for printing the chunk callback *)
value rec print_all text cpts chunks index = match chunks with
  [ [] → ()
  | [ chunk :: rest ] → do
      { print_first text cpts chunk chunk index
      ; print_all text cpts rest (succ (Word.length chunk))
      }
  ]
;
value print_word last_ind text cpts (rword, phase, k, conflict) =
  let word = Word.mirror rword in do
  { let extra_space = k − last_ind in
    if extra_space > 0 then print_extra extra_space else ()
  ; td_begin_att [ ("colspan",string_of_int (seg_length word))
                 ; ("align","left")
                 ] |> ps
  ; let back = background (color_of_phase phase) in
    table_begin back |> pl
  ; tr_begin |> ps
  ; "<td␣" ^ display_morph_action ^ "=\"showBox('" —> ps
  ; print_morpho phase word
  ; let close_box =
        "<a␣href=&quot;javascript:hideBox()&quot;>␣" ^ x_sign ^ "</a>',␣'" in
    close_box ^ rgb (color_of_phase phase) ^ "',␣this,␣event)\">" —> ps
  ; Morpho_html.print_final rword (* visarga correction *)
  ; td_end |> ps
  ; tr_end |> ps
```

```
  ; table_end |> ps
  ; call_back text cpts (k, (phase, rword)) conflict |> ps
  ; td_end |> ps
  }
;
value max_col = ref 0
;
value print_row text cpts = print_this text cpts 0
  where rec print_this text cpts last_ind = fun
  [ [] → let adjust = max_col.val − last_ind in
            if adjust > 0 then print_extra adjust else ()
  | [ (word, phase, k, conflict) :: rest ] → do
      { print_word last_ind text cpts (word, phase, k, conflict)
      ; print_this text cpts (k + seg_length word) rest
      }
  ]
;
value print_interf text cpts () = vgrec 0
  where rec vgrec k =
  match visual_width.(k) with
  [ 0 → ()
  | _ → do
    { tr_begin |> ps
    ; print_row text cpts visual_conf.(k)
    ; tr_end |> ps
    ; vgrec (succ k)
    }
  ]
;
value update_col_length chunk =
  max_col.val := succ (max_col.val + Word.length chunk)
and update_text_with_sol text count = text ^ ";allSol=" ^ string_of_int count
;
value call_undo text cpts =
  let string_pts = match cpts with
      [ [] → "" (* Could raise warning "undo␣stack␣empty" *)
      | [ _ :: rest ] → string_points rest
      ] in
  let cgi = graph_cgi ^ "?" ^ text ^ ";cpts=" ^ string_pts in
  anchor Green_ (invoke cgi) check_sign
```

```
;
(* The main procedure for computing the graph segmentation structure *)
value check_sentence translit us text_orig checkpoints sentence
                (* finally SL corpus links: *) sol_num corpus sent_id link_num =
  let encode = Encode.switch_code translit in
  let chunker = if us (* sandhi undone *) then Sanskrit.read_raw_sanskrit
                else (* blanks non-significant *) Sanskrit.read_sanskrit in
  let chunks = chunker encode sentence in
  let devachunks = List.map Canon.unidevcode chunks in
  let devainput = String.concat " " devachunks
  and cpts = sort_check checkpoints in
  let _ = chkpts.all_checks := cpts
  and (full, count) = segment_iter chunks in (* full iff all chunks segment *)
  let text = match sol_num with
              [ "0" -> update_text_with_sol text_orig count
              | _ -> text_orig
              ] in do
  { make_visual cur_chunk.offset
  ; find_conflict 0
  ; html_break |> pl
  ; html_latin16 "Sentence: " --> pl
  ; deva16_blue devainput |> ps (* devanagari *)
  ; html_break |> ps
  ; div_begin Latin16 |> ps
  ; table_begin Spacing20 |> pl
  ; tr_begin |> pl (* tr begin *)
  ; td_wrap (call_undo text checkpoints ^ "Undo") |> ps
  ; let call_scl_parser n = (* invocation of scl parser *)
        if scl_toggle then
           td_wrap (call_reader text cpts "o" ^ "UoH Analysis Mode") |> ps
        else () (* scl_parser is not visible unless toggle is set *) in
    if count > Web.max_count then
       (* too many solutions would choke the parsers *)
       td_wrap ("(" ^ string_of_int count ^ " Solutions)") |> ps
    else if count = 1 (* Unique remaining solution *) then do
             { td_wrap (call_parser text cpts ^ "Unique Solution") |> ps
             ; call_scl_parser 1
             }
         else do
       { td_wrap (call_reader text cpts "p" ^ "Filtered Solutions") |> ps
```

```
      ; let info = string_of_int count ^ if full then "" else "␣Partial" in
        td_wrap (call_reader text cpts "t" ^ "All␣" ^ info ^ "␣Solutions") |> ps
      ; call_scl_parser count
      }
  ; tr_end |> pl (* tr end *)
  ; table_end |> pl
  ; div_end |> ps (* Latin16 *)
  ; html_break |> pl
  ; div_begin Latin12 |> ps
  ; table_begin Tcenter |> pl
  ; tr_begin |> ps
  ; List.iter update_col_length chunks
  ; if Paths.platform ="Station" then print_all text checkpoints chunks 0
                                 else List.iter print_first_server chunks
  ; tr_end |> pl
  ; print_interf text checkpoints ()
  ; table_end |> pl
  ; div_end |> ps (* Latin12 *)
  ; html_break |> pl
  }
;
value arguments trans lex cache st us cp input topic abs sol_num corpus id ln
                corpus_permission corpus_dir sentence_no =
  "t=" ^ trans ^ ";lex=" ^ lex ^ ";cache=" ^ cache ^ ";st=" ^ st ^ ";us=" ^ us ^
  ";cp=" ^ cp ^ ";text=" ^ input ^ ";topic=" ^ topic ^ ";abs=" ^ abs ^
  match sol_num with
    [ "0" → ""
    | n → ";allSol=" ^ n
    ] ^
  match corpus with
    [ "" → ""
    | c → ";corpus=" ^ c ^ ";sentenceNumber=" ^ id ^ ";linkNumber=" ^ ln
    ] ^
  ";" ^ Params.corpus_permission ^ "=" ^ corpus_permission ^
  ";" ^ Params.corpus_dir ^ "=" ^ corpus_dir ^
  ";" ^ Params.sentence_no ^ "=" ^ sentence_no
;
```

Cache management

$(Morphology.inflected\_map \times Morphology.inflected\_map) \rightarrow unit$

```
value make_cache_transducers (cache, cachei) =
  let deco_cache = Mini.minimize (Deco.forget_deco cache)
  and deco_cachei = Mini.minimize (Deco.forget_deco cachei) in
  let auto_cache = Automaton.compile Deco.empty deco_cache
  and auto_cachei = Automaton.compile Deco.empty deco_cachei in do
  { Gen.dump cache Data.public_cache_file (* for Load_morphs *)
  ; Gen.dump cachei Data.public_cachei_file (* id *)
  ; Gen.dump auto_cache Data.public_trans_cache_file (* for Load_transducers *)
  ; Gen.dump auto_cachei Data.public_trans_cachei_file (* id *)
  }
;
(* We fill gendered entries incrementally in public_cache_txt_file *)
value append_cache entry gender =
  let cho = open_out_gen [ Open_wronly; Open_append; Open_text ] 0o777
                         Data.public_cache_txt_file in do
  { output_string cho ("[{" ^ entry ^ "}] ({" ^ gender ^ "})\n")
  ; close_out cho
  }
;
value save_button query nb_sols =
  center_begin ^
  cgi_begin save_corpus_cgi "" ^
  hidden_input Save_corpus_params.state (escape query) ^
  hidden_input Save_corpus_params.nb_sols (nb_sols |> string_of_int |> escape) ^
  submit_input "Save" ^
  cgi_end ^
  center_end
;
value quit_button corpmode corpdir sentno =
  let submit_button_label = Web_corpus.(
    match corpmode with
    [ Annotator -> "Abort"
    | Reader | Manager -> "Continue reading"
    ])
  and permission = Web_corpus.string_of_permission corpmode in
  center_begin ^
      cgi_begin (url corpus_manager_cgi ~fragment:sentno) "" ^
          hidden_input Params.corpus_dir corpdir ^
          hidden_input Params.corpus_permission permission ^
          submit_input submit_button_label ^
```

```
      cgi_end ^
  center_end
;
(* Main body of sktgraph cgi *)
value graph_engine () = do
  { Prel.prelude ()
  ; let query = Sys.getenv "QUERY_STRING" in
    let env = create_env query in
    (* Multiple environment variables according to modes of use are: text topic st cp us t lex cache abs cpts (standard mode) allSol (deprecated Validate mode) corpus sentenceNumber linkNumber (Corpus mode) corpdir sentno corpmode (defined in Params) guess gender revised rev_off rev_ind (User-aid) *)
    let url_encoded_input = get "text" env ""
    and url_encoded_topic = get "topic" env "" (* topic carry-over *)
    and st = get "st" env "t" (* sentence parse default *)
    and cp = get "cp" env "t" (* complete mode default *)
    and us = get "us" env "f" (* sandhied text default *)
    and translit = get "t" env Paths.default_transliteration (* translit input *)
    and lex = get "lex" env Paths.default_lexicon (* lexicon choice *)
    and cache = get "cache" env "f" (* no cache default *) in
    let () = cache_active.val := cache
    and abs = get "abs" env "f" (* default local paths *) in
    let lang = language_of lex (* language default *)
    and input = decode_url url_encoded_input (* unnormalized string *)
    and uns = us ="t" (* unsandhied vs sandhied corpus *)
    and () = if st ="f" then iterate.val := False else () (* word stemmer? *)
    and () = let full = (cp ="t") in do
           { Lexer_control.full.val := full
           ; Lexer_control.transducers_ref.val := Transducers.mk_transducers full
           }
    and () = toggle_lexicon lex (* sticky lexicon switch *)
    and corpus = get "corpus" env ""
    and sent_id = get "sentenceNumber" env "0"
    and link_num = get "linkNumber" env "0" (* is there a better default? *)
    and sol_num = get "allSol" env "0" in (* Needed for Validate mode *)
    let url_enc_corpus_permission = (* Corpus mode *)
        get Params.corpus_permission env "true" in
    let corpus_permission =
      url_enc_corpus_permission
      —> decode_url
```

```
      —> Web_corpus.permission_of_string in
  let corpus_dir = get Params.corpus_dir env "" in
  let sentence_no = get Params.sentence_no env "" in
  let text = arguments translit lex cache st us cp url_encoded_input
                       url_encoded_topic abs sol_num corpus sent_id link_num
                       url_enc_corpus_permission corpus_dir sentence_no
  and checkpoints =
    try let url_encoded_cpts = List.assoc "cpts" env in (* do not use get *)
        parse_cpts (decode_url url_encoded_cpts)
    with [ Not_found → [] ]
  and guess_morph = decode_url (get "guess" env "") (* User-aid guessing *)
  and pseudo_gender = decode_url (get "gender" env "") in
  let _ = if String.length guess_morph > 0 ∧ Paths.platform ="Station" then
              (* User-aid cache acquisition *)
              let (entry, gender) = match pseudo_gender with
                                    [ "" → parse_guess guess_morph
                                    | g → (guess_morph, g)
                                    ] in do
              { append_cache entry gender
              ; let cache_txt_file = Data.public_cache_txt_file in
                let caches = Nouns.extract_current_caches cache_txt_file in
                make_cache_transducers caches
              }
          else () in
  let revised = decode_url (get "revised" env "") (* User-aid revision *)
  and rev_off = int_of_string (get "rev_off" env "-1")
  and rev_ind = int_of_string (get "rev_ind" env "-1") in
 try do
 { match (revised, rev_off, rev_ind) with
   [ ("",-1,-1) → (* Standard input processing *** Main call *** *)
     check_sentence translit uns text checkpoints input sol_num
                    corpus sent_id link_num
   | (new_word, word_off, chunk_ind) (* User-aid revision mode *) →
     let chunks = Sanskrit.read_sanskrit (Encode.switch_code translit) input in
     let rec decoded init ind = fun
         [ [] → String.sub init 0 ((String.length init) − 1)
         | [ a :: rest ] →
             let ind′ = ind + 1
             and init′ = if ind = chunk_ind then init ^ new_word ^ "+"
                         else init ^ Canon.switch_decode translit a ^ "+" in
```

```
            decoded init' ind' rest
         ] in
      let updated_input = decoded "" 1 chunks in
      let rec find_word_len cur_ind = fun
         [ [] → 0
         | [ a :: rest ] → if cur_ind = chunk_ind then Word.length a
                            else find_word_len (cur_ind + 1) rest
         ] in
      let word_len = find_word_len 1 chunks
      and new_chunk_len = Word.length (Encode.switch_code translit revised) in
      let diff = new_chunk_len − word_len in
      let revised_check =
        let revise (k, sec, sel) = (if k < word_off then k else k + diff, sec, sel) in
        List.map revise checkpoints
      and updated_text = arguments translit lex cache st us cp updated_input
                           url_encoded_topic abs sol_num corpus sent_id link_num
                           url_enc_corpus_permission corpus_dir sentence_no
      and new_input = decode_url updated_input in
      check_sentence translit uns updated_text revised_check
                     new_input sol_num corpus sent_id link_num
    ]
    (* Rest of the code concerns Corpus mode *)
    (* automatically refreshing the page only if guess parameter *)
  ; if String.length guess_morph > 0 then
       ps ("<script>\nwindow.onload␣=␣function␣()␣{window.location=\"" ^
           graph_cgi ^ "?" ^ text ^
           ";cpts=" ^ (string_points checkpoints) ^ "\";}\n</script>")
    else ()
    (* Save sentence button *)
  ; if corpus_permission = Web_corpus.Annotator then
            save_button query 0 |> pl
    else ()
  ; html_break |> pl
    (* Quit button: continue reading (reader mode) or quit without saving (annotator
mode) *)
  ; if sentence_no ≠ "" then
       quit_button corpus_permission
                   (decode_url corpus_dir) (decode_url sentence_no) |> pl
    else ()
  ; close_page_with_margin ()
```

```
    ; page_end lang True
    }
    with
  [ Sys_error s → abort lang Control.sys_err_mess s (* file pb *)
  | Stream.Error s → abort lang Control.stream_err_mess s (* file pb *)
  | Encode.In_error s → abort lang "Wrong input " s
  | Exit (* Sanskrit *) → abort lang "Wrong character in input" ""
  | Overflow → abort lang "Maximum input size exceeded" ""
  | Invalid_argument s → abort lang Control.fatal_err_mess s (* sub array *)
  | Failure s → abort lang Control.fatal_err_mess s (* anomaly *)
  | End_of_file → abort lang Control.fatal_err_mess "EOF" (* EOF *)
  | Not_found → let s = "You must choose a parsing option" in
                              abort lang "Unset button in form - " s
  | Control.Fatal s → abort lang Control.fatal_err_mess s (* anomaly *)
  | Control.Anomaly s → abort lang Control.anomaly_err_mess s
  | _ → abort lang Control.fatal_err_mess "Unexpected anomaly"
  ]
  }
;
value safe_engine () =
   (* Problem: in case of error, we lose the current language of the session *)
   let abor = abort default_language in
   try graph_engine () with
   [ Failure s → abor Control.fatal_err_mess s (* parse_cpts phase_string ? *)
   | _ → abor Control.fatal_err_mess "Unexpected anomaly - broken session"
   ]
;
end (* Interface *)
;
Interface.safe_engine () (* Should always produce a compliant HTML page *)
;
```

Module User_aid

Sanskrit Reader summarizing interface. User aid with unrecognized segs.

```
open Html;
open Web; (* ps pl abort etc. remote_server_host *)
open Cgi;
open Phases;
```

```
open Checkpoints; (* phase_encode *)

module Prel = struct (* Interface's lexer prelude *)
 value prelude_user () = do
  { pl http_header
  ; page_begin user_aid_meta_title
  ; pl (body_begin Chamois_back)
  ; pl user_aid_title
  ; open_page_with_margin 15
  }
 ;
 end (* Prel *)
;
value rpc = remote_server_host
and remote = ref False (* local invocation of cgi by default *)
;
value string_point (offset, len_chunk) (k, (phase, rword), select) =
  let pada = Canon.rdecode rword in
  let updated_k = if k < offset then k else (k - len_chunk - 1) in
  string_of_int updated_k ^ "," ^ phase_encode phase ^ ",{" ^ pada ^ "},{"
                              ^ bool_encode select ^ "}"
;
value rec string_points off = fun (* off = (offset, len_chunk) *)
  [ [] -> ""
  | [ last ] -> string_point off last
  | [ first :: rest ] -> string_point off first ^ "|" ^ string_points off rest
  ]
;
value call_partial text (offset, len_chunk) cpts =
  let list_points = match cpts with
        [ [] -> []
        | [ _ :: rest ] -> rest
        ] in
  let cgi = graph_cgi ^ "?" ^ text ^ ";cpts=" ^
              (string_points (offset, len_chunk) list_points) in
  let invocation = if remote.val then rpc ^ cgi else cgi in
  anchor Green_ invocation check_sign
;
value string_point_orig (k, (phase, rword), select) =
  let pada = Canon.rdecode rword in
    string_of_int k ^ "," ^ phase_encode phase ^ ",{" ^ pada ^ "},{"
```

```
                    ^ bool_encode select ^ "}"
;
value rec string_points_orig = fun
  [ [] → ""
  | [ last ] → string_point_orig last
  | [ first :: rest ] → string_point_orig first ^ "|" ^ string_points_orig rest
  ]
;
value cpt_partial cpts =
  let list_points = match cpts with
      [ [] → []
      | [ _ :: rest ] → rest
      ] in
  string_points_orig list_points
;
(* Parsing mandatory checkpoints *)
open Checkpoints;

value sort_check cpts =
  let compare_index (a, _, _) (b, _, _) = compare a b in
  List.sort compare_index cpts
;
value rec find_chunk chunks ind = fun
  [ 0 → ind
  | l → match chunks with
        [ [ a :: rest ] → find_chunk rest (ind + 1) (l − ((List.length a) + 1))
        | _ → −1
        ]
  ]
;
value user_cgi_begin cgi =
  xml_begin_with_att "form"
    [ ("action",cgi); ("method","get") ] (* input conversion script *)
  ^ xml_begin "div"
;
value arguments trs lex cache st us cp input topic abs corpus sent_id link_num =
  let corpus_link = match corpus with
      [ "" → ""
      | _ → ";corpus=" ^ corpus ^ ";sentenceNumber=" ^ sent_id ^
            ";linkNumber=" ^ link_num
      ] in
```

```
    "t=" ^ trs ^ ";lex=" ^ lex ^ ";cache=" ^ cache ^ ";st=" ^ st ^
    ";us=" ^ us ^ ";cp=" ^ cp ^ ";text=" ^ input ^ ";topic=" ^ topic ^
    ";abs=" ^ abs ^ corpus_link
;
value print_hidden topic st cp us lex cache abs translit corpus sent_id
                        link_num = do
  { pl (hidden_input "topic" topic)
  ; pl (hidden_input "st" st)
  ; pl (hidden_input "cp" cp)
  ; pl (hidden_input "us" us)
  ; pl (hidden_input "t" translit)
  ; pl (hidden_input "lex" lex)
  ; pl (hidden_input "cache" cache)
  ; pl (hidden_input "abs" abs)
  ; match corpus with
    [ "" → ()
    | corpus_val → do
    { pl (hidden_input "corpus" corpus_val)
    ; pl (hidden_input "sentenceNumber" sent_id)
    ; pl (hidden_input "linkNumber" link_num)
    }
    ]
  }
;
value read_guess_index () =
  (Gen.gobble Data.public_guess_auto : Deco.deco (string × string))
;
value read_mw_index () =
  (Gen.gobble Data.public_mw_index_file : Deco.deco (string × string × string))
;

value rec mw_sol cur_sol word = fun
 [ [] → cur_sol
 | [ (entry, lex, page) :: rest ] → let updated_sol =
    match lex with
    [ "Noun" | "Ind." → cur_sol ^ Morpho_html.skt_anchor_M word entry page False
    | _ → cur_sol
    ] in mw_sol updated_sol word rest
 ]
;
```

function to find only the gender

```
value find_gen morph = String.sub morph (String.length morph − 2) 2
;
value print_word word (entry, morph) =
    let final_ent = word ^ entry in
    let mw_index = read_mw_index () in
    let words = List.rev (Deco.assoc (Encode.code_string final_ent) mw_index) in
    let header = td_begin ^ (table_begin Deep_sky_back) ^
                  tr_begin ^ th_begin
    and (sol, is_checked) = match words with
          [ [] → (" [" ^ Html.anchor_begin ^ Morpho_html.skt_roma final_ent ^
                        xml_end "a" ^ "]", False)
          | _ → let mw_solution = mw_sol "" final_ent words in
                  if (String.length mw_solution) > 0 then
                      (" [" ^ mw_solution ^ "]", True)
                  else (" [" ^ Html.anchor_begin ^ Morpho_html.skt_roma final_ent
                          ^ xml_end "a" ^ "]", False)
          ]
     and footer = th_end ^ tr_end ^ table_end ^ td_end in
     let radio = (Html.radio_input_dft "guess" ("{" ^ final_ent ^ "},{" ^
                      find_gen morph ^ "}") "" is_checked) ^ morph in
     header ^ radio ^ sol ^ footer
;
value rec string_word sol_st word = fun
  [ [] → sol_st
  | [ a :: rest ] → let new_sol = sol_st ^ (print_word word a) in
                      string_word new_sol word rest
  ]
;
```

We should replace the following function by a more standard primitive

```
value normalize_end = fun
  [ [ a :: rest ] →
     let normalized_a = match a with
                         [ 16 → 48 (* .h -¿ s *)
                         | 14 → 41 (* .m -¿ m *)
                         | c → c
                         ] in
     [ normalized_a :: rest ]
```

```
  | other → other
  ]
;
value aid_using translit checkpts sentence topic st cp us lex cache abs
                 corpus sent_id link_num =
  let encode = Encode.switch_code translit
  and decode = Canon.switch_decode translit in
  let chunks = Sanskrit.read_sanskrit encode sentence in
  let devachunks = List.map Canon.unidevcode chunks in
  let devainput = String.concat "␣" devachunks in do
  { pl html_break
  ; pl (html_latin16 "Sentence:␣")
  ; ps (deva16_blue devainput) (* devanagari *)
  ; pl html_break
  ; pl html_break
  ; pl center_begin
  ; pl (user_cgi_begin graph_cgi)
  ; print_hidden topic st cp us lex cache abs translit corpus sent_id link_num
  ; pl (xml_begin_with_att "textarea"
      [ ("name","text"); ("rows","1"); ("cols","100") ] ^
      sentence ^ xml_end "textarea")
  ; pl html_break
  ; pl (submit_input "Submit␣Revised␣Sentence")
  ; pl cgi_end
  ; pl html_break
  ; pl html_break
  ; pl (user_cgi_begin graph_cgi)
  ; print_hidden topic st cp us lex cache abs translit corpus sent_id link_num
  ; pl (hidden_input "text" sentence)
  ; let (offset, chunk_rev) = match checkpts with
        [ [ (k,(_,word),_) :: _ ] → (k, Word.mirror word)
        | _ → (0,[])
        ] in
    let len_chunk = Word.length chunk_rev
    and chunk_ind = find_chunk chunks 1 offset in do
  { pl (xml_begin_with_att "textarea"
        [ ("name","revised"); ("rows","1"); ("cols","30") ] ^
        (decode chunk_rev) ^ xml_end "textarea")
  ; pl (hidden_input "rev_off" (string_of_int offset))
  ; pl (hidden_input "rev_ind" (string_of_int chunk_ind))
```

```
; pl (hidden_input "cpts" (cpt_partial checkpts))
; pl html_break
; pl (submit_input "Submit␣Revised␣Chunk")
; pl cgi_end
; pl html_break
; pl html_break
; if List.length chunks > 1 then do
    { ps (div_begin Latin12)
    ; pl (table_begin Spacing20)
    ; pl tr_begin
    ; let rec decoded init cur_ind = fun
            [ [] → String.sub init 0 ((String.length init) − 1)
            | [ a :: rest ] →
                if cur_ind = chunk_ind then decoded init (cur_ind + 1) rest
                else decoded (init ^ (decode a) ^ "+") (cur_ind + 1) rest
            ] in
      let updated_text = decoded "" 1 chunks in
      let arg_string = arguments translit lex cache st us cp updated_text topic
                                         abs corpus sent_id link_num in
      ps (td_wrap (call_partial arg_string (offset, len_chunk) checkpts
                  ^ "Show␣partial␣solution␣without␣this␣chunk"))
    ; pl tr_end
    ; pl table_end
    ; ps div_end (* Latin12 *)
    }
  else ()
(* adding new module to show the possibilities *)
; if Paths.platform = "Station" then do
{ ps html_paragraph
; ps (div_begin Latin16)
; ps "Possible␣lemmatizations␣for␣the␣chunk:"
; ps div_end
; pl par_end
; ps (div_begin Latin12)
; pl (user_cgi_begin graph_cgi)
; print_hidden topic st cp us lex cache abs translit corpus sent_id link_num
; pl (hidden_input "cpts" (cpt_partial checkpts))
; pl (hidden_input "text" sentence)
; pl html_break
; let guess_auto = read_guess_index () in
```

```
let rec match_decl sol_string init = (* init is the last *) fun
[ [] → sol_string
| [ a :: rest ] →
  let updated_init = [ a :: init ] in
  let words = List.rev (Deco.assoc (Word.mirror updated_init) guess_auto) in
  let new_sol = match words with
      [ [] → sol_string
      | _ → let str_rest = Canon.decode (Word.mirror rest) in
            let lemma = string_word "" str_rest words in
            let this_string = tr_begin ^ lemma ^ tr_end in
            [ this_string :: sol_string ]
      ] in
  match_decl new_sol updated_init rest
] in do
{ pl (table_begin_style Blue_ [ noborder; ("align","center"); spacing20 ])
; List.iter pl (match_decl [] [] (normalize_end (List.rev chunk_rev)))
  (* [] = sol_string, [] = last *)
; pl table_end
}
; pl html_break
; pl (submit_input "Submit␣Morphology")
; pl cgi_end (* graph_cgi *)
; ps div_end (* Latin12 *)
; ps (par_begin Latin16)
; ps "Enter␣your␣own␣lemmatization:"
; pl par_end
; pl (user_cgi_begin graph_cgi)
; print_hidden topic st cp us lex cache abs translit corpus sent_id link_num
; pl (hidden_input "cpts" (cpt_partial checkpts))
; pl (hidden_input "text" sentence)
; pl html_break
; ps (text_area "guess" 1 20 "")
; pl html_break
; ps (option_select [ ("nom.","Nominative"); ("acc.","Accusative");
                      ("ins.","Instrumental"); ("dat.","Dative");
                      ("abl.","Ablative"); ("gen.","Genitive");
                      ("loc.","Locative"); ("voc.","Vocative") ])
; ps (option_select_label "gender"
        [ ("m.","Masculine"); ("f.","Feminine"); ("n.","Neuter") ])
; ps (option_select [ ("sg.","Singular"); ("du.","Dual"); ("pl.","Plural") ])
```

```
  ; pl html_break
  ; pl (submit_input "Submit␣Choices")
  ; pl cgi_end
  } (* Paths.platform = "Station" *) else ()
  ; pl center_end
  ; pl html_break
  }
}
;
value user_aid_engine () = do
  { Prel.prelude_user ()
  ; let query = Sys.getenv "QUERY_STRING" in
    let env = create_env query in
    let url_encoded_input = get "text" env ""
    and url_encoded_topic = get "topic" env ""
    and st = get "st" env "t"
    and cp = get "cp" env "t"
    and us = get "us" env "f"
    and translit = get "t" env "SL" (* default SLP1 *)
    and lex = get "lex" env "SH" (* default Heritage *)
    and cache = get "cache" env "f" in
    let () = cache_active.val := cache in
    let corpus = get "corpus" env ""
    and sent_id = get "sentenceNumber" env "0"
    and link_num = get "linkNumber" env "0"
    and abs = get "abs" env "f" (* default local paths *) in
    let lang = language_of lex
    and input = decode_url url_encoded_input (* unnormalized string *) in
    let checkpts =
        try let url_encoded_cpts = List.assoc "cpts" env in (* do not use get *)
             parse_cpts (decode_url url_encoded_cpts)
        with [ Not_found -> [] ] in
    try do
     { aid_using translit checkpts input url_encoded_topic st cp us lex cache
                 abs corpus sent_id link_num
     ; close_page_with_margin ()
     ; page_end lang True
     }
    with
    [ Sys_error s -> abort lang Control.sys_err_mess s (* file pb *)
```

```
  | Stream.Error s → abort lang Control.stream_err_mess s (* file pb *)
  | Encode.In_error s → abort lang "Wrong␣input␣" s
  | Exit (* Sanskrit *) → abort lang "Wrong␣character␣in␣input" ""
  | Invalid_argument s → abort lang Control.fatal_err_mess s (* sub *)
  | Failure s → abort lang Control.fatal_err_mess s (* anomaly *)
  | End_of_file → abort lang Control.fatal_err_mess "EOF" (* EOF *)
  | Not_found → let s = "You␣must␣choose␣a␣parsing␣option" in
                        abort lang "Unset␣button␣in␣form␣-␣" s
  | Control.Fatal s → abort lang Control.fatal_err_mess s (* anomaly *)
  | Control.Anomaly s → abort lang Control.fatal_err_mess ("Anomaly:␣" ^ s)
  | _ → abort lang Control.fatal_err_mess "Unknown␣anomaly"
  ]
}
;
value safe_engine () =
  let abor = abort default_language in
  try user_aid_engine () with
  [ _ → abor Control.fatal_err_mess "Unexpected␣anomaly␣-␣broken␣session" ]
;
safe_engine () (* Should always produce a valid xhtml page *)
;
```

Module Reset_caches

Reset_caches
Used for initializing or resetting the cache databases
Caution. Execution of this program erases the contents of the caches

```
open Morphology;
open Auto;

value empty_inflected_map = (Deco.empty : inflected_map) (* dummy morpho bank *)
and empty_trans = Auto.State(False, [], []) (* dummy empty transducer *)
;

Gen.dump empty_inflected_map Data.public_cache_file
;
Gen.dump empty_inflected_map Data.public_cachei_file
;
Gen.dump empty_trans Data.public_trans_cache_file
;
```

```
Gen.dump empty_trans Data.public_trans_cachei_file
;
Unix.system (":>" ^ Data.public_cache_txt_file) (* resets the master text cache *)
;
```

Module Restore_caches

Restore caches
Restore the cache databases from master cache file
Cache management (from Interface)
(*Morphology.inflected_map* × *Morphology.inflected_map*) → *unit*

```
value make_cache_transducers (cache, cachei) =
  let deco_cache = Mini.minimize (Deco.forget_deco cache)
  and deco_cachei = Mini.minimize (Deco.forget_deco cachei) in
  let auto_cache = Automaton.compile Deco.empty deco_cache
  and auto_cachei = Automaton.compile Deco.empty deco_cachei in do
  { Gen.dump cache Data.public_cache_file (* for Load_morphs *)
  ; Gen.dump cachei Data.public_cachei_file (* id *)
  ; Gen.dump auto_cache Data.public_trans_cache_file (* for Load_transducers *)
  ; Gen.dump auto_cachei Data.public_trans_cachei_file (* id *)
  }
;
value restore_caches () =
let cache_txt_file = Data.public_cache_txt_file in
let caches = Nouns.extract_current_caches cache_txt_file in
make_cache_transducers caches
;
restore_caches ()
;
```

Interface for module Params

Common parameters of different CGIs related to the reader
Parameter for specifying the corpus subdirectory when the corpus mode is enabled.

```
value corpus_dir : string
;
(* Parameter for specifying the sentence number when the corpus mode is enabled. *)
value sentence_no : string
```

```
;
(* Parameter for specifying the permission of the corpus user: "reader", "annotator" or
"manager". *)
value corpus_permission : string
;
```

Module Params

```
value corpus_dir = "corpdir"
;
value sentence_no = "sentno"
;
value corpus_permission = "corpmode"
;
```

Module Html

Pidgin ML as scripting langage of the poor for HTML and XML
Generic HTML scripting
All values are pure, with no side-effect, no printing.
Attributes given as association lists $(label, value) : (string \times string)$

```
value assoc_quote (label, valu) =
  let sp_label = "␣" ^ label in
  sp_label ^ "=\"" ^ valu ^ "\""
;
value rec quote_alist = fun
  [ [] → ""
  | [ assoc_list ] → assoc_quote assoc_list
  | [ assoc_list :: rest ] → (assoc_quote assoc_list ^ quote_alist rest)
  ]
;
(* Elementary XML constructors *)
value xml_begin xml_op = "<" ^ xml_op ^ ">"
and xml_begin_with_att xml_op atts = "<" ^ xml_op ^ (quote_alist atts) ^ ">"
and xml_end xml_op = "</" ^ xml_op ^ ">"
and xml_empty xml_op = "<" ^ xml_op ^ ">"
and xml_empty_with_att xml_op atts = "<" ^ xml_op ^ (quote_alist atts) ^ ">"
;
value xml_next op = xml_end op ^ xml_begin op
```

```
;
value html_break = xml_empty "br"
and html_paragraph = xml_begin "p" ^ xml_end "p"
;
(* Array operations *)
value tr_begin = xml_begin "tr"
and tr_end = xml_end "tr"
and th_begin = xml_begin "th"
and th_end = xml_end "th"
and td_begin = xml_begin "td"
and td_end = xml_end "td"
and td = xml_empty_with_att "td"
;
value td_wrap text = td_begin ^ text ^ td_end
and cell item = tr_begin ^ th_begin ^ item ^ th_end ^ tr_end
;
(* Dynamic colors depending on mouse position *)
value tr_mouse_begin color_over color_out =
  xml_begin_with_att "tr"
          (* beware case of attributes below; colors must be quoted *)
          [ ("onMouseover","this.bgColor=" ^ color_over)
          ; ("onMouseout" ,"this.bgColor=" ^ color_out)
          ]
;
value input_id = "focus"
and focus_script = (* selection of input window *)
    "(function(){var src = document.getElementById('focus');src.select();})();"
;
value text_area control width length sentence =
  let w = string_of_int width
  and l = string_of_int length in
  xml_begin_with_att "textarea"
      [ ("id",input_id); ("name",control); ("rows",w); ("cols",l) ] ^ sentence ^
  xml_end "textarea" (* Caution - necessary to separate begin and end *) ^ "\n" ^
  xml_begin_with_att "script" [ ("type","text/javascript") ] ^ focus_script ^
  xml_end "script"
;
(* printing options for the user to choose lemma *)
value option_print id control =
  xml_begin_with_att "option" [ ("value",id) ] ^ control ^ xml_end "option"
```

```
;
value option_print_default id control b =
  let value_param = ("value",id) in
  let params = if b then [ value_param; ("selected","selected") ]
                    else [ value_param ] in
  let menu = xml_begin_with_att "option" params in
  menu ^ control ^ xml_end "option"
;
value rec print_options = fun
   [ [] → ""
   | [ (id, control) :: rest ] → option_print id control ^ print_options rest
   ]
;
value rec print_options_default = fun
   [ [] → ""
   | [ (control, id, b) :: rest ] →
       (option_print_default id control b) ^ (print_options_default rest)
   ]
;
value option_select list_options =
  xml_begin "select" ^ print_options list_options ^ xml_end "select"
;
value option_select_label label list_options = xml_begin_with_att "select"
  [ ("name",label) ] ^ print_options list_options ^ xml_end "select"
;
value option_select_default label list_options = xml_begin_with_att "select"
  [ ("name",label) ] ^ print_options_default list_options ^ xml_end "select"
;
value option_select_default_id id label list_options =
  (xml_begin_with_att "select" [ ("id",id); ("name",label) ]) ^
  print_options_default list_options ^ xml_end "select"
;
value text_input id control =
  xml_empty_with_att "input" [ ("id",id); ("type","text"); ("name",control) ]
;
value add_opt_attrs opt_attrs attrs =
  let add_attr acc (label, v) =
    match v with
    [ None → acc
    | Some v → [ (label, v) :: acc ]
```

```
    ] in
  List.fold_left add_attr attrs opt_attrs
;
value int_input ?id ?val ?(step = 1) ?(min = min_int) ?(max = max_int) name =
  let attrs =
    [ ("type", "number")
    ; ("name", name)
    ; ("step", string_of_int step)
    ; ("min", string_of_int min)
    ; ("max", string_of_int max)
    ] in
  let opt_attrs =
    [ ("id", id); ("value", Gen.opt_app string_of_int val) ] in
  let attrs = add_opt_attrs opt_attrs attrs in
  xml_empty_with_att "input" attrs
;
value radio_input control v label =
  let attrs = [ ("type","radio"); ("name",control); ("value",v) ] in
  (xml_empty_with_att "input" attrs) ^ label
;
value select control =
  List.map (fun (label, v) → radio_input control v label)
;
value radio_input_dft control v label checked =
  let check = if checked then [ ("checked","checked") ] else [] in
  let attrs = [ ("type","radio"); ("name",control); ("value",v) ] @ check in
  (xml_empty_with_att "input" attrs) ^ label
;
value select_default name =
  List.map (fun (label, v, checked) → radio_input_dft name v label checked)
;
value submit_input label =
  xml_empty_with_att "input" [ ("type","submit"); ("value",label) ]
;
value reset_input label =
  xml_empty_with_att "input" [ ("type","reset"); ("value",label) ]
;
value hidden_input name label =
  xml_empty_with_att "input" [ ("type","hidden"); ("name",name); ("value",label) ]
;
```

Lists
List item

```
value li ?id item =
  let li = "li" in
  let attrs = add_opt_attrs [ ("id", id) ] [] in
  xml_begin_with_att li attrs ^ item ^ xml_end li
;
(* Ordered list *)
value ol ?id ?li_id_prefix ?(start = 1) items =
  let ol = "ol" in
  let process i item =
    let id = let genid prefix = prefix ^ string_of_int (start + i) in
             Gen.opt_app genid li_id_prefix in
    li ?id item in
  let lines = List.mapi process items in
  let list = String.concat "\n" lines in
  let attrs = add_opt_attrs [ ("id", id) ] [ ("start", string_of_int start) ] in
  xml_begin_with_att ol attrs ^ "\n" ^ list ^ "\n" ^ xml_end ol
;
value fieldn name content = [ ("name",name); ("content",content) ]
and fieldp name content = [ ("property",name); ("content",content) ]
;
type position = [ Left | Center | Right ]
and font_family = list string
and font_style = [ Normal | Italic | Slanted ]
;
type color =
  [ Black | White | Red | Blue | Green | Yellow | Orange | Deep_sky | Purple
  | Grey | Navy | Cyan | Brown | Carmin | Chamois | Broon | Maroon |
 Aquamarine
  | Gold | Magenta | Mauve | Pink | Salmon | Lime | Light_blue | Lavender
  | Lawngreen | Deep_pink | Pale_yellow | Pale_rose | Beige ]
;
type basic_style =
  [ Font_family of font_family
  | Font_style of font_style
  | Font_size of int
  | Textalign of position
  | Tablecenter
  | Color of color
```

```
    | Bgcolor of color
    | Position of string
    | Full_width
    | Height of int
    | No_margin
    | Border of int
    | Padding of int
    | Cellpadding of int
    | No_border
    | Border_sep
    | Border_col
    | Border_sp of int
    | Hidden
    ] (* font-weight not supported *)
;
value rgb = fun (* a few selected HTML colors in rgb data *)
  [ Black → "#000000"
  | White → "#FFFFFF" (* Wheat = "#F0E0B0" ou "#F5DEB3" *)
  | Red → "#FF0000" (* Firebrick = "#B02020" *)
  | Blue → "#0000FF" (* Canard = "#0000C0" ou "#0080FF" *)
  | Green → "#008000" (* Teal = "#008080" Olive = "#808000" *)
  | Aquamarine → "#6FFFC3" (* actually Light Aquamarine *)
  | Lawngreen → "#7CFC00"
  | Yellow → "#FFFF00"
  | Orange → "#FFA000"
  | Cyan → "#00FFFF" (* Aqua = Cyan, Turquoise = "#40E0D0" *)
  | Purple → "#800080" (* Plum = "#E0A0E0" *)
  | Grey → "#B8B8B8" (* Slategrey = "#708090" *)
  | Navy → "#000080" (* Midnight blue = "#101870" *)
  | Deep_sky → "#00C0FF"
  | Brown → "#A02820" (* Chocolate = "#D06820" *)
  | Maroon → "#800000"
  | Carmin → "#FF1975" (* Carmin = "#FF0066" Deep pink= "#FF1090" *)
  | Chamois → "#F5F5DC" (* gris-beige *)
  | Broon → "#852B1D" (* good with gold *)
  | Gold → "#A58959" (* Silver = "#C0C0C0" *)
  | Magenta → "#FF00FF" (* Violet = "#F080F0" Blueviolet = "#8028E0" *)
  | Mauve → "#FF99FF" (* Orchid = "#D070D0" *)
  | Pink → "#FFC0C0" (* Hotpink = "#FF68B0" Thisle = "#D0C0D0" *)
  | Deep_pink → "#FF1493"
```

```
  | Salmon → "#F08070"
  | Beige → "#FFCCA0"
  | Lime → "#00FF00" (* Chartreuse = "#80FF00" *)
  | Pale_yellow → "#FFFF66"
  | Pale_rose → "#FFDDDD" (* Mistyrose = "#FFE4E1" *)
  | Light_blue → "#ADD8E6"
  | Lavender → "#E6E6FA" (* or "#E0E8F0" *)
  ]
;
(* quoted color needed for arguments of tr_mouse_begin exclusively *)
value color c = "'" ^ rgb c ^ "'"
;
(* Special symbols *)
value check_sign = "&#x2713;"
and spade_sign = "&#x2660;"
and heart_sign = "&#x2661;"
and x_sign = "&#10008;"
;
(* Fonts used for the Web site. *)
(* "Times␣IndUni" is deprecated, now called "IndUni-T" (John Smith's fonts) *)
value roman_fonts = [ "IndUni-T"; "Arial␣Unicode␣MS" ] (* "Times_CSX" *)
and greek_fonts = [ "Arial␣Unicode␣MS"; "Symbol" ] (* "Latin␣Extended-B" Greek *)
and diacr_fonts = [ "IndUni-T"; "Arial␣Unicode␣MS" ]
    (* Sanskrit transliteration in romanised script with diacritics *)
and deva_fonts = [ "Arial␣Unicode␣MS" ] (* Devanagari fonts *)
(* NB: "Devanagari␣MT" deprecated because wrong rendering of tacchrutvaa *)
;
value roman_font = Font_family roman_fonts
and greek_font = Font_family greek_fonts
and trans_font = Font_family diacr_fonts
and deva_font = Font_family deva_fonts
;
value points n = string_of_int n ^ "pt"
and pixels n = string_of_int n ^ "px"
and percent n = string_of_int n ^ "%"
;
value font_style = fun
  [ Normal → "normal"
  | Italic → "italic"
  | Slanted → "oblique" (* Not well-supported by browsers *)
```

```
  ]
;
value justify = fun
  [ Left → "left"
  | Center → "center"
  | Right → "right"
  ]
;
(* Style sheet generator *)
value style_sheet = fun
  [ Font_family fonts → "font-family:␣" ^ family
      where family = String.concat "," fonts
  | Font_style fs → "font-style:␣" ^ font_style fs
  | Font_size sz → "font-size:" ^ points sz
  | Textalign p → "text-align:" ^ justify p
  | Color cl → "color:" ^ rgb cl
  | Bgcolor cl → "background-color:" ^ rgb cl
  | Position pos → pos
  | Tablecenter → "margin:0␣auto"
  | No_border → "border:␣0"
  | Border n → "border:␣outset␣" ^ points n
  | Padding n → "padding:" ^ points n
  | Cellpadding p → "cellpadding:" ^ percent p
  | Full_width → "width:100%"
  | Height h → "height:" ^ points h
  | No_margin → "margin-left:␣0pt;␣margin-right:␣0pt;␣margin-top:␣0pt"
  | Border_sep → "border-collapse:separate"
  | Border_col → "border-collapse:collapse"
  | Border_sp n → "border-spacing:" ^ points n
  | Hidden → "display:␣none"
  ]
;
(* Style of enpied bandeau with fixed position at bottom of page - fragile *)
value enpied = "position:␣fixed;␣bottom:␣0pt;␣width:␣100%"
;
(* All the styles of the various sections - terminology to be streamlined *)
(* NB: When style_class is changed, module Css ought to be adapted *)
type style_class =
    [ Blue_ | Green_ | Navy_ | Red_ | Magenta_ | Hidden_
    | Header_deva | Header_tran | Bandeau | Body | Spacing20 | Pad60 | Border2
```

```
    | Latin12 | Trans12 | Deva | Devac | Deva16 | Deva16c | Deva20c
    | Roma16o | Roma12o | Inflexion
    | Alphabet | G2 | Title | Latin16 | Trans16 | Devared_ | Math | Enpied
    | B1 | B2 | B3 | C1 | C2 | C3 | Cell5 | Cell10 | Center_ | Tcenter |
 Centered
    | Gold_cent | Mauve_cent | Yellow_cent | Cyan_cent | Deep_sky_cent
    | Yellow_back | Blue_back | Salmon_back | Light_blue_back | Gold_back
    | Pink_back | Chamois_back | Cyan_back | Brown_back | Lime_back | Grey_back
    | Deep_sky_back | Carmin_back | Orange_back | Red_back | Mauve_back
    | Lavender_back | Lavender_cent | Green_back | Lawngreen_back | Magenta_back
    | Aquamarine_back
    ]
;
value background = fun
    [ Mauve → Mauve_back
    | Magenta → Magenta_back
    | Pink → Pink_back
    | Chamois → Chamois_back
    | Yellow → Yellow_back
    | Salmon → Salmon_back
    | Cyan → Cyan_back
    | Gold → Gold_back
    | Brown → Brown_back
    | Lime → Lime_back
    | Blue → Blue_back
    | Light_blue → Light_blue_back
    | Deep_sky → Deep_sky_back
    | Carmin → Carmin_back
    | Orange → Orange_back
    | Red → Red_back
    | Lavender → Lavender_back
    | Green → Green_back
    | Lawngreen → Lawngreen_back
    | Aquamarine → Aquamarine_back
    | Grey → Grey_back
    | _ → failwith "Unknown␣background␣style"
    ]
and centered = fun
    [ Mauve → Mauve_cent
    | Yellow → Yellow_cent
```

```
    | Gold → Gold_cent
    | Deep_sky → Deep_sky_cent
    | Cyan → Cyan_cent
    | Lavender → Lavender_cent
    | _ → failwith "Unknown␣centered␣style"
    ]
  ;
(* Table of styles of each style class *)
value styles = fun
    [ Centered → [ Tablecenter ]
    | Mauve_cent → [ Bgcolor Mauve; Tablecenter; Border 8; Padding 10 ]
    | Yellow_cent → [ Bgcolor Yellow; Tablecenter; Border 5; Padding 10 ]
    | Lavender_cent → [ Bgcolor Lavender; Tablecenter; Border 5; Padding 10 ]
    | Inflexion → [ Bgcolor Yellow; Tablecenter; Border 2; Padding 5 ]
    | Deep_sky_cent → [ Bgcolor Deep_sky; Tablecenter; Border 5; Padding 10 ]
    | Gold_cent → [ Bgcolor Gold; Tablecenter; Border 0; Padding 10 ]
    | Cyan_cent → [ Bgcolor Cyan; Tablecenter; Border 5; Padding 10 ]
    | Mauve_back → [ Bgcolor Mauve ]
    | Magenta_back → [ Bgcolor Magenta ]
    | Aquamarine_back → [ Bgcolor Aquamarine ]
    | Pink_back → [ Bgcolor Pale_rose; No_margin ] (* Pink *)
    | Yellow_back → [ Bgcolor Yellow ]
    | Salmon_back → [ Bgcolor Salmon ]
    | Chamois_back → [ Bgcolor Chamois; No_margin; Full_width ]
    | Cyan_back → [ Bgcolor Cyan ]
    | Gold_back → [ Bgcolor Gold ]
    | Brown_back → [ Bgcolor Brown; No_margin; Padding 10; Full_width ]
    | Lime_back → [ Bgcolor Lime ]
    | Deep_sky_back → [ Bgcolor Deep_sky ]
    | Carmin_back → [ Bgcolor Carmin ]
    | Orange_back → [ Bgcolor Orange ]
    | Red_back → [ Bgcolor Red ]
    | Grey_back → [ Bgcolor Grey ]
    | Blue_back → [ Bgcolor Blue ]
    | Lawngreen_back → [ Bgcolor Lawngreen ]
    | Green_back → [ Bgcolor Green ]
    | Light_blue_back → [ Bgcolor Light_blue ]
    | Lavender_back → [ Bgcolor Lavender ]
    | Blue_ → [ trans_font; Color Blue ]
    | Green_ → [ trans_font; Color Green ]
```

```
| Navy_ → [ trans_font; Color Navy ]
| Red_ → [ trans_font; Color Red ]
| Roma16o → [ trans_font; Color Red; Font_size 16; Font_style Slanted ]
| Devared_ → [ deva_font; Color Red ]
| Magenta_ → [ trans_font; Color Magenta ]
| Header_deva → [ deva_font; Color Red; Font_size 24; Textalign Left ]
| Header_tran → [ trans_font; Color Red; Font_size 24; Textalign Left ]
| Deva → [ deva_font; Color Maroon; Font_size 12 ]
| Devac → [ deva_font; Color Blue; Font_size 12; Textalign Center ]
| Deva16 → [ deva_font; Color Blue; Font_size 16 ]
| Deva16c → [ deva_font; Color Blue; Font_size 16; Textalign Center ]
| Deva20c → [ deva_font; Color Blue; Font_size 20; Textalign Center ]
| Alphabet → [ trans_font; Font_size 24; Textalign Center ]
| Title → [ roman_font; Color Blue; Font_size 24; Textalign Center ]
| Trans12 → [ trans_font; Font_size 12 ]
| B1 → [ roman_font; Color Blue; Font_size 20 ]
| B2 → [ roman_font; Color Blue; Font_size 16 ]
| B3 → [ roman_font; Color Blue; Font_size 12 ]
| C1 → [ roman_font; Color Blue; Font_size 20; Textalign Center ]
| C2 → [ roman_font; Color Blue; Font_size 16; Textalign Center ]
| C3 → [ roman_font; Color Blue; Font_size 12; Textalign Center ]
| G2 → [ roman_font; Color Green; Font_size 16 ]
| Center_ → [ Textalign Center ]
| Pad60 → [ Textalign Center; Height 60; Full_width ]
| Tcenter → [ Tablecenter ]
| Roma12o → [ trans_font; Color Black; Font_size 12; Font_style Slanted ]
| Latin12 → [ roman_font; Color Black; Font_size 12 ]
| Latin16 → [ roman_font; Color Black; Font_size 16 ]
| Trans16 → [ trans_font; Color Black; Font_size 16 ]
| Math → [ greek_font; Color Black; Font_size 12 ]
| Enpied → [ Position enpied ]
| Bandeau → [ roman_font; Bgcolor Cyan; Border_sep; Border_sp 10
                    ; Full_width ]
| Body → [ roman_font; Bgcolor Pale_rose; Border_sep; Border_sp 10
                    ; Full_width ]
| Spacing20 → [ Border_sep; Border_sp 20 ]
| Cell5 → [ Padding 5 ]
| Cell10 → [ Padding 10 ]
| Border2 → [ Border 2 ]
| Hidden_ → [ Hidden ]
```

```
    ]
;
(* Compiles a class into its style for non-css compliant browsers *)
(* Nowadays mostly used by Css to compile the css style sheet *)
value style cla = String.concat "; " (List.map style_sheet (styles cla))
;
value class_of = fun
    [ Mauve_cent → "mauve_cent"
    | Yellow_cent → "yellow_cent"
    | Inflexion → "inflexion"
    | Deep_sky_cent → "deep_sky_cent"
    | Centered → "centered"
    | Cyan_cent → "cyan_cent"
    | Mauve_back → "mauve_back"
    | Magenta_back → "magenta_back"
    | Pink_back → "pink_back"
    | Gold_cent → "gold_cent"
    | Yellow_back → "yellow_back"
    | Blue_back → "blue_back"
    | Light_blue_back → "light_blue_back"
    | Salmon_back → "salmon_back"
    | Chamois_back → "chamois_back"
    | Cyan_back → "cyan_back"
    | Gold_back → "gold_back"
    | Lavender_back → "lavender_back"
    | Lavender_cent → "lavender_cent"
    | Brown_back → "brown_back"
    | Lime_back → "lime_back"
    | Deep_sky_back → "deep_sky_back"
    | Carmin_back → "carmin_back"
    | Orange_back → "orange_back"
    | Red_back → "red_back"
    | Green_back → "green_back"
    | Lawngreen_back → "lawngreen_back"
    | Aquamarine_back → "aquamarine_back"
    | Grey_back → "grey_back"
    | Blue_ → "blue"
    | Green_ → "green"
    | Navy_ → "navy"
    | Red_ → "red"
```

```
  | Roma16o → "red16"
  | Roma12o → "roma12o"
  | Magenta_ → "magenta"
  | Header_deva → "header_deva"
  | Header_tran → "header_tran"
  | Latin12 → "latin12"
  | Deva → "deva"
  | Devared_ → "devared"
  | Devac → "devac"
  | Deva16 → "deva16"
  | Deva16c → "deva16c"
  | Deva20c → "deva20c"
  | Alphabet → "alphabet"
  | Title → "title"
  | Trans12 → "trans12"
  | B1 → "b1"
  | B2 → "b2"
  | B3 → "b3"
  | C1 → "c1"
  | C2 → "c2"
  | C3 → "c3"
  | G2 → "g2"
  | Center_ → "center"
  | Tcenter → "center"
  | Spacing20 → "spacing20"
  | Latin16 → "latin16"
  | Trans16 → "trans16"
  | Math → "math"
  | Enpied → "enpied"
  | Bandeau → "bandeau"
  | Pad60 → "pad60"
  | Cell5 → "cell5"
  | Cell10 → "cell10"
  | Border2 → "border2"
  | Body → "body"
  | Hidden_ → "hidden"
  ]
;
(* Allows css style compiling even when browser does not support css *)
(* This support was necessary for Simputer platform - now deprecated *)
```

```
value elt_begin_attrs attrs elt cl =
  let style_attr = (* if Install.css then *) ("class",class_of cl)
                                  (* else ("style",style cl) *) in
  xml_begin_with_att elt [ style_attr :: attrs ]
;
value elt_begin = elt_begin_attrs [ ]
;
value par_begin = elt_begin "p"
and h1_begin = elt_begin "h1"
and h2_begin = elt_begin "h2"
and h3_begin = elt_begin "h3"
and span_begin = elt_begin "span"
and span_skt_begin = elt_begin_attrs [ ("lang","sa") ] "span" (* EXP *)
and div_begin = elt_begin "div"
and body_begin = elt_begin "body"
and body_begin_style = elt_begin_attrs margins "body" (* Body margins are null *)
  where margins = [ ("style","margin-left:␣0;␣margin-right:␣0;␣margin-top:␣0;") ]
(* Caution: table_begin_style is not compliant with HTML5 (dynamic style) *)
and table_begin_style style attrs = elt_begin_attrs attrs "table" style
and table_begin = elt_begin "table"
and td_begin_class = elt_begin "td"
and th_begin_class = elt_begin "th"
and td_begin_att = xml_begin_with_att "td" (* depr *)
;
value par_end = xml_end "p"
and h1_end = xml_end "h1"
and h2_end = xml_end "h2"
and h3_end = xml_end "h3"
and span_end = xml_end "span"
and div_end = xml_end "div"
and body_end = xml_end "body"
and table_end = xml_end "table"
;
(* table parameters *)
value noborder = ("border","0")
and nopadding = ("cellpadding","0%")
and padding5 = ("cellpadding","5%")
and padding10 = ("cellpadding","10%")
and nospacing = ("cellspacing","0")
and spacing5 = ("cellspacing","5pt")
```

```
and spacing20 = ("cellspacing","20pt")
and fullwidth = ("width","100%")
;
value span style text = span_begin style ^ text ^ span_end
and span_skt style text = span_skt_begin style ^ text ^ span_end
and div style text = div_begin style ^ text ^ div_end
;
(* Centering old style - deprecated *)
value center = div Center_
and center_begin = div_begin Center_
and center_end = div_end
;
value html_red = span Red_
and html_devared = span_skt Devared_
and html_magenta = span Magenta_
and html_blue = span Blue_
and html_green = span Green_
and html_math = span Math
and html_trans12 = span Trans12
and html_trans16 = span Trans16
and html_latin12 = span Latin12
and html_latin16 = span Latin16
and roma16_red_sl = span Roma16o
and roma12_sl = span Roma12o
and span2_center = span B2
and span3_center = span B3
and deva12_blue_center = span_skt Devac
and deva16_blue = span_skt Deva16
and deva16_blue_center = span_skt Deva16c
and deva20_blue_center = span_skt Deva20c
;
value title s = xml_begin "title" ^ s ^ xml_end "title"
and h1_title s = h1_begin Title ^ s ^ h1_end
and h1_center s = h1_begin B1 ^ s ^ h1_end
;
value italics s = xml_begin "i" ^ s ^ xml_end "i"
and emph s = xml_begin "b" ^ s ^ xml_end "b"
;
value hr = xml_empty "hr"
;
```

```
value anchor_ref url link =
  (xml_begin_with_att "a" [ ("href",url) ]) ^ link ^ (xml_end "a")
;
value anchor cl url link =
  (elt_begin_attrs [ ("href",url) ] "a" cl) ^ link ^ (xml_end "a")
;
value anchor_def label link =
  (xml_begin_with_att "a" [ ("name",label) ]) ^ link ^ (xml_end "a")
;
value anchor_define cl label link =
  (elt_begin_attrs [ ("name",label) ] "a" cl) ^ link ^ (xml_end "a")
;
value anchor_graph cl url link =
  "<a␣href=&quot;" ^ url ^ "&quot;>" ^ link ^ "</a>"
; (* NB: use ∧quot; and not quote sign for Javascript *)
value anchor_begin = xml_begin_with_att "a" [ ("class", "navy") ]
;
value anchor_pseudo url link =
    (xml_begin_with_att "a" [ ("href",url); ("style","text-decoration:␣none") ])
  ^ link
  ^ (xml_end "a")
;
```

Specific HTML scripting

```
value start_year = "␣1994-"
and current_year = "2020"
and author_name = "Gérard␣Huet"
;
value copyright = "©␣" ^ author_name ^ start_year ^ current_year
;
value author = fieldn "author" author_name
and date_copyrighted = fieldp "dc:datecopyrighted" current_year
and rights_holder = fieldp "dc:rightsholder" author_name
and keywords = fieldn "keywords"
    "sanskrit,dictionary,heritage,dictionnaire,sanscrit,india,inde,indology,linguisti
;
value heritage_dictionary_title = title "Sanskrit␣Heritage␣Dictionary"
;
(* Supported publishing media *)
type medium = [ Html | Tex ]
```

```
;
(* Supported HTTP platforms *)
type platform = [ Simputer | Computer | Station | Server ]
;
(* Current target platform to customize - needs recompiling if changed *)
value target = match Paths.platform with
  [ "Simputer" → Simputer (* Historical - small screen *)
  | "Smartphone" | "Tablet" → Simputer (* TODO *)
  | "Computer" → Computer (* Standard client installation *)
  | "Station" → Station (* Permits external Analysis mode *)
  | "Server" → Server (* Http server for Internet web services *)
  | _ → failwith "Unknown␣target␣platform"
  ]
;
(* Features of target architecture *)
value (narrow_screen, screen_char_length, css) =
 match target with
  [ Simputer → (True, 40, False) (* Historical for Simputer platform *)
  | Station (* Privileged client mode *)
  | Computer
  | Server → (False, 80, True) (* Server mode *)
  ]
;
(* Internationalisation *)
type language = [ French | English ]
;
(* Two indexing lexicons are supported, French SH and English MW.*)
value lexicon_of = fun
  [ French → "SH" (* Sanskrit Heritage *)
  | English → "MW" (* Monier-Williams *)
  ]
and language_of = fun
  [ "SH" → French
  | "MW" → English
  | _ → failwith "Unknown␣lexicon"
  ]
;
value default_language = language_of Paths.default_lexicon
and default_mode = (* TODO - add as config parameter *)
 match target with
```

```
  [ Station | Computer | Server → "t" (* default Complete mode *)
  | _ → "f" (* default Simplified mode *)
  ]
;
(* linked lexical resource - initialized at configuration *)
value lexicon_toggle = ref Paths.default_lexicon (* mutable for lexicon access *)
;
value toggle_lexicon lex = lexicon_toggle.val := lex
;
value page_extension lang =
  let lang_sfx = fun
      [ French → "fr"
      | English → "en"
      ] in
  "." ^ lang_sfx lang ^ ".html"
;
value wrap_ext page lang = page ^ page_extension lang
;
value site_entry_page = wrap_ext "index"
and dico_index_page = wrap_ext "index"
and dico_reader_page = wrap_ext "reader"
and dico_grammar_page = wrap_ext "grammar"
and dico_sandhi_page = wrap_ext "sandhi"
and dico_corpus_page = wrap_ext "corpus"
and faq_page = wrap_ext "faq"
and portal_page = wrap_ext "portal"
;
(* URLs relative to DICO for static pages *)
value rel_dico_path = "../"
;
value images_top_path = "IMAGES/"
;
value rel_sanskrit_page_url l = rel_dico_path ^ (site_entry_page l)
and rel_faq_page_url l = rel_dico_path ^ (faq_page l)
and rel_portal_page_url l = rel_dico_path ^ (portal_page l)
and rel_web_images_url = rel_dico_path ^ images_top_path
;
value rel_image name = rel_web_images_url ^ name
(* rel image is relative in order to pre-compile DICO in distribution site *)
;
```

```
value rel_ocaml_logo = rel_image "icon_ocaml.png"
and rel_inria_logo = rel_image "logo_inria.png"
and left_blue_arr = rel_image "arrw01_16a.gif"
and right_blue_arr = rel_image "arrw01_06a.gif"
and rel_favicon = rel_image "favicon.ico"
;
value meta_prefix = xml_empty_with_att "meta"
;
value contents_instructions =
  [ [ ("charset","utf-8") ] ]
;
value title_instructions =
  [ author; date_copyrighted; rights_holder; keywords ]
;
value doctype = "<!DOCTYPE␣html>" (* Assuming HTML5 *)
;
value url dns = "http://" ^ dns
;
value ocaml_site = url "ocaml.org"
and inria_site = url "www.inria.fr/"
and tomcat = url "localhost:8080/" (* Sanskrit Library runs Tomcat *)
;
```

Button

```
value js_string_arg s =
  let delim delim s = delim ^ s ^ delim in
  delim "'" s
;
type js_funcall = { js_funid : string; js_funargs : list string }
;
value string_of_js_funcall f =
  let js_funargs = List.map js_string_arg f.js_funargs in
  f.js_funid ^ "(" ^ String.concat ",␣" js_funargs ^ ")"
;
value button ?id ?cl ?onclick label =
  let button = "button" in
  let attrs = add_opt_attrs
      [ ("onclick", Gen.opt_app string_of_js_funcall onclick)
      ; ("id", id)
      ; ("class", Gen.opt_app class_of cl)
```

```
      ] [] in
  let button_begin = xml_begin_with_att button attrs in
  let button_end = xml_end button in
  button_begin ^ label ^ button_end
;
(* Return a copy of the given string with special HTML characters represented by escaped sequences (e.g. '&' is replaced with "&amp;" ). *)
value escape s =
  let conversion_tbl =
    [ ("\"", "quot")
    ; ("&", "amp")
    ; ("'", "apos")
    ; ("<", "lt")
    ; (">", "gt")
    ] in
  let escape s =
    try "&" ^ List.assoc s conversion_tbl ^ ";" with [ Not_found -> s ] in
  let special_chars = Str.regexp
      ("[" ^ String.concat "" (conversion_tbl |> List.split |> fst) ^ " ]") in
  let subst s = s |> Str.matched_string |> escape in
  Str.global_substitute special_chars subst s
;
```

Module Web

module Web html = struct
Module Web reads localisation parameters from paths.ml, created by "make configure" in main directory, called by configure script. Describes all installation parameters and resources other than Install.
Dynamic html rendering, used by cgis

```
open Html;
```

truncation is the maximum number of solutions computed by the lexer. Too small a truncation limit will miss solutions, too large a truncation limit will provoke un unrecoverable choking server failure. This is relevant only for the parser (deprecated) mode. The graph interface has no limit.

```
value truncation = 10000
;
```

threshold for printing the list of explicit segmentation solutions

```
value max_count = 100 (* do not exceed - rather use the graphical interface *)
;
value cache_allowed = target = Station (* cache allowed only on Station *)
;
value cache_active = ref (if cache_allowed then "t" else "f")
;
(* For interface look-and-feel *)
value (display_morph_action, mouse_action_help) = match Paths.mouse_action with
  [ "CLICK" → ("onclick","Click")
  | "OVER" → ("onMouseover","Mouse")
  | _ → failwith "Unknown mouse action, change config file"
  ]
;
value cgi_bin name = Paths.cgi_dir_url ^ name
;
(* Call-backs as cgi binaries *)
value index_cgi = cgi_bin Paths.cgi_index (* index *)
and dummy_cgi = cgi_bin Paths.cgi_indexd (* index for dummies *)
and decls_cgi = cgi_bin Paths.cgi_decl (* declensions *)
and conjs_cgi = cgi_bin Paths.cgi_conj (* conjugations *)
and lemmatizer_cgi = cgi_bin Paths.cgi_lemmatizer (* lemmatizer *)
and reader_cgi = cgi_bin Paths.cgi_reader (* reader *)
and parser_cgi = cgi_bin Paths.cgi_parser (* parser *)
and graph_cgi = cgi_bin Paths.cgi_graph (* summarizer graphical interface *)
and user_aid_cgi = cgi_bin Paths.cgi_user_aid (* unknown chunks processing *)
and sandhier_cgi = cgi_bin Paths.cgi_sandhier (* sandhier *)
and corpus_manager_cgi = cgi_bin Paths.cgi_corpus_manager (* Corpus manager *)
and save_corpus_cgi = cgi_bin Paths.cgi_save_corpus
and mkdir_corpus_cgi = cgi_bin Paths.cgi_mkdir_corpus
;
value dico_page = Data.dico_page
;
value skt_dir_url = Paths.skt_dir_url
;
(* Relative paths of top directory of site and sub directories *)
value web_dico_url = skt_dir_url ^ "DICO/"
and mw_dico_url = skt_dir_url ^ "MW/"
and web_images_url = skt_dir_url ^ "IMAGES/"
and sanskrit_page_url l = skt_dir_url ^ (site_entry_page l)
and faq_page_url l = skt_dir_url ^ (faq_page l)
```

```
and portal_page_url l = skt_dir_url ^ (portal_page l)
;
(* style sheet built by Css module *)
value style_sheet = "style.css"
;
value css_file = dico_page style_sheet
;
(* javascript to fake dev UTF8 as VH *)
value deva_reader = "utf82VH.js"
;
(* Absolute URLs for cgis *)
value dico_page_url name = web_dico_url ^ name
;
value style_sheet_url = dico_page_url style_sheet
and deva_reader_url = dico_page_url deva_reader
and indexer_page_url l = dico_page_url (dico_index_page l)
and reader_page_url l = dico_page_url (dico_reader_page l)
and grammar_page_url l = dico_page_url (dico_grammar_page l)
and sandhi_page_url l = dico_page_url (dico_sandhi_page l)
and corpus_page_url l = dico_page_url (dico_corpus_page l)
;
value image name = web_images_url ^ name
;
value ocaml_logo = image "icon_ocaml.png"
and inria_logo = image "logo_inria.png"
and favicon = image "favicon.ico"
;
value reader_meta_title = title "Sanskrit Reader Companion"
and parser_meta_title = title "Sanskrit Reader Assistant"
and dico_title_fr = h1_title "Dictionnaire H&eacute;ritage du Sanscrit"
and dummy_title_fr = h1_title "Le sanscrit pour les nuls"
and dico_title_en = h1_title (if narrow_screen then "Sanskrit Lexicon"
                              else "Monier-Williams Dictionary")
and dummy_title_en = h1_title "Sanskrit made easy"
and stem_title_en = h1_title (if narrow_screen then "Sanskrit Stemmer"
                              else "Search for atomic inflected forms")
and reader_title = h1_title (if narrow_screen then "Sanskrit Reader"
                             else "The Sanskrit Reader Companion")
and parser_title = h1_title (if narrow_screen then "Sanskrit Parser"
                             else "The Sanskrit Parser Assistant")
```

```
and graph_meta_title = title "Sanskrit␣Segmenter␣Summary"
and user_aid_meta_title = title "User␣Feedback"
and interface_title = h1_title (if narrow_screen then "Summarizer"
                                else "Sanskrit␣Segmenter␣Summary")
and user_aid_title = h1_title (if narrow_screen then "User␣Feedback"
                              else "Feedback␣for␣Unknown␣Chunks")
;
value dico_title = fun
  [ French → dico_title_fr
  | English → dico_title_en
  ]
;
(* We set and reset output_channel to designate either a static html file under creation or
stdout to produce a cgi output dynamic page. Ugly. *)
value output_channel = ref stdout
;
value ps s = output_string output_channel.val s
and pc c = output_char output_channel.val c
and pi i = output_string output_channel.val (string_of_int i)
;
value line () = pc '\n'
and sp () = ps "␣"
and pl s = ps (s ^ "\n")
;
value meta_program l = List.iter pl (List.map meta_prefix l)
;
value javascript ref =
  xml_begin_with_att "script" [ ("type","text/javascript"); ("src",ref) ]
 (* Caution - necessary to separate begin and end *)
 ^ xml_end "script"
;
(* dyn=True for dynamic pages created by cgis, False for static pages in DICO *)
value deva_read_script dyn =
  let ref = if dyn then deva_reader_url
                   else deva_reader in
  javascript ref
;
value js_util_script dyn =
  let js_util_file = "util.js" in
  let prefix = if dyn then dico_page_url else (fun x → x) in
```

```
  javascript (prefix js_util_file)
;
value css_link dyn =
  let ref = if dyn then style_sheet_url (* dynamic page, absolute URL *)
            else style_sheet (* static page in DICO, relative URL *) in
  xml_empty_with_att "link" [ ("rel","stylesheet"); ("type","text/css");
                              ("href",ref); ("media","screen,tv") ]
;
value caml_inside dyn =
  let logo = if dyn then ocaml_logo else rel_ocaml_logo in
  let ocaml_logo = xml_empty_with_att "img"
      [ ("src",logo); ("alt","Le␣chameau␣Ocaml"); ("height","50") ] in
  anchor_ref ocaml_site ocaml_logo
and inria_inside dyn = (* Inria new logo - clickable *)
  let logo = if dyn then inria_logo else rel_inria_logo in
  let inria_logo = xml_empty_with_att "img"
      [ ("src",logo); ("alt","Logo␣Inria"); ("height","50") ] in
  anchor_ref inria_site inria_logo
;
value favicon dyn =
  let path = if dyn then favicon else rel_favicon in
  "<link␣rel=\"shortcut␣icon\"␣href=\"" ^ path ^ "\">"
;
value page_begin_dyn dyn title = do
  { doctype |> pl
  ; xml_begin_with_att "html" [] |> ps
  ; xml_begin "head" —> pl (* ( *)
  ; meta_program contents_instructions (* . *)
  ; title |> pl (* . *)
  ; meta_program title_instructions (* . *)
  ; css_link dyn |> pl (* . *)
  ; favicon dyn |> pl (* . *)
  ; deva_read_script dyn |> pl (* devanagari input *)(* . *)
  ; js_util_script dyn |> pl (* . *)
  ; xml_end "head" —> pl (* ) *)
  }
;
value open_html_file f title = do (* for building the Web services pages *)
  { output_channel.val := open_out f; page_begin_dyn False title }
;
```

```
value page_begin = page_begin_dyn True (* for cgi output page *)
;
value version lang =
  let lang_str =
      match lang with
      [ Some Html.French → "␣(French)"
      | Some Html.English → "␣(English)"
      | None → ""
      ] in
  h3_begin B3 ^ Date.version ^ lang_str ^ h3_end
;
value print_title lang title = do
  { table_begin Centered |> pl
  ; tr_begin |> ps
  ; th_begin |> ps
  ; title |> pl
  ; version lang |> pl
  ; th_end |> ps
  ; tr_end |> ps
  ; table_end |> pl
  }
and print_title_solid color lang title = do
  { table_begin (centered color) |> pl
  ; tr_begin |> ps
  ; th_begin |> ps
  ; title |> pl
  ; version lang |> pl
  ; th_end |> ps
  ; tr_end |> ps
  ; table_end |> pl
  }
;
value print_transliteration_help lang =
  if narrow_screen then () else do
  { "Transliteration␣help␣" —> ps
  ; anchor_ref (rel_faq_page_url lang ^ "#transliteration") "here" —> pl
  }
;
value transliteration_switch_default dft id =
  option_select_default_id id "t"
```

```
      [ (" Velthuis ","VH",dft ="VH") (* Default Velthuis *)
      ; ("    WX    ","WX",dft ="WX") (* Infamous WaX from U. Hyderabad *)
      ; ("    KH    ","KH",dft ="KH") (* Kyoto-Harvard *)
      ; ("   SLP1   ","SL",dft ="SL") (* Sanskrit Library Sloppy 1 *)
      ; ("Devanagari","DN",dft ="DN") (* Devanagari UTF-8 *)
      ; ("   IAST   ","RN",dft ="RN") (* Indological romanisation in UTF-8 *)
      ]
;
value print_transliteration_switch id =
  transliteration_switch_default Paths.default_transliteration id |> pl
;
value print_lexicon_select lexicon = do
  { "Lexicon Access " —> ps
  ; option_select_default "lex"
        [ ("    Heritage     ","SH","SH"=lexicon) (* Sanskrit Heritage *)
        ; (" Monier-Williams ","MW","MW"=lexicon) (* Monier-Williams *)
        ] |> pl
  }
;
value print_index_help lang =
  if narrow_screen then () else do
  { par_begin G2 |> pl
  ; html_break |> pl
  ; "Search for an entry matching an initial pattern:" —> ps
  ; html_break |> pl
  ; print_transliteration_help lang
  ; par_end |> pl (* G2 *)
  }
;
value print_dummy_help_en () =
  if narrow_screen then () else do
  { par_begin G2 |> pl
  ; "The simplified interface below allows search without diacritics" —> ps
  ; html_break |> pl
  ; "Proper names may be entered with an initial capital" —> pl
  ; par_end |> pl (* G2 *)
  }
;
value print_stemmer_help_en () =
  if narrow_screen then () else do
```

```
  { par_begin G2 |> ps
  ; "Submit␣candidate␣form␣and␣category" —> pl
  ; html_break |> pl
  ; "Forms␣ended␣in␣r␣should␣not␣be␣entered␣with␣final␣visarga" —> pl
  ; html_break |> pl
  ; "Compound␣words␣may␣be␣recognized␣with␣the␣Reader␣interface" —> pl
  ; html_break |> pl
  ; par_end |> pl (* G2 *)
  }
;
value open_page_with_margin width =
  let margin = string_of_int width ^ "pt" in
  let attr = [ noborder; nopadding; ("cellspacing",margin); fullwidth ] in do
  { table_begin_style (background Chamois) attr |> pl
  ; tr_begin |> ps (* closed by close_page_with_margin *)
  ; td_begin |> pl
  }
and close_page_with_margin () = do
  { html_break |> pl
  ; td_end |> ps
  ; tr_end |> ps
  ; table_end |> pl
  }
;
value indexer_page l = dico_page (dico_index_page l) (* mk_index_page *)
and grammar_page l = dico_page (dico_grammar_page l) (* mk_grammar_page *)
and reader_page l = dico_page (dico_reader_page l) (* mk_reader_page *)
and sandhi_page l = dico_page (dico_sandhi_page l) (* mk_sandhi_page *)
and corpus_page l = dico_page (dico_corpus_page l) (* mk_corpus_page *)
;
value print_site_map dyn lang = (* the various Web services of the site *)
  if dyn then do
  { anchor_ref (sanskrit_page_url lang) (emph "Top") |> ps; "␣|␣" —> pl
  ; anchor_ref (indexer_page_url lang) (emph "Index") |> ps; "␣|␣" —> pl
  ; anchor_ref (indexer_page_url lang ^ "#stemmer") (emph "Stemmer") |> ps; "␣|␣" —> pl
  ; anchor_ref (grammar_page_url lang) (emph "Grammar") |> ps; "␣|␣" —> pl
  ; anchor_ref (sandhi_page_url lang) (emph "Sandhi") |> ps; "␣|␣" —> pl
  ; anchor_ref (reader_page_url lang) (emph "Reader") |> ps; "␣|␣" —> pl
  ; anchor_ref (corpus_page_url lang) (emph "Corpus") |> ps; "␣|␣" —> pl
  ; anchor_ref (faq_page_url lang) (emph "Help") |> ps; "␣|␣" —> pl
```

```
  ; anchor_ref (portal_page_url lang) (emph "Portal") |> pl
  }
 else do
  { anchor_ref (rel_sanskrit_page_url lang) (emph "Top") |> ps; " | " --> pl
  ; anchor_ref (dico_index_page lang) (emph "Index") |> ps; " | " --> pl
  ; anchor_ref (dico_index_page lang ^ "#stemmer") (emph "Stemmer") |> ps; " | " --> pl
  ; anchor_ref (dico_grammar_page lang) (emph "Grammar") |> ps; " | " --> pl
  ; anchor_ref (dico_sandhi_page lang) (emph "Sandhi") |> ps; " | " --> pl
  ; anchor_ref (dico_reader_page lang) (emph "Reader") |> ps; " | " --> pl
  ; anchor_ref (dico_corpus_page lang) (emph "Corpus") |> ps; " | " --> pl
  ; anchor_ref (rel_faq_page_url lang) (emph "Help") |> ps; " | " --> pl
  ; anchor_ref (rel_portal_page_url lang) (emph "Portal") |> pl
  }
;
value pad () = do (* ad-hoc vertical padding to make room for the bandeau *)
  { table_begin Pad60 |> pl
  ; tr_begin |> ps
  ; td_begin |> ps
  ; td_end |> ps
  ; tr_end |> ps
  ; table_end |> pl
  }
;
value print_bandeau_enpied_dyn dyn lang color = do
  { pad () (* necessary padding to avoid hiding by bandeau *)
  ; elt_begin "div" Enpied |> pl
  ; table_begin Bandeau |> ps
  ; tr_begin |> ps (* main row begin *)
  ; td_begin |> pl
  ; caml_inside dyn |> pl
  ; td_end |> ps
  ; td_begin |> pl
  ; table_begin Tcenter |> pl
  ; tr_begin |> ps
  ; td_begin |> pl
  ; print_site_map dyn lang
  ; td_end |> ps
  ; tr_end |> ps
  ; tr_begin |> ps
  ; td_begin |> pl
```

```
  ; copyright |> ps
  ; td_end |> ps
  ; tr_end |> ps (* copyright row end *)
  ; table_end |> ps
  ; td_end |> ps
  ; td_begin |> pl
  ; inria_inside dyn |> pl
  ; html_break |> ps
  ; td_end |> ps
  ; tr_end |> ps
  ; table_end |> ps (* Bandeau *)
  ; xml_end "div" —> pl (* end Enpied *)
  }
;
(* Simputer - legacy code - could be reused for smartphones *)
value print_bandeau_entete color =
  let margin_bottom height = "margin-bottom:" ^ points height in
  let interval height = do
    { tr_begin |> ps
    ; td [ ("width","100%"); ("style",margin_bottom height) ] |> pl
    ; tr_end |> ps
    } in do
  { table_begin_style (background color)
             [ noborder; nopadding; ("cellspacing","5pt"); fullwidth ] |> pl
  ; interval 10
  ; tr_begin |> ps
  ; xml_begin_with_att "td" [ fullwidth; ("align","center") ] |> pl
  ; print_site_map True Html.English
  ; td_end |> ps
  ; tr_end |> ps
  ; interval 10
  ; table_end |> pl
  }
;
value page_end_dyn dyn lang bandeau = do
  { match Html.target with
    [ Html.Simputer → ()
    | Html.Computer | Html.Station | Html.Server
      → if bandeau then print_bandeau_enpied_dyn dyn lang Cyan else ()
    ]
```

```
  ; body_end |> pl
  ; xml_end "html" —> pl
  }
;
value page_end = page_end_dyn True
;
value close_html_file lang b = do
  { page_end_dyn False lang b; close_out output_channel.val }
;
value close_html_dico () = close_html_file Html.French True
;
value http_header = "Content-Type:␣text/html\n"
;
(* Print the HTTP header only when it is required, i.e. only if it is a CGI output. *)
value maybe_http_header () =
  if output_channel.val = stdout then http_header |> pl else ()
;
value javascript_tooltip ="wz_tooltip.js"
;
(* This could be any absolute server where Platform is installed *)
(* Maybe should be put back in config? but versioning problem... *)
value remote_server_host = "http://sanskrit.inria.fr/"
;
(* This toggle controls accessibility of University of Hyderabad tools. It is controled by
ML/SCLpaths.ml, which is not part of the git repository, and is initialised by default to
SETUP/dummy_SCLpaths.ml at make time. *)
value scl_toggle =
  ¬ (SCLpaths.scl_url ="") (* True if SCL tools are installed *)
;
value corpus_read_only =
  match target with
  [ Station → False
  | Computer | Server | Simputer → True
  ]
;
value interaction_modes_default mode =
  [ ("␣Summary␣","g",mode ="g")
  ; ("␣Tagging␣","t",mode ="t")
  ; ("␣Parsing␣","p",mode ="p")
  ] @ if scl_toggle then (* Needs the SCL tools *)
```

```
  [ ("␣Analysis␣","o",mode ="o") ] else [ ]
;
value interaction_modes =
  interaction_modes_default "g" (* default summary mode *)
;
```

NB Interface and Parser have their own prelude.
reader_prelude is invoked by Parser through Rank and by *Mk_reader_page*

```
value reader_prelude title = do
  { http_header |> pl
  ; page_begin reader_meta_title
  ; body_begin Chamois_back |> pl
  ; if scl_toggle then (* external call SCL (experimental) *)
        javascript (SCLpaths.scl_url ^ javascript_tooltip) |> pl
    else ()
  ; title |> pl
  ; open_page_with_margin 15
  }
;
(* cgi invocation *)
value cgi_begin cgi convert =
  xml_begin_with_att "form"
    [ ("action",cgi); ("method","get")
    ; ("onsubmit","return␣" ^ convert ^ "()") ] (* input conversion script *)
  ^ elt_begin "span" Latin12
and cgi_reader_begin cgi convert = (* do not use for pages with multiple cgi *)
  xml_begin_with_att "form"
    [ ("id","this_form"); ("action",cgi); ("method","get")
    ; ("onsubmit","return␣" ^ convert ^ "()") ] (* input conversion script *)
  ^ elt_begin "span" Latin12
and cgi_end = xml_end "span" ^ xml_end "form"
;
```

Failsafe aborting of cgi invocation

```
value abort lang s1 s2 = do
  { table_begin_style (centered Yellow) [ noborder; ("cellspacing","20pt") ] |> pl
  ; tr_begin |> ps
  ; th_begin |> ps
  ; html_red s1 |> ps (* Report anomaly *)
  ; html_blue s2 |> pl (* Optional specific message *)
  ; th_end |> ps
```

```
  ; tr_end |> ps
  ; table_end |> pl
  ; close_page_with_margin ()
  ; page_end lang True
  }
;
(* Build an HTML page to report error. *)
value error_page title_str msg submsg =
  do
  { maybe_http_header ()
  ; page_begin (title title_str)
  ; body_begin Chamois_back |> pl
  ; open_page_with_margin 15
  ; h1_title title_str |> print_title (Some default_language)
  ; abort default_language msg submsg
  }
;
(* invalid_corpus_mode_page expected_mode current_mode generates an HTML on output_channel
to notify the user that the requested operation on the corpus is available only in expected_mode
and not in current_mode. *)
value invalid_corpus_permission_page expected current =
  error_page "Corpus Manager" "Invalid permission "
    ("Expected permission: " ^ expected ^ " | Current permission: " ^ current)
;
```

Module Css

Stand-alone module for generating the css file style.css

```
open Html; (* class_of style *)
open Web;

value cascade (elt, cla) = (* cascading style sheets generator *)
  elt ^ "." ^ (class_of cla) ^ " {" ^ (style cla) ^ "}"
;
value sheets = (* cascading style sheets data *)
  [ ("a",Blue_); ("a",Green_); ("a",Navy_); ("a",Red_)
  ; ("h1",Title); ("h1",C1); ("h2",C2); ("h3",C3)
  ; ("h1",B1); ("h2",B2); ("h3",B3); ("p",G2)
  ; ("div",Latin12); ("div",Latin16); ("div",Enpied); ("div",Center_)
  ; ("span",Alphabet); ("span",Deva); ("span",Trans12); ("span",Devared_)
```

```
  ; ("span",Red_); ("span",Roma16o); ("span",Magenta_); ("span",Blue_)
  ; ("span",Green_); ("span",Latin12); ("span",Latin16); ("span",Trans16)
  ; ("span",Title); ("span",C1); ("span",C2); ("span",C3); ("span",Deva20c)
  ; ("span",B1); ("span",B2); ("span",B3); ("span",Header_deva); ("span",Math)
  ; ("span",Devac); ("span",Header_tran); ("span",Deva16); ("span",Deva16c)
  ; ("body",Mauve_back); ("body",Pink_back); ("body",Chamois_back)
  ; ("table",Bandeau); ("table",Center_); ("table",Body); ("table",Pad60)
  ; ("table",Yellow_back); ("table",Yellow_cent); ("table",Deep_sky_cent)
  ; ("table",Salmon_back); ("table",Aquamarine_back)
  ; ("table",Mauve_back); ("table",Magenta_back); ("table",Mauve_cent)
  ; ("table",Cyan_back); ("table",Cyan_cent); ("table",Lavender_cent)
  ; ("table",Gold_back); ("table",Gold_cent); ("table",Inflexion)
  ; ("table",Chamois_back); ("table",Blue_back) ; ("table",Green_back)
  ; ("table",Brown_back); ("table",Lime_back); ("table",Deep_sky_back)
  ; ("table",Carmin_back); ("table",Orange_back); ("table",Red_back)
  ; ("table",Grey_back); ("table",Pink_back); ("table",Spacing20)
  ; ("table",Light_blue_back); ("table",Lavender_back); ("table",Lawngreen_back)
  ; ("td",Yellow_back); ("td",Yellow_cent); ("td",Deep_sky_cent)
  ; ("td",Salmon_back); ("td",Aquamarine_back)
  ; ("td",Mauve_back); ("td",Magenta_back); ("td",Mauve_cent)
  ; ("td",Cyan_back); ("td",Cyan_cent); ("td",Lavender_cent)
  ; ("td",Gold_back); ("td",Gold_cent); ("td",Inflexion)
  ; ("td",Chamois_back); ("td",Blue_back) ; ("td",Green_back)
  ; ("td",Brown_back); ("td",Lime_back); ("td",Deep_sky_back)
  ; ("td",Carmin_back); ("td",Orange_back); ("td",Red_back)
  ; ("td",Grey_back); ("td",Pink_back); ("td",Spacing20)
  ; ("td",Light_blue_back); ("td",Lavender_back); ("td",Lawngreen_back)
  ; ("th",Cell5); ("th",Cell10); ("th",Border2); ("td",Center_)
  ; ("table",Centered); ("table",Tcenter) ; ("", Hidden_)
  ];
value css_decls =
  [ "a:link {color: Blue}"
  ; "a:visited {color: Purple}"
  ; "a:active {color: Fuchsia}"
  ; "img {border: 0}"
  ; "li " ^ "{" ^ (style B3) ^ "}" (* patch for line numbers in corpus *)
  ] @ List.map cascade sheets
;
value pop_up_spec =
  "#popBox { position: absolute; z-index: 2; background: " ^ rgb Mauve ^
```

```
        ";␣padding:␣0.3em;␣border:␣none;␣white-space:␣nowrap;␣}"
;
value print_css_file () =
  let output_channel = open_out css_file in
  let ps = output_string output_channel in
  let pl s = ps (s ^ "\n") in
  let css_style l = List.iter pl l in do
  { css_style css_decls
  ; pl pop_up_spec
  ; close_out output_channel
  }
;
print_css_file ()
;
```

Module Cgi

CGI utilities
Decoding utilities, author Daniel de Rauglaudre
ddr begin

```
value hexa_val conf =
  match conf with
  [ '0'..'9' → Char.code conf − Char.code '0'
  | 'a'..'f' → Char.code conf − Char.code 'a' + 10
  | 'A'..'F' → Char.code conf − Char.code 'A' + 10
  | _ → 0
  ]
;
value decode_url s =
  let rec need_decode i =
    if i < String.length s then
      match s.[i] with
      [ '%' | '+' → True
      | _ → need_decode (succ i)
      ]
    else False in
  let rec compute_len i i1 =
    if i < String.length s then
      let i =
```

```
        match s.[i] with
        [ '%' when i + 2 < String.length s → i + 3
        | _ → succ i
        ]
      in
      compute_len i (succ i1)
    else i1 in
  let rec copy_decode_in s1 i i1 =
    if i < String.length s then
      let i =
        match s.[i] with
        [ '%' when i + 2 < String.length s →
            let v = hexa_val s.[i + 1] × 16 + hexa_val s.[i + 2] in
            do {Bytes.set s1 i1 (Char.chr v); i + 3}
        | '+' → do {Bytes.set s1 i1 ' '; succ i}
        | x → do {Bytes.set s1 i1 x; succ i}
        ] in
      copy_decode_in s1 i (succ i1)
    else s1 in
  let rec strip_heading_and_trailing_spaces s =
    if String.length s > 0 then
      if s.[0] ≡ ' ' then
        strip_heading_and_trailing_spaces (String.sub s 1 (String.length s − 1))
      else if s.[String.length s − 1] ≡ ' ' then
        strip_heading_and_trailing_spaces (String.sub s 0 (String.length s − 1))
      else s
    else s in
  if need_decode 0 then
    let len = compute_len 0 0 in
    let s1 = Bytes.create len in
    strip_heading_and_trailing_spaces (Bytes.to_string (copy_decode_in s1 0 0))
  else s
;
```

(∗ converts a string coming from the URL into an a-list; the string is a sequence of pairs key=value separated by ; or & ∗)

```
value create_env s =
  let rec get_assoc beg i =
    if i ≡ String.length s then
      if i ≡ beg then [] else [String.sub s beg (i − beg)]
    else if s.[i] ≡ ';' ∨ s.[i] ≡ '&' then
```

```
      let next_i = succ i in
      [String.sub s beg (i - beg) :: get_assoc next_i next_i]
    else get_assoc beg (succ i) in
  let rec separate i s =
    if i = String.length s then (s, "")
    else if s.[i] ≡ '=' then
      (String.sub s 0 i, String.sub s (succ i) (String.length s - succ i))
    else separate (succ i) s in
  List.map (separate 0) (get_assoc 0 0)
;
```

ddr end

```
value get key alist default =
  try List.assoc key alist with [ Not_found → default ]
;
value decoded_get key default alist = decode_url (get key alist default)
;
value query_string_env_var = "QUERY_STRING"
;
value query_string () =
  try Sys.getenv query_string_env_var with [ Not_found → "" ]
;
value url_encode s =
  let hexa_str c = Printf.sprintf "%.2X" (Char.code c) in
```

Reference: RFC 3986 appendix A

```
  let url_encode = fun
    (* Unreserved characters *)
    [ 'a' .. 'z' | 'A' .. 'Z' | '0' .. '9' | '-' | '.' | '_' | '~' as c →
      String.make 1 c
    (* Special case of the space character *)
    | ' ' → "+"
    (* Reserved characters *)
    | c → "%" ^ hexa_str c
    ] in
  let char_of_string s =
    if String.length s = 1 then s.[0] else failwith "char_of_string" in
  let subst s = s |> Str.matched_string |> char_of_string |> url_encode in
  let any_char = Str.regexp ".\\|\n" in
  Str.global_substitute any_char subst s
;
```

```
value query_of_env env =
  String.concat "&" (List.map (fun (k, v) → k ^ "=" ^ url_encode v) env)
;
value url ?query ?fragment path =
  let opt_part prefix = fun
    [ None → ""
    | Some part → prefix ^ part
    ] in
  let query_part = opt_part "?" query in
  let fragment_part = opt_part "#" fragment in
  path ^ query_part ^ fragment_part
;
```

Module Morpho_html

This module contains various service utilities for CGI programs

```
open Html;
open Web; (* ps etc. *)
open Multilingual; (* Roma Deva *)
```

```
module Out_chan = struct value chan = Web.output_channel; end;
module Morpho = Morpho.Morpho_out Out_chan;
```

This loads dynamically the MW exceptions database

```
value mw_defining_page s =
  let mw_exceptions =
    try (Gen.gobble Data.public_mw_exc_file : Deco.deco int)
    with [ _ → failwith "mw_exceptions" ] in
  Chapters.mw_defining_page_exc s mw_exceptions
;
(* Absolute url on local site *)
value url s =
  let (page, pref) = match lexicon_toggle.val with
    [ "SH" → (web_dico_url ^ Chapters.sh_defining_page s,"")
    | "MW" → (mw_dico_url ^ mw_defining_page s,"H_")
    | _ → failwith "Unknown␣lexicon"
    ] in
  page ^ "#" ^ pref ^ Encode.anchor s
;
value url_cache s =
```

```
  mw_dico_url ^ mw_defining_page s ^ "#" ^ Encode.anchor s
;
(* Romanisation of Sanskrit *)
value skt_roma s = italics (Transduction.skt_to_html s)
(* Function skt_roma differs from Encode.skt_to_roma because it does not go through en-
coding s as a word, and the complications of dealing with possible hiatus. *)
;
value skt_red s = html_red (skt_roma s)
;
value skt_anchor cached font form = (* for Declension Conjugation *)
  let s = match font with
          [ Deva → deva20_blue_center (Encode.skt_raw_strip_to_deva form)
          | Roma → skt_roma form (* no stripping in Roma *)
          ]
  and url_function = if cached then url_cache else url in
  anchor Navy_ (url_function form) s
;
value skt_anchor_R cached = skt_anchor cached Roma (* for Declension, Indexer *)
and skt_anchor_R2 s s' = anchor Navy_ (url s) (skt_roma s') (* for Indexer *)
;
value no_hom entry = (* low-level string hacking *)
  match (String.sub entry ((String.length entry) − 1) 1) with
  [ "1"| "2"| "3"| "4"| "5"| "6"| "7"| "8"| "9" → False
  | _ → True
  ]
;
(* Used for printing MW in indexing mode *)
(* Note the difference between word and entry, word is the normalized form of entry. We
need entry to link to the MW page, where it is unnormalized *)
value skt_anchor_M word entry page cache =
  let anchor_used = if cache then anchor_graph else anchor in
  let anc = mw_dico_url ^ page ^ "#" ^ entry in
  let anchor_mw = anchor_used Navy_ anc in
  let vocable = if no_hom entry then word
                else let pos = (String.length entry) − 1 in
                     word ^ "#" ^ (String.sub entry pos 1) in
  anchor_mw (skt_roma vocable)
;
value skt_graph_anchor_R cache form =
  let s = skt_roma form in
```

```
  let url_function = if cache then url_cache else url in
  anchor_graph Navy_ (url_function form) s
;
value print_stem w = Canon.uniromcode w |> ps (* w in lexicon or not *)
and print_chunk w = Canon.uniromcode w |> ps
and print_entry w = skt_anchor_R False (Canon.decode w) |> ps (* w in lexicon *)
and print_ext_entry ps w = skt_anchor_R False (Canon.decode w) |> ps (* idem *)
and print_cache w = skt_anchor_R True (Canon.decode w) |> ps
and print_graph_entry w = skt_graph_anchor_R False (Canon.decode w) |> ps
and print_graph_cache w = skt_graph_anchor_R True (Canon.decode w) |> ps
;
```

Used in *Indexer* and *Lemmatizer*

```
value print_inflected gen word inverse = do
  { Morpho.print_inv_morpho print_entry print_stem print_chunk word (0,0)
                                  gen inverse
  ; html_break |> pl
  }
;
(* Used in Lexer.print_morph *)
value print_inflected_link pvs cached =
  let print_fun = if cached then print_cache else print_entry in
  Morpho.print_inv_morpho_link pvs print_fun print_stem print_chunk
;
(* Used in Interface to print the lemmas *)
value print_graph_link pvs cached =
  let print_fun = if cached then print_graph_cache else print_graph_entry in
  Morpho.print_inv_morpho_link pvs print_fun print_stem print_chunk
;
(* Final visarga form for display: final s and r are replaced by visarga. There is some information loss here, since -ar and -a.h do not have the same behaviour with external sandhi, eg punar-api, antar-a'nga, antar-gata, etc. For this reason the morphological tables do not keep forms in terminal sandhi, and distinguish forms ended in -as and -ar. It should not be applied to stems, only to padas *)
value visargify rw = Word.mirror
  (match rw with
     [ [ 48 (* s *) :: r ] | [ 43 (* r *) :: r ] → [ 16 (* .h *) :: r ]
     | _ → rw
     ])
;
```

```
value final w = visargify (Word.mirror w) (* Declension, Conjugation *)
;
value print_final rw = print_chunk (visargify rw) (* Interface *)
;
value hdecode word = Transduction.skt_to_html (Canon.decode word)
;
value html_blue_off offset text =
  (* Temporary use of title attribute for XHTML 1.0 Strict offset recording, *)
  (* should be replaced by data-offset for future HTML 5 compliance. *)
  (* This is only needed for the SL annotator interface. *)
  (* It has the unpleasant side effect of showing offsets on mouse over. *)
  let offset_attr offset = ("title",string_of_int offset) in
  (elt_begin_attrs [ offset_attr offset ] "span" Blue_) ^ text ^ span_end
;
(* indicates offset of segment in attribute "title" of Blue_ span *)
value blue_word_off word offset = (* deprecated *)
  html_blue_off offset (emph (hdecode word))
;
value print_sandhi u v w = do
  { html_magenta (hdecode (visargify u)) |> ps (* visarga form *)
  ; html_green "|" --> ps
  ; html_magenta (hdecode v) |> ps
  ; html_blue " &rarr; " --> ps (* -¿ *)
  ; html_red (hdecode w) |> ps
  }
;
value print_signifiant rword =
  let word = visargify rword in (* visarga form : final s and r visarged *)
  html_blue (hdecode word) |> ps
;
(* used in Lexer.print_segment with offset indication *)
value print_signifiant_off rword offset =
  let word = visargify rword in (* visarga form : final s and r visarged *)
  blue_word_off word offset |> ps
;
(* used in Lexer.print_proj *)
value print_signifiant_yellow rword = do
  { th_begin |> ps
  ; table_begin_style (background Yellow) [ padding5 ] |> pl
  ; td_begin |> ps
```

```
  ; print_signifiant rword
  ; td_end |> ps
  ; table_end |> ps
  ; th_end |> ps
  }
;
```

Module Chapters

module Chapter = struct
This module ensures that each individual HTML page of the DICO site is not too big, by slicing them into small chapters determined by prefixes of the vocables they define.

```
type chapters = list Word.word (* chapter boundaries *)
;
```

The chapter mechanism - slicing Dico into moderate size html pages

```
value (dico_chapters : chapters) = List.map Encode.code_string
  (* "a" in 1.html *)
  [ "ad" (* 2.html *)
  ; "anu" (* 3.html *)
  ; "ap" (* 4.html *)
  ; "abh" (* 5.html *)
  ; "ar" (* 6.html *)
  ; "av" (* 7.html *)
  ; "ast" (* 8.html *)
  ; "aa" (* 9.html *)
  ; "aam" (* 10.html *)
  ; "i" (* 11.html *)
  ; "ii" (* 12.html *)
  ; "u" (* 13.html *)
  ; "ut" (* 14.html *)
  ; "up" (* 15.html *)
  ; "u.s" (* 16.html *)
  ; ".r" (* 17.html *)
  ; "k" (* 18.html *)
  ; "kan" (* 19.html *)
  ; "kaa" (* 20.html *)
  ; "kaay" (* 21.html *)
  ; "k.r" (* 22.html *)
```

```
; "k.s" (* 23.html *)
; "g" (* 24.html *)
; "g.r" (* 25.html *)
; "c" (* 26.html *)
; "j" (* 27.html *)
; "jh" (* 28.html *)
; "taa" (* 29.html *)
; "t.r" (* 30.html *)
; "d" (* 31.html *)
; "di" (* 32.html *)
; "dev" (* 33.html *)
; "dh" (* 34.html *)
; "naa" (* 35.html *)
; "ni" (* 36.html *)
; "nii" (* 37.html *)
; "p" (* 38.html *)
; "par" (* 39.html *)
; "paa" (* 40.html *)
; "pi" (* 41.html *)
; "po" (* 42.html *)
; "prat" (* 43.html *)
; "prab" (* 44.html *)
; "praa" (* 45.html *)
; "bal" (* 46.html *)
; "bh" (* 47.html *)
; "bhe" (* 48.html *)
; "man" (* 49.html *)
; "mar" (* 50.html *)
; "mi" (* 51.html *)
; "muu" (* 52.html *)
; "y" (* 53.html *)
; "r" (* 54.html *)
; "ro" (* 55.html *)
; "lam" (* 56.html *)
; "v" (* 57.html *)
; "vaa" (* 58.html *)
; "vi" (* 59.html *)
; "vip" (* 60.html *)
; "vi.s" (* 61.html *)
; "v.r" (* 62.html *)
```

```
  ; "z" (* 63.html *)
  ; "zu" (* 64.html *)
  ; ".s" (* 65.html *)
  ; "s" (* 66.html *)
  ; "san" (* 67.html *)
  ; "sap" (* 68.html *)
  ; "sar" (* 69.html *)
  ; "sii" (* 70.html *)
  ; "sur" (* 71.html *)
  ; "sn" (* 72.html *)
  ; "h" (* 73.html *)
  ]
;
value (mw_chapters : chapters) = List.map Encode.code_string
  [ "agni" (* 2.html *)
  ; "acira" (* 3.html *)
  ; "atikandaka" (* 4.html *)
  ; "adeya" (* 5.html *)
  ; "adhyaavap" (* 6.html *)
  ; "anaarambha.na" (* 7.html *)
  ; "anunii" (* 8.html *)
  ; "anu.sa.n.da" (* 9.html *)
  ; "anti" (* 10.html *)
  ; "apatrap" (* 11.html *)
  ; "apaas" (* 12.html *)
  ; "abuddha" (* 13.html *)
  ; "abhiprastu" (* 14.html *)
  ; "abhisa.mnam" (* 15.html *)
  ; "abhra" (* 16.html *)
  ; "ambhi.nii" (* 17.html *)
  ; "aruza" (* 18.html *)
  ; "arvaac" (* 19.html *)
  ; "avatap" (* 20.html *)
  ; "avas.rj" (* 21.html *)
  ; "avo.sa" (* 22.html *)
  ; "azvanta" (* 23.html *)
  ; "asukha" (* 24.html *)
  ; "ahe" (* 25.html *)
  ; "aa" (* 26.html *)
  ; "aacchid" (* 27.html *)
```

```
; "aaditeya" (* 28.html *)
; "aapaali" (* 29.html *)
; "aara.t.ta" (* 30.html *)
; "aav.r" (* 31.html *)
; "aahitu.n.dika" (* 32.html *)
; "i" (* 33.html *)
; "i.s" (* 34.html *)
; "ii" (* 35.html *)
; "u" (* 36.html *)
; "uttama" (* 37.html *)
; "utpat" (* 38.html *)
; "udak" (* 39.html *)
; "udyam" (* 40.html *)
; "upajan" (* 41.html *)
; "uparuc" (* 42.html *)
; "upaacar" (* 43.html *)
; "ulkaa" (* 44.html *)
; "uu" (* 45.html *)
; ".r" (* 46.html *)
; ".rr" (* 47.html *)
; ".l" (* 48.html *)
; ".lr" (* 49.html *)
; "e" (* 50.html *)
; "et.r" (* 51.html *)
; "ai" (* 52.html *)
; "o" (* 53.html *)
; "au" (* 54.html *)
; "k" (* 55.html *)
; "ka.n.th" (* 56.html *)
; "kapi" (* 57.html *)
; "karakaayu" (* 58.html *)
; "karma.sa" (* 59.html *)
; "kazcana" (* 60.html *)
; "kaaniita" (* 61.html *)
; "kaartsna" (* 62.html *)
; "kaaz" (* 63.html *)
; "kiim" (* 64.html *)
; "ku.na" (* 65.html *)
; "kuyoga" (* 66.html *)
; "kuu.t" (* 67.html *)
```

```
; "k.rp" (* 68.html *)
; "kela" (* 69.html *)
; "ko.s.na" (* 70.html *)
; "kra.s.tavya" (* 71.html *)
; "k.santavya" (* 72.html *)
; "k.sud" (* 73.html *)
; "kh" (* 74.html *)
; "khav" (* 75.html *)
; "g" (* 76.html *)
; "gandharva" (* 77.html *)
; "gav" (* 78.html *)
; "giita" (* 79.html *)
; "guh" (* 80.html *)
; "go" (* 81.html *)
; "godha" (* 82.html *)
; "graama" (* 83.html *)
; "gh" (* 84.html *)
; "f" (* 85.html *)
; "c" (* 86.html *)
; "catas.r" (* 87.html *)
; "candhana" (* 88.html *)
; "caara" (* 89.html *)
; "citka.nakantha" (* 90.html *)
; "caitra" (* 91.html *)
; "ch" (* 92.html *)
; "j" (* 93.html *)
; "jam" (* 94.html *)
; "jala.daa" (* 95.html *)
; "jina" (* 96.html *)
; "j~naa" (* 97.html *)
; "jh" (* 98.html *)
; "~n" (* 99.html *)
; ".t" (* 100.html *)
; ".th" (* 101.html *)
; ".d" (* 102.html *)
; ".dh" (* 103.html *)
; ".n" (* 104.html *)
; "t" (* 105.html *)
; "tanaka" (* 106.html *)
; "tavas" (* 107.html *)
```

```
; "taavac" (* 108.html *)
; "tuk" (* 109.html *)
; "t.r.naafku" (* 110.html *)
; "tri" (* 111.html *)
; "trifkh" (* 112.html *)
; "th" (* 113.html *)
; "d" (* 114.html *)
; "dandaza" (* 115.html *)
; "dahara" (* 116.html *)
; "dina" (* 117.html *)
; "diirgha" (* 118.html *)
; "dur" (* 119.html *)
; "durdhar.sa" (* 120.html *)
; "duraaka" (* 121.html *)
; "devajana" (* 122.html *)
; "deva.ta" (* 123.html *)
; "dyuka" (* 124.html *)
; "dvaa.mdvika" (* 125.html *)
; "dvai" (* 126.html *)
; "dh" (* 127.html *)
; "dhari.ni" (* 128.html *)
; "dharka.ta" (* 129.html *)
; "dhuu" (* 130.html *)
; "dhva~nj" (* 131.html *)
; "n" (* 132.html *)
; "nad" (* 133.html *)
; "narda.taka" (* 134.html *)
; "naagammaa" (* 135.html *)
; "naarifga" (* 136.html *)
; "ni.h" (* 137.html *)
; "niryuktika" (* 138.html *)
; "niguh" (* 139.html *)
; "nimitta" (* 140.html *)
; "niryat" (* 141.html *)
; "ni.skira" (* 142.html *)
; "niilafgu" (* 143.html *)
; "naivaki" (* 144.html *)
; "p" (* 145.html *)
; "pa~nc" (* 146.html *)
; "pa.t" (* 147.html *)
```

```
; "pad" (* 148.html *)
; "payora" (* 149.html *)
; "paraacar" (* 150.html *)
; "paridih" (* 151.html *)
; "parividhaav" (* 152.html *)
; "par.n" (* 153.html *)
; "pavaru" (* 154.html *)
; "paa.daliipura" (* 155.html *)
; "paapacaka" (* 156.html *)
; "paava.s.turikeya" (* 157.html *)
; "pipi.svat" (* 158.html *)
; "pu.n.dariika" (* 159.html *)
; "pura~njara" (* 160.html *)
; "pu.skaletra" (* 161.html *)
; "puul" (* 162.html *)
; "painya" (* 163.html *)
; "prak.rrt" (* 164.html *)
; "pra.nij" (* 165.html *)
; "pratika" (* 166.html *)
; "prativid" (* 167.html *)
; "pratyabhiprasthaa" (* 168.html *)
; "pradhuu" (* 169.html *)
; "pramii" (* 170.html *)
; "pravical" (* 171.html *)
; "prasah" (* 172.html *)
; "praa.mzu" (* 173.html *)
; "praatikaa" (* 174.html *)
; "priitu" (* 175.html *)
; "ph" (* 176.html *)
; "b" (* 177.html *)
; "balaasa" (* 178.html *)
; "bahiinara" (* 179.html *)
; "bid" (* 180.html *)
; "b.rh" (* 181.html *)
; "brahman" (* 182.html *)
; "braadhnaayanya" (* 183.html *)
; "bh" (* 184.html *)
; "bhand" (* 185.html *)
; "bhaziraa" (* 186.html *)
; "bhaava" (* 187.html *)
```

```
; "bhiilabhuu.sa.naa" (* 188.html *)
; "bhuu" (* 189.html *)
; "bhuu.hkhaara" (* 190.html *)
; "bhraj" (* 191.html *)
; "m" (* 192.html *)
; "ma.nittha" (* 193.html *)
; "madhu" (* 194.html *)
; "madhva" (* 195.html *)
; "manauu" (* 196.html *)
; "marb" (* 197.html *)
; "mah" (* 198.html *)
; "mahaaprabhaava" (* 199.html *)
; "mahaazairii.sa" (* 200.html *)
; "maa.msp.r.s.ta" (* 201.html *)
; "maanava" (* 202.html *)
; "maas" (* 203.html *)
; "muku.ta" (* 204.html *)
; "mummuni" (* 205.html *)
; "m.r" (* 206.html *)
; "m.r.saalaka" (* 207.html *)
; "moci" (* 208.html *)
; "y" (* 209.html *)
; "yata" (* 210.html *)
; "yam" (* 211.html *)
; "yaak.rtka" (* 212.html *)
; "yuvan" (* 213.html *)
; "r" (* 214.html *)
; "ra.t" (* 215.html *)
; "ram" (* 216.html *)
; "rasna" (* 217.html *)
; "raajakineya" (* 218.html *)
; "raayaana" (* 219.html *)
; "ruddha" (* 220.html *)
; "ro.nii" (* 221.html *)
; "l" (* 222.html *)
; "lataa" (* 223.html *)
; "laalii" (* 224.html *)
; "lok" (* 225.html *)
; "v" (* 226.html *)
; "va~ncati" (* 227.html *)
```

```
; "vanara" (* 228.html *)
; "varola" (* 229.html *)
; "valbh" (* 230.html *)
; "vask" (* 231.html *)
; "vaaca" (* 232.html *)
; "vaayu" (* 233.html *)
; "vaalguda" (* 234.html *)
; "vi" (* 235.html *)
; "vi.mza" (* 236.html *)
; "vicitra" (* 237.html *)
; "vid" (* 238.html *)
; "vidhaav" (* 239.html *)
; "vipadumaka" (* 240.html *)
; "vimala" (* 241.html *)
; "vilinaatha" (* 242.html *)
; "vizii" (* 243.html *)
; "vizvi" (* 244.html *)
; "vi.sayaka" (* 245.html *)
; "vi.spanda" (* 246.html *)
; "viir" (* 247.html *)
; "v.rddha" (* 248.html *)
; "ve.n.tha" (* 249.html *)
; "veza" (* 250.html *)
; "vaimaatra" (* 251.html *)
; "vya~nj" (* 252.html *)
; "vyah" (* 253.html *)
; "vy.r" (* 254.html *)
; "z" (* 255.html *)
; "zata" (* 256.html *)
; "zabd" (* 257.html *)
; "zaraketu" (* 258.html *)
; "zazamaana" (* 259.html *)
; "zaa.mtanava" (* 260.html *)
; "zaaha" (* 261.html *)
; "zivaga.na" (* 262.html *)
; "ziita" (* 263.html *)
; "zu.n.d" (* 264.html *)
; "zuurta" (* 265.html *)
; "zai.siri" (* 266.html *)
; "zyai" (* 267.html *)
```

```
; "zraama" (* 268.html *)
; "zriikajaaka" (* 269.html *)
; "zvabhr" (* 270.html *)
; ".s" (* 271.html *)
; "s" (* 272.html *)
; "sa.mzu.s" (* 273.html *)
; "sa.msthaa" (* 274.html *)
; "sakalakala" (* 275.html *)
; "sa.mgha.t" (* 276.html *)
; "satii" (* 277.html *)
; "satak.san" (* 278.html *)
; "sa.mtap" (* 279.html *)
; "sapak.sa" (* 280.html *)
; "sabhaaj" (* 281.html *)
; "samave" (* 282.html *)
; "samifg" (* 283.html *)
; "sam.r" (* 284.html *)
; "samphe.ta" (* 285.html *)
; "saragh" (* 286.html *)
; "sarva" (* 287.html *)
; "sarvasuuk.sma" (* 288.html *)
; "sazakala" (* 289.html *)
; "sahama" (* 290.html *)
; "saa.mjiiviiputra" (* 291.html *)
; "saamanii" (* 292.html *)
; "saar.sapa" (* 293.html *)
; "sidgu.n.da" (* 294.html *)
; "siila" (* 295.html *)
; "sucakra" (* 296.html *)
; "sund" (* 297.html *)
; "suma" (* 298.html *)
; "sur" (* 299.html *)
; "su.sa.msad" (* 300.html *)
; "suutr" (* 301.html *)
; "setu" (* 302.html *)
; "sodara" (* 303.html *)
; "sora" (* 304.html *)
; "skandha" (* 305.html *)
; "stha" (* 306.html *)
; "snaayu" (* 307.html *)
```

```
  ; "sm.rta" (* 308.html *)
  ; "svasvadha" (* 309.html *)
  ; "svanuruupa" (* 310.html *)
  ; "svaakta" (* 311.html *)
  ; "h" (* 312.html *)
  ; "hari" (* 313.html *)
  ; "hala" (* 314.html *)
  ; "hi.ms" (* 315.html *)
  ; "huu" (* 316.html *)
  ; "ho.dha" (* 317.html *)
  ]
;
value look_up_chap w n =
(* let v = match w with [ 0 (× − × ) :: stem ] → stem | _ → w in *)
  look_up n
  where rec look_up n = fun
  [ [] → n
  | [ frontier :: l ] → if Order.lexico frontier w then look_up (n + 1) l else n
  ]
;
(* Enter in this table associations between a defined form and its defining entry, whenever there is a chapter boundary in between. In a future version this table ought to be mechanically built. *)
value vocable s =
  let entry = fun
    [ "Dyaus" → "div"
    | s → s
    ] in
  Encode.code_skt_ref (entry s)
;
value dico_chapter s = (* defining chapter of Sanskrit word form s *)
  let lower = fun
    [ [ 0 :: w ] → w (* remove initial hyphen of suffixes *)
    | [ c :: w ] → [ (if c > 100 then c − 100 else c) :: w ] (* remove capital *)
    | [] → []
    ] in
  let defining_word = lower (vocable s) in
  look_up_chap defining_word 1 dico_chapters
;
value cypher = string_of_int (* no cyphering so far *)
```

```
;
value dico_page chapter = (* each chapter in its own page *)
  cypher chapter ^ ".html"
;
(* Used in Morpho_html *)
value sh_defining_page s = dico_page (dico_chapter s)
;
value mw_defining_page_exc s mw_exceptions =
  let exc_page = Deco.assoc (Encode.rev_code_string s) mw_exceptions in
  let file_name = match exc_page with
    [ [] → let initial = fun
        [ [ 0 :: w ] → w (* remove initial hyphen of suffixes *)
        | [ c :: w ] → [ (if c > 100 then c − 100 else c) :: w ] (* remove capital *)
        | [] → []
        ] in let defining_word = initial (vocable s) in
              look_up_chap defining_word 1 mw_chapters
    | [ n ] → n
    | _ → failwith "mw_defining_page"
    ] in
  (cypher file_name) ^ ".html"
;
```

end;

Module Morpho_scl

Prints lists of inflected forms in XML for use by external Web services.
Adapted from *Morpho_xml*
Uses WX for transliteration output.

```
open Skt_morph;
open Morphology; (* inflected and its constructors Noun_form, ... *)
open Naming; (* look_up_homo homo_undo unique_kridantas lexical_kridantas preverbs_structure
*)
value ps = print_string
;
value pr_scl_gana k = ps (string_of_int k)
;
value print_scl_number = fun
  [ Singular → ps "<sg/>"
```

```
  | Dual → ps "<du/>"
  | Plural → ps "<pl/>"
  ]
and print_scl_gender = fun
  [ Mas → ps "<m/>"
  | Neu → ps "<n/>"
  | Fem → ps "<f/>"
  | Deictic _ → ps "<d/>"
  ]
and print_scl_case = fun
  [ Nom → ps "<nom/>"
  | Acc → ps "<acc/>"
  | Ins → ps "<ins/>"
  | Dat → ps "<dat/>"
  | Abl → ps "<abl/>"
  | Gen → ps "<gen/>"
  | Loc → ps "<loc/>"
  | Voc → ps "<voc/>"
  ]
and print_scl_person = fun
  [ First → ps "<fst/>"
  | Second → ps "<snd/>"
  | Third → ps "<thd/>"
  ]
and print_scl_voice = fun
  [ Active → ps "<ac/>"
  | Middle → ps "<md/>"
  | Passive → ps "<ps/>"
  ]
and print_scl_pr_mode = fun
  [ Present → ps "<pr␣gana="
  | Imperative → ps "<imp␣gana="
  | Optative → ps "<opt␣gana="
  | Imperfect → ps "<impft␣gana="
  ]
and print_scl_pr_mode_ps = fun
  [ Present → ps "<prps/>"
  | Imperative → ps "<impps/>"
  | Optative → ps "<optps/>"
  | Imperfect → ps "<impftps/>"
```

```
  ]
and print_scl_tense = fun
  [ Future → ps "<fut/>"
  | Perfect → ps "<pft/>"
  | Aorist k → do { ps "<aor␣gana="; pr_scl_gana k; ps "/>" }
  | Injunctive k → do { ps "<inj␣gana="; pr_scl_gana k; ps "/>" }
  | Benedictive → ps "<ben/>"
  | Conditional → ps "<cond/>"
  | Subjunctive → ps "<subj/>"
  ]
;
value print_scl_paradigm = fun
  [ Conjug t v → do { print_scl_tense t; print_scl_voice v }
  | Presenta k pr → do { print_scl_pr_mode pr; pr_scl_gana k;
                          ps "/><ac/>" }
  | Presentm k pr → do { print_scl_pr_mode pr; pr_scl_gana k;
                          ps "/><md/>" }
  | Presentp pr → print_scl_pr_mode_ps pr
  | Perfut v → ps "<perfut/>" (* TODO: mark voice *)
  ]
and print_scl_conjugation = fun
  [ Primary → ()
  | Causative → ps "<ca/>"
  | Intensive → ps "<int/>"
  | Desiderative → ps "<des/>"
  ]
and print_scl_nominal = fun
  [ Ppp → ps "<pp/>"
  | Pppa → ps "<ppa/>"
  | Ppra k → do { ps "<ppr␣gana="; pr_scl_gana k; ps "/>";
                  print_scl_voice Active }
  | Pprm k → do { ps "<ppr␣gana="; pr_scl_gana k; ps "/>";
                  print_scl_voice Middle }
  | Pprp → do { ps "<ppr/>"; print_scl_voice Passive }
  | Ppfta → do { ps "<ppf/>"; print_scl_voice Active }
  | Ppftm → do { ps "<ppf/>"; print_scl_voice Middle }
  | Pfuta → do { ps "<pfu/>"; print_scl_voice Active }
  | Pfutm → do { ps "<pfu/>"; print_scl_voice Middle }
  | Pfutp k → do { ps "<pfp/>"; pr_scl_gana k }
  | _ → ps "<act/>" (* action verbal nouns *)
```

```
  ]
and print_scl_invar = fun
  [ Infi → ps "<inf/>"
  | Absoya → ps "<abs/>"
  | Perpft → ps "<perpft/>"
  ]
and print_scl_kind = fun
  [ Part → ps "<part/>"
  | Prep → ps "<prep/>"
  | Conj → ps "<conj/>"
  | Abs → ps "<abs/>"
  | Adv → ps "<adv/>"
  | _ → ps "<ind/>"
  ]
;
value print_scl_finite (c, p) =
  do { print_scl_conjugation c; print_scl_paradigm p }
and print_scl_verbal (c, n) =
  do { print_scl_conjugation c; print_scl_nominal n }
and print_scl_modal (c, i) =
  do { print_scl_conjugation c; print_scl_invar i }
;
value print_scl_morph = fun
  [ Noun_form g n c
  | Part_form _ g n c → do
      { print_scl_case c
      ; print_scl_number n
      ; print_scl_gender g
      }
  | Bare_stem | Avyayai_form → ps "<iic/>"
  | Verb_form f n p → do
      { print_scl_finite f
      ; print_scl_number n
      ; print_scl_person p
      }
  | Ind_form k → print_scl_kind k
  | Avyayaf_form → ps "<avya/>"
  | Abs_root c → do { print_scl_conjugation c; ps "<abs/>" }
  | Gati → ps "<iiv/>"
  | Ind_verb m → print_scl_modal m
```

```
  | PV _ → ps "<pv/>"
  | Unanalysed → ps "<unknown/>"
  ]
;
value print_scl_morphs =
  let choice () = ps "</choice><choice>" in
  List2.process_list_sep print_scl_morph choice
;
value print_inv_morpho_scl pe form generative (delta, morphs) =
  let stem = Word.patch delta form in do (* stem may have homo index *)
    { ps "<morpho_infl><choice>"
    ; print_scl_morphs morphs
    ; ps "</choice></morpho_infl>"
    ; ps "<morpho_gen>"
    ; if generative then (* interpret stem as unique name *)
          let (homo, bare_stem) = homo_undo stem in
          let krid_infos = Deco.assoc bare_stem unique_kridantas in
          try let (verbal, root) = look_up_homo homo krid_infos in do
          { pe bare_stem
          ; ps "<krid>"; print_scl_verbal verbal
          ; ps "</krid><root>"; pe root; ps "</root>"
          } with [ _ → pe bare_stem ]
        else pe stem
    ; ps "</morpho_gen>"
    }
      ;
value print_scl_entry w = (* ps offline in WX notation for UoH interface *)
  ps ("<entry␣wx=\"" ^ Canon.decode_WX w ^ "\"/>")
;
(* Decomposes a preverb sequence into the list of its components *)
(* Similar to Morpho.decomp_pvs *)
value decomp_pvs pvs =
  Deco.assoc pvs preverbs_structure
;
value print_inv_morpho_scl pvs form =
  let pv = if Phonetics.phantomatic form then [ 2 ] (* aa- *)
           else pvs in
  let encaps e = if pv = [] then print_scl_entry e
                 else let pv_list = decomp_pvs pvs in do
                      { List2.process_list_sep pr_pv (fun () → ps "-") pv_list
```

```
                    where pr_pv pv = Canon.decode_WX pv |> ps
                  ; print_scl_entry e
                  } in
  print_inv_morpho_scl encaps form
;
(* Used in Lexer.print_scl_morph *)
value print_scl_inflected pvs =
  print_inv_morpho_scl pvs
;
```

Module Mk_index_page

This stand-alone program produces the page *indexer_page.html* used as index interface to the Sanskrit Heritage dictionary.

```
open Html;
open Web; (* ps pl abort etc. *)
value deva = (Paths.default_display_font ="deva")
;
value print_query lang cgi = do
  { pl (cgi_begin cgi "convert")
  ; print_lexicon_select (lexicon_of lang)
  ; pl html_break
(* ; ps "Output␣font␣for␣inflexion␣tool" ; pl (hidden_input "font" Paths.default_display_font) ;
) ; ("␣Devanagari","deva",deva) (× default deva − Simputer × ) ]) ; pl html_break
TODO: switch to specific version of dictionaries *)
  ; pl (text_input "focus" "q")
  ; print_transliteration_switch "trans"
  ; pl html_break
  ; pl (submit_input "Search")
  ; pl (reset_input "Reset")
  ; pl cgi_end
  }
;
value print_query_dummy lang cgi = do
  { pl (cgi_begin cgi "convert")
  ; pl (hidden_input "lex" (lexicon_of lang))
  ; pl (text_input "unused" "q")
  ; ps "ASCII"
  ; pl html_break
```

```
  ; pl (submit_input "Search")
  ; pl (reset_input "Reset")
  ; pl cgi_end
  }
;
value print_query_lemma lang cgi = do
  { pl (cgi_begin cgi "convert1")
  ; pl (hidden_input "lex" (lexicon_of lang))
  ; pl (text_input "focus1" "q")
  ; print_transliteration_switch "trans1"
  ; pl html_break
  ; pl (option_select_default "c"
          [ (" Noun ","Noun",True) (* default Noun *)
          ; (" Pron ","Pron",False)
          ; (" Verb ","Verb",False)
          ; (" Part ","Part",False)
          ; (" Inde ","Inde",False)
          ; (" Absya ","Absya",False)
          ; (" Abstvaa ","Abstvaa",False)
          ; (" Voca ","Voca",False)
          ; (" Iic " ,"Iic", False)
          ; (" Ifc " ,"Ifc", False)
          ; (" Iiv " ,"Iiv", False)
          ; (" Piic ","Piic",False)
          ])
  ; pl html_break
  ; pl (submit_input "Search")
  ; pl (reset_input "Reset")
  ; pl cgi_end
  }
;
value indexer lang = do (* Not yet in xhtml validated form *)
  { open_html_file (indexer_page lang) heritage_dictionary_title
  ; pl (body_begin (background Chamois))
     (* will be closed by close_html_file *)
  ; print_title (Some lang) (dico_title lang)
  ; pl center_begin (* closed at the end *)
      (* Sankskit index section *)
  ; print_index_help lang
  ; print_query lang index_cgi
```

```
  ; pl html_paragraph
  ; pl hr
    (* Sankskrit made easy section (Sanskrit for dummies) *)
  ; pl (anchor_def "easy" "")
  ; pl dummy_title_en
  ; print_dummy_help_en ()
  ; print_query_dummy lang dummy_cgi
  ; pl html_paragraph
  ; pl hr
    (* Stemmer section *)
  ; pl stem_title_en
  ; pl (anchor_def "stemmer" "") (* for access from dock link *)
  ; print_stemmer_help_en ()
  ; print_query_lemma lang lemmatizer_cgi
  ; pl html_break
  ; pl center_end
  ; close_html_file lang True
  }
;
indexer French
;
indexer English
;
```

Module Mk_grammar_page

This program produces the page grammar.html (Grammarian interface)

```
open Html;
open Web; (* ps pl abort etc. *)
value title = h1_title "The␣Sanskrit␣Grammarian"
and subtitle_d = h1_title "Declension"
and subtitle_c = h1_title "Conjugation"
and meta_title = title "Sanskrit␣Grammarian␣Query"
;
value deva = (Paths.default_display_font ="deva")
;
value print_declension_help lang =
  if narrow_screen then () else do
    { ps (par_begin G2)
```

```
    ; ps "Submit␣stem␣and␣gender␣for␣declension:"
    ; pl html_break
    ; ps "(Use␣Any␣for␣deictic␣pronouns␣and␣numbers)"
    ; pl par_end (* G2 *)
    }
;
value print_conjugation_help lang =
  if narrow_screen then () else do
    { ps (par_begin G2)
    ; ps "Submit␣root␣and␣present␣class"
    ; pl html_break
    ; ps "(Use␣0␣for␣secondary␣conjugations)"
    ; pl par_end (* G2 *)
    }
;
value print_output_font () = do
  { pl html_break
  ; ps "Output␣font␣"
  ; pl (option_select_default "font"
        [ ("␣Roman","roma",¬ deva) (* default roma - Computer *)
        ; ("␣Devanagari","deva",deva) (* default deva - Simputer *)
        ])
  ; pl html_break
  ; pl (submit_input "Send")
  ; pl (reset_input "Reset")
  ; pl cgi_end
  }
;
value grammarian lang = do
  { open_html_file (grammar_page lang) meta_title
  ; pl (body_begin (background Chamois))
  ; print_title (Some lang) title
  ; pl center_begin
  ; pl subtitle_d
  ; print_declension_help lang
  ; pl (cgi_begin decls_cgi "convert")
  ; pl (hidden_input "lex" (lexicon_of lang))
(* pl (hidden_input "v" (Install.stamp)) OBS *)
  ; pl (text_input "focus" "q")
  ; print_transliteration_switch "trans"
```

```
  ; pl html_break
  ; ps "Gender␣"
  ; pl (option_select_default "g"
          [ ("␣Mas␣","Mas",True) (* default Mas *)
          ; ("␣Fem␣","Fem",False)
          ; ("␣Neu␣","Neu",False)
          ; ("␣Any␣","Any",False) (* deictic pronouns and numbers *)
          ])
  ; print_output_font ()
  ; pl html_break
  ; pl subtitle_c
  ; pl (xml_empty_with_att "a" [ ("name","roots") ]) (* for portal ref *)
  ; print_conjugation_help lang
  ; pl (cgi_begin conjs_cgi "convert1")
  ; pl (hidden_input "lex" (lexicon_of lang))
(* pl (hidden_input "v" (Install.stamp)) OBS *)
  ; pl (text_input "focus1" "q")
  ; print_transliteration_switch "trans1"
  ; pl html_break
  ; ps "Present␣class␣"
  ; pl (option_select_default "c" (* gana = present class *)
          [ ("␣1␣", "1", True) (* default 1 *)
          ; ("␣2␣", "2", False)
          ; ("␣3␣", "3", False)
          ; ("␣4␣", "4", False)
          ; ("␣5␣", "5", False)
          ; ("␣6␣", "6", False)
          ; ("␣7␣", "7", False)
          ; ("␣8␣", "8", False)
          ; ("␣9␣", "9", False)
          ; ("␣10", "10", False)
          ; ("␣11", "11", False) (* denominative verbs *)
          ; ("␣0", "0", False) (* secondary conjugations *)
          ])
  ; print_output_font ()
  ; pl center_end
  ; close_html_file lang True
  }
;
grammarian French
```

```
;
grammarian English
;
```

Module Mk_reader_page

This program creates the page *reader_page* (Sanskrit Reader Interface) invoking the CGI sktreader alias reader. Invoked without language argument, it is itself the CGI *skt_heritage* invokable separately.

```
open Html;
open Web; (* ps pl abort etc. *)
open Cgi; (* create_env get *)
value back_ground = background Chamois
;
value out_mode = ref None
;
value set_cho () = Arg.parse
  [ ("-fr", Arg.Unit (fun () → out_mode.val := Some French), "French")
  ; ("-en", Arg.Unit (fun () → out_mode.val := Some English), "English")
  ; ("", Arg.Unit (fun () → out_mode.val := None), "default␣language␣for␣cgi")
  ]
  (fun s → raise (Arg.Bad s))
  "Usage:␣mk_reader_page␣-en␣or␣mk_reader_page␣-fr␣or␣mk_reader_page"
;
value print_cache_policy cache_active = do
  { ps "␣Cache␣"
  ; let options =
      [ ("␣On␣","t",cache_active ="t") (* Cache active *)
      ; ("␣Off␣","f",cache_active ="f") (* Ignore cache *)
      ] in
    pl (option_select_default "cache" options)
  }
;
value reader_input_area_default =
  text_area "text" 1 screen_char_length
;
value reader_input_area = reader_input_area_default ""
;
value reader_page () = do
```

```
{ set_cho ()
; let (lang, query) = match out_mode.val with
    [ Some lang → do
      { open_html_file (reader_page lang) reader_meta_title; (lang,"") }
    | None → do
      { reader_prelude ""; (default_language, Sys.getenv "QUERY_STRING") }
    ] in try
  let env = create_env query in
  let url_encoded_input = get "text" env ""
  and url_encoded_mode = get "mode" env "g"
  and url_encoded_topic = get "topic" env ""
  and st = get "st" env "t" (* default vaakya rather than isolated pada *)
  and cp = get "cp" env default_mode
  and us = get "us" env "f" (* default input sandhied *)
  and cache_active = get "cache" env cache_active.val
  and translit = get "t" env Paths.default_transliteration in
  (* Contextual information from past discourse *)
  let topic_mark = decode_url url_encoded_topic
  and text = decode_url url_encoded_input in
```

Corpus parameters

```
  let corpus_permission = Cgi.decoded_get Params.corpus_permission "" env in
  let corpus_dir = Cgi.decoded_get Params.corpus_dir "" env in
  let sentence_no = Cgi.decoded_get Params.sentence_no "" env in do
{ pl (body_begin back_ground)
; print_title (Some lang) reader_title
; h3_begin C3 |> pl
; if Web_corpus.(permission_of_string corpus_permission = Annotator) then
    "Corpus annotator permission - " ^ corpus_dir |> pl
  else
    ()
; h3_end |> pl
; pl center_begin
; pl (cgi_reader_begin reader_cgi "convert")
; print_lexicon_select (lexicon_of lang)
; if cache_allowed then print_cache_policy cache_active else ()
; pl html_break
; pl "Text "
; pl (option_select_default "st"
      [ (" Sentence ","t",st ="t")
```

```
          ; ("␣␣␣Word␣␣␣","f",st ="f")
          ])
  ; pl "␣Format␣"
  ; pl (option_select_default "us"
          [ ("␣Unsandhied␣","t",us ="t")
          ; ("␣␣Sandhied␣␣","f",us ="f")
          ])
  ; pl "␣Parser␣strength␣"
  ; pl (option_select_default "cp"
          [ ("␣␣Full␣␣","t",cp ="t")
          ; ("␣Simple␣","f",cp ="f")
          ])
  ; pl html_break
  ; ps (reader_input_area_default text)
  ; pl html_break
  ; ps "Input␣convention␣"
  ; ps (transliteration_switch_default translit "trans")
  ; pl "␣Optional␣topic␣" (* For the moment assumed singular *)
  ; pl (option_select_default "topic"
          [ ("␣Masculine␣","m",topic_mark ="m")
          ; ("␣Feminine␣␣","f",topic_mark ="f")
          ; ("␣␣Neuter␣␣␣","n",topic_mark ="n")
          ; ("␣␣␣Void␣␣␣␣","" ,topic_mark ="")
          ])
  ; pl "␣Mode␣"
  ; pl (option_select_default_id "mode_id" "mode"
          (interaction_modes_default url_encoded_mode))
```

Corpus parameters

```
  ; hidden_input Params.corpus_permission corpus_permission |> pl
  ; hidden_input Params.corpus_dir corpus_dir |> pl
  ; hidden_input Params.sentence_no sentence_no |> pl

  ; pl html_break
  ; pl (submit_input "Read")
  ; pl (reset_input "Reset")
  ; pl cgi_end
  ; pl center_end
  ; match out_mode.val with
    [ Some lang → close_html_file lang True
    | None →
```

```
      do { close_page_with_margin (); page_end default_language True }
    ]
  }
    with
    [ Sys_error s → abort lang Control.sys_err_mess s (* file pb *)
    | Stream.Error s → abort lang Control.stream_err_mess s (* file pb *)
    | Exit (* Sanskrit *) → abort lang "Wrong character in input" ""
    | Invalid_argument s → abort lang Control.fatal_err_mess s (* sub *)
    | Failure s → abort lang Control.fatal_err_mess s (* anomaly *)
    | End_of_file → abort lang Control.fatal_err_mess "EOF" (* EOF *)
    | Not_found → let s = "You must choose a parsing option" in
                          abort lang "Unset button in form - " s
    | Control.Fatal s → abort lang Control.fatal_err_mess s (* anomaly *)
    | Control.Anomaly s → abort lang Control.fatal_err_mess ("Anomaly: " ^ s)
    | _ → abort lang Control.fatal_err_mess "Unexpected anomaly"
    ]
 }
;
reader_page ()
;
```

Module Mk_sandhi_page

This stand-alone program produces the page *sandhi_page.html* used as sandhi computation interface to the Sandhi Engine.

```
open Html;
open Web; (* ps pl abort etc. *)
value title = h1_title "The Sandhi Engine"
and meta_title = title "Sanskrit Sandhi Engine"
and back_ground = background Chamois
  (* obs if narrow_screen then background Chamois else Pict_geo *)
;
value sandhier lang = do
  { open_html_file (sandhi_page lang) meta_title
  ; pl (body_begin back_ground)
  ; print_title None title
  ; pl center_begin
  ; pl (cgi_begin sandhier_cgi "convert2")
```

```
    (* following necessary to transmit the lexicon choice of the session *)
  ; pl (hidden_input "lex" (lexicon_of lang))
  ; pl (text_input "focus1" "l")
  ; pl (text_input "focus2" "r")
  ; print_transliteration_switch "trans"
  ; pl html_break
  ; pl (option_select_default "k"
        [ (" External ","external",True) (* default external *)
        ; (" Internal ","internal",False)
        ])
  ; pl html_break
  ; pl (submit_input "Send")
  ; pl (reset_input "Reset")
  ; pl cgi_end
  ; pl html_break
  ; pl center_end
  ; close_html_file lang True
  }
;
sandhier French
;
sandhier English
;
```

Module Mk_corpus_page

This program produces the pages corpus.html (Corpus interface).

```
open Html;
open Web;

value permission_selection =
  let selection permissions =
      List.map select permissions
      where select permission =
        let permission_str = Web_corpus.string_of_permission permission in
        (String.capitalize_ascii permission_str, permission_str,
         permission = Web_corpus.Reader) in
  let read_only_permissions = [ Web_corpus.Reader ] in
  let other_permissions = Web_corpus.[ Annotator; Manager ] in
  let all_permissions = read_only_permissions @ other_permissions in
```

```
  selection (if corpus_read_only then read_only_permissions else all_permissions)
;
value make lang =
  let title_str = "Sanskrit␣Corpus" in do
  { open_html_file (corpus_page lang) (title title_str)
  ; body_begin Chamois_back |> pl
  ; open_page_with_margin 15
  ; h1_title title_str |> print_title (Some lang)
  ; center_begin |> pl
  ; cgi_begin corpus_manager_cgi "" ^
    "Capacity:␣" ^
    option_select_default Params.corpus_permission permission_selection ^ "␣" ^
    submit_input "OK" ^
    cgi_end |> pl
  ; center_end |> pl
  ; close_page_with_margin ()
  ; close_html_file lang True
  }
;
value main = do
  { make English
  ; make French
  }
;
```

Interface for module Corpus

Operations on the corpus tree

```
module Section : sig
  type t
  ;
  value label : t → string
  ;
end
;
module Analyzer : sig
  type t = [ Graph ]
  ;
  value path : t → string
```

```
  ;
  value relocatable_path : t → string
  ;
end
;
module Analysis : sig
  type t
  ;
  value make : Analyzer.t → Html.language → string → int (* Num.num *) → t
  ;
  value analyzer : t → Analyzer.t
  ;
  value lang : t → Html.language
  ;
  value checkpoints : t → string
  ;
  value nb_sols : t → int (* Num.num *)
  ;
end
;
module Encoding : sig
  type t = [ Velthuis | WX | KH | SLP1 | Devanagari | IAST ]
  ;
  value to_string : t → string
  ;
  value of_string : string → t
  ;
  value encode : t → string → Word.word
  ;
  value decode : t → Word.word → string
  ;
end
;
module Sentence : sig
  type t
  ;
  value make : int → list Word.word → bool → Analysis.t → t
  ;
  value id : t → int
  ;
```

```
  value text : Encoding.t → t → string
  ;
  value analysis : t → Analysis.t
  ;
end
;
module type Location = sig
  value path : string
  ;
end
;
module type S = sig
  (* Contents of a corpus subdirectory: either it is empty (constructor Empty), otherwise
we are on leaves of the tree (constructor Sentences) or on branches (constructor Sections).
*)
  type contents =
    [ Empty
    | Sections of list Section.t
    | Sentences of list Sentence.t
    ]
  ;
  (* List the contents of the given corpus subdirectory. Note that the returned elements
are sorted according to Section.compare or Sentence.compare depending on the case. Raise
Sys_error when an operating system error occurs. *)
  value contents : string → contents
  ;
  (* Exception raised by save_sentence when the sentence to be saved already exists. *)
  exception Sentence_already_exists
  ;
  (* Raise Sentence_already_exists if the sentence to be saved already exists and force is
False and Sys_error when an operating system error occurs. *)
  value save_sentence :
    bool → string → int → list Word.word → bool → Analysis.t → unit
  ;
  exception Section_already_exists of string
  ;
  (* Raise Section_already_exists if the given corpus directory already exists and Unix.Unix_error
when an operating system error occurs. *)
  value mkdir : string → unit
  ;
```

```
  exception No_such_sentence of int
  ;
  (* Raise No_such_sentence i if the sentence i does not exist. *)
  value sentence : string → int → Sentence.t
  ;
  type permission = [ Reader | Annotator | Manager ]
  ;
  value default_permission : permission
  ;
  value string_of_permission : permission → string
  ;
  value permission_of_string : string → permission
  ;
  value url : string → permission → Sentence.t → string
  ;
  value relocatable_url : string → permission → Sentence.t → string
  ;
  (* citation subdir id ] returns an URL to the analysis of the sentence whose number is id
in the corpus subdirectory subdir. Raise Failure "citation" if an error occurs. *)
  value citation : string → int → string
  ;
end
;
module Make (Loc : Location) : S
;
```

Module Corpus

```
module Section : sig
  type t
  ;
  value make : string → t
  ;
  value label : t → string
  ;
  value compare : t → t → int
  ;
end = struct
  type t = string
```

```
  ;
  value make h = h
  ;
  value label h = h
  ;
  value compare h h' = String.compare (label h) (label h')
  ;
end
;
module Analyzer : sig
  type t = [ Graph ]
  ;
  value path : t → string
  ;
  value relocatable_path : t → string
  ;
end = struct
  type t = [ Graph ]
  ;
  value path = fun [ Graph → Paths.(cgi_dir_url ^ cgi_graph) ]
  and relocatable_path = fun [ Graph → "!CGIGRAPH" ]
  ;
end
;
module Analysis : sig
  type t
  ;
  value make :
    Analyzer.t → Html.language → string → int (* Num.num *) → t
  ;
  value analyzer : t → Analyzer.t
  ;
  value lang : t → Html.language
  ;
  value checkpoints : t → string
  ;
  value nb_sols : t → int (* Num.num *)
  ;
end = struct
  type t =
```

```
    { analyzer : Analyzer.t
    ; lang : Html.language
    ; checkpoints : string
    ; nb_sols : int (* Num.num *)
    }
  ;
  value make analyzer lang checkpoints nb_sols =
    { analyzer ; lang; checkpoints; nb_sols }
  ;
  value analyzer a = a.analyzer
  ;
  value lang a = a.lang
  ;
  value checkpoints a = a.checkpoints
  ;
  value nb_sols a = a.nb_sols
  ;
end
;
module Encoding : sig
  type t = [ Velthuis | WX | KH | SLP1 | Devanagari | IAST ]
  ;
  value to_string : t → string
  ;
  value of_string : string → t
  ;
  value encode : t → string → Word.word
  ;
  value decode : t → Word.word → string
  ;
end = struct
  type t = [ Velthuis | WX | KH | SLP1 | Devanagari | IAST ]
  ;
  value to_string = fun
    [ Velthuis → "VH"
    | WX → "WX"
    | KH → "KH"
    | SLP1 → "SL"
    | Devanagari → "deva"
    | IAST → "roma"
```

```
    ]
  ;
  value rec of_string = fun
    [ "VH" → Velthuis
    | "WX" → WX
    | "KH" → KH
    | "SL" → SLP1
    | "deva" → Devanagari
    | "roma" → IAST
    | _ → Velthuis
    ]
  ;
  value encode encoding = encoding |> to_string |> Encode.switch_code
  ;
  value decode = fun
  [ Velthuis | WX | KH | SLP1 as encoding →
      encoding |> to_string |> Canon.switch_decode
  | Devanagari → Canon.unidevcode
  | IAST → Canon.uniromcode
  ]
  ;
end
;
(* What about metadata (date, author, history...) ? *)
module Sentence : sig
  type t
  ;
  value make : int → list Word.word → bool → Analysis.t → t
  ;
  value id : t → int
  ;
  value text : Encoding.t → t → string
  ;
  value analysis : t → Analysis.t
  ;
  value compare : t → t → int
  ;
end = struct
  type t =
    { id : int
```

```
    ; text : list Word.word
    ; unsandhied : bool
    ; analysis : Analysis.t
    }
  ;
  value make id text unsandhied analysis =
    { id = id
    ; text = text
    ; unsandhied = unsandhied
    ; analysis = analysis
    }
  ;
  value id s = s.id
  ;
  value text encoding s =
    s.text |> List.map (Encoding.decode encoding) |> String.concat "␣"
  ;
  value unsandhied s = s.unsandhied
  ;
  value analysis s = s.analysis
  ;
  value compare s s′ = compare (id s) (id s′)
  ;
end
;
module type Location = sig
  value path : string
  ;
end
;
module type S = sig
  (* Contents of a corpus subdirectory: either we are on leaves of the tree (constructor
Sentences) or on branches (constructor Sections). *)
  type contents =
    [ Empty
    | Sections of list Section.t
    | Sentences of list Sentence.t
    ]
  ;
  (* List the contents of the given corpus subdirectory. Note that the returned elements are
```

```
sorted according to Section.compare or Sentence.compare depending on the case. *)
  value contents : string → contents
  ;
  exception Sentence_already_exists
  ;
  value save_sentence :
    bool → string → int → list Word.word → bool → Analysis.t → unit
  ;
  exception Section_already_exists of string
  ;
  value mkdir : string → unit
  ;
  exception No_such_sentence of int
  ;
  value sentence : string → int → Sentence.t
  ;
  type permission = [ Reader | Annotator | Manager ]
  ;
  value default_permission : permission
  ;
  value string_of_permission : permission → string
  ;
  value permission_of_string : string → permission
  ;
  value url : string → permission → Sentence.t → string
  ;
  value relocatable_url : string → permission → Sentence.t → string
  ;
  value citation : string → int → string
  ;
end
;
module Make (Loc : Location) : S = struct
  type contents =
    [ Empty
    | Sections of list Section.t
    | Sentences of list Sentence.t
    ]
  ;
  value ( /^ ) = Filename.concat
```

```
;
value ( ~/ ) file = Loc.path /^ file
;
value sentence_ext = "rem"
;
value sentence_file subdir id =
  ~/subdir /^ Printf.sprintf "%d.%s" id sentence_ext
;
exception No_such_sentence of int
;
value sentence subdir id =
  let file = sentence_file subdir id in
  if Sys.file_exists file then (Gen.gobble file : Sentence.t)
                          else raise (No_such_sentence id)
;
value contents subdir =
  let subdir = ~/subdir in
  match Dir.subdirs subdir with
  [ [] →
    let sentences =
      subdir
      —> Dir.files_with_ext sentence_ext
      —> List.map (fun x → (Gen.gobble (subdir /^ x) : Sentence.t))
      —> List.sort Sentence.compare
    in
    match sentences with [ [] → Empty | sentences → Sentences sentences ]
  | subdirs →
    let sections =
      subdirs
      —> List.map Section.make
      —> List.sort Section.compare
    in
    Sections sections
  ]
;
value metadata_file dir id = ~/dir /^ "." ^ string_of_int id
;
exception Sentence_already_exists
;
value save_sentence force dir id text unsandhied analysis =
```

```
  let file = sentence_file dir id in
  let sentence = Sentence.make id text unsandhied analysis in
  if ¬ force ∧ Sys.file_exists file then raise Sentence_already_exists
                                    else Gen.dump sentence file
;
exception Section_already_exists of string
;
value mkdir dirname =
  try Unix.mkdir ˜/dirname 755₈ with
  [ Unix.Unix_error (Unix.EEXIST, _, _) →
    raise (Section_already_exists (Filename.basename dirname))
  ]
;
type permission = [ Reader | Annotator | Manager ]
;
value default_permission = Reader
;
value string_of_permission = fun
  [ Reader → "reader"
  | Annotator → "annotator"
  | Manager → "manager"
  ]
;
value restrict_permission perm =
  match Html.target with
  [ Html.Server → Reader
  | Html.Simputer | Html.Computer | Html.Station → perm
  ]
;
value permission_of_string s = s |> to_perm |> restrict_permission
  where to_perm = fun
    [ "annotator" → Annotator
    | "manager" → Manager
    | _ → Reader
    ]
;
value url dir permission sentence =
  let analysis = Sentence.analysis sentence in
  let encoding = Encoding.of_string Paths.default_transliteration in
  let env =
```

```
        [ (Params.corpus_permission, string_of_permission permission)
        ; ("text", Sentence.text encoding sentence)
        ; ("cpts", Analysis.checkpoints analysis)
        ; (Params.corpus_dir, dir)
        ; (Params.sentence_no, sentence |> Sentence.id |> string_of_int)
        ]
      in
      let path =
        analysis
        —> Analysis.analyzer
        —> Analyzer.path
      in
      Cgi.url path ~query : (Cgi.query_of_env env)
    ;
    value relocatable_url dir permission sentence =
      let analysis = Sentence.analysis sentence in
      let env =
        [ (Params.corpus_permission, string_of_permission permission)
        ; ("text", Sentence.text Encoding.Velthuis sentence)
        ; ("t", Encoding.(to_string Velthuis))
        ; ("cpts", Analysis.checkpoints analysis)
        ; (Params.corpus_dir, dir)
        ; (Params.sentence_no, sentence |> Sentence.id |> string_of_int)
        ]
      in
      let path =
        analysis
        —> Analysis.analyzer
        —> Analyzer.relocatable_path
      in
      Cgi.url path ~query : (Cgi.query_of_env env)
    ;
value citation subdir id =
   relocatable_url subdir Reader (sentence subdir id)
;
end
;
```

Interface for module Web_corpus

include *Corpus.S*
;

Module Web_corpus

include *Corpus.Make* (struct *value path* = *Paths.public_skt_dir* ^ "CORPUS"; end)
;

Interface for module Corpus_manager

Generation of corpus manager's pages
Generate the page displaying a view of the given corpus subdirectory. The output channel is as always either *stdout* for CGI output or a static HTML file (according to the "magic switch" *Web.output_channel*). NB: No error handling is done by this function.

value mk_page : *string* → *Web_corpus.permission* → *unit*
;

Module Corpus_manager

open *Html*;
open *Web*;

Utilities
Type representing interval of missing integers in a sorted list.

type *gap* = { *start* : *int*; *stop* : *int* }
;
(∗ The following functions assume that the given list is sorted in increasing order and represents a subset of positive integers. In particular, the lowest bound of a gap is at least 1 and the greatest at most *max_int*). We call "group" a list of consecutive integers. ∗)

```
value max_gap = { start = 1; stop = max_int }
;
value string_of_gap gap =
  if gap.stop = max_int then
    Printf.sprintf ">␣%d" (gap.start - 1)
  else
    Printf.sprintf "%d␣-␣%d" gap.start gap.stop
;
(* Return a triple (g, gap, rest) where g is the first group of the given list, gap the gap to
the next group and rest the given list without its first group. *)
value rec first_group = fun
  [ [ x :: ([ y :: _ ] as t) ] →
    if y = x + 1 then
      let (group, gap, rest) = first_group t in
      ([ x :: group ], gap, rest)
    else
      ([ x ], { start = x + 1; stop = y - 1 }, t)
  | [] → ([], max_gap, [])
  | [ x ] as l →
    (l, { start = x + 1; stop = max_int }, [])
  ]
;
value groups_with_gaps l =
  let rec aux l =
    let (group, gap, rest) = first_group l in
    let group_gap = (group, gap) in
    match rest with
    [ [] → [ group_gap ]
    | _ → [ group_gap :: aux rest ]
    ]
  in
  aux l
;
value add_init_gap groups =
  let init_gap = fun
    [ [ ([ x :: _ ], _) :: _ ] →
      if x ≠ 1 then Some { start = 1; stop = x - 1 } else None
    | _ → None
    ]
  in
```

```
  match init_gap groups with
  [ None → groups
  | Some gap → [ ([], gap) :: groups ]
  ]
;
(*****************)
(* Page generation *)
(*****************)
value big text = div Latin16 text
;
value link permission dir =
  let url =
    let query =
      Cgi.query_of_env
        [ (Params.corpus_dir, dir)
        ; (Params.corpus_permission, Web_corpus.string_of_permission permission)
        ]
    in
    Cgi.url corpus_manager_cgi ~query |> escape
  in
  let label = Filename.basename dir in
  anchor_ref url label
;
value uplinks dir permission =
  let aux dir =
   let updirs = Dir.split dir in
   let updirs =
     List.mapi (fun i x →
         String.concat Filename.dir_sep (List2.take_prefix (i + 1) updirs)
       ) updirs
   in
   List.map (link permission) updirs
  in
  let uplinks_str =
    dir
    —> aux
    —> String.concat "␣/␣"
  in
  let final_sep = if uplinks_str ≠ "" then "␣/␣" else "" in
  uplinks_str ^ final_sep
```

```
;
(* Display sentences with format "sentence␣||␣sentno" like in citations file. *)
value sentence_links dir permission sentences =
  let to_anchor_ref sentence =
    let font = Multilingual.font_of_string Paths.default_display_font in
    let encoding =
        match font with
        [ Multilingual.Deva → Corpus.Encoding.Devanagari
        | Multilingual.Roma → Corpus.Encoding.IAST
        ] in
    let text = Corpus.Sentence.text encoding sentence in
    let display =
      match font with
      [ Multilingual.Deva → deva16_blue
      | Multilingual.Roma → span Trans16
      ] in
    text
    —> anchor_ref (sentence |> Web_corpus.url dir permission |> escape)
    —> display in
  List.map to_anchor_ref sentences
;
value section_selection dir sections =
  let options =
    let prefixes =
      List.map (fun x → Filename.concat dir x) sections in
    List.combine prefixes sections in
  option_select_label Params.corpus_dir options
;
value add_sentence_form dir permission gap =
  cgi_begin (cgi_bin "skt_heritage") "" ^
  "Add␣sentence:␣" ^ uplinks dir permission ^
  hidden_input Params.corpus_dir dir ^
  hidden_input Params.corpus_permission (Web_corpus.string_of_permission permission) ^
  int_input Params.sentence_no
    ~step : 1
    ~min : gap.start
    ~max : gap.stop
    ~val:gap.start
    ~id : Params.sentence_no ^ "␣" ^
  submit_input "Add"
```

```
  ^
  cgi_end
  ;
value htmlify_group dir permission (group, gap) =
  let (ol, group_id) =
    match group with
    [ [] → ("", "")
    | [ h :: _ ] →
      let id = Corpus.Sentence.id h in
      let group_id = string_of_int id in
      (ol ~li_id_prefix :"" ~start : id (sentence_links dir permission group),
       group_id)
    ]
  in
  let div_id = "group" ^ group_id in
  let add_sentence_form =
    button
      ~id :"add_sentence"
      ~onclick : { js_funid = "hideShowElement" ; js_funargs = [ div_id ] }
      (string_of_gap gap) ^
    elt_begin_attrs [ ("id", div_id) ] "div" Hidden_ ^
    html_paragraph ^
    add_sentence_form dir permission gap ^
    div_end
  in
  ol ^ if permission = Web_corpus.Annotator then add_sentence_form else ""
;
value group_sentences dir sentences =
  let ids = List.map Corpus.Sentence.id sentences in
  let dict = List.combine ids sentences in
  let groups = ids |> groups_with_gaps |> add_init_gap in
  List.map (fun (x, y) → (List.map (fun x → List.assoc x dict) x, y)) groups
;
value new_section_form dir permission =
  cgi_begin mkdir_corpus_cgi "" ^
  "New␣section:␣" ^ uplinks dir permission ^
  hidden_input Mkdir_corpus_params.parent_dir dir ^
  hidden_input Mkdir_corpus_params.permission (Web_corpus.string_of_permission permission) ^
  text_input "new_section" Mkdir_corpus_params.dirname ^ "␣" ^
  submit_input "Create"
```

```
  ^
  cgi_end
  ;
value section_selection_form dir permission sections =
  let selection_prompt =
    let submit_button_label = Web_corpus.(
      match permission with
      [ Reader → "Read"
      | Annotator → "Annotate"
      | Manager → "Manage"
      ]
    )
    in
    uplinks dir permission ^
    section_selection dir (List.map Corpus.Section.label sections) ^ "␣" ^
    submit_input submit_button_label
  in
  cgi_begin corpus_manager_cgi "" ^
  big (
    selection_prompt ^
    hidden_input Params.corpus_permission (Web_corpus.string_of_permission permission)
  ) ^
  cgi_end
;
value body dir permission =
  match Web_corpus.contents dir with
  [ Web_corpus.Empty →
    do
    { uplinks dir permission |> big |> pl
    ; open_page_with_margin 30
    ; match permission with
        [ Web_corpus.Reader → "Empty␣corpus"
        | Web_corpus.Annotator → add_sentence_form dir permission max_gap
        | Web_corpus.Manager → new_section_form dir permission
        ]
      —> pl
    ; close_page_with_margin ()
    }
  | Web_corpus.Sentences sentences →
    let groups = group_sentences dir sentences in
```

```
      do
      { uplinks dir permission |> big |> pl
      ; open_page_with_margin 30
      ; if permission = Web_corpus.Manager then
          "No␣action␣available." —> pl
        else
          groups |> List.map (htmlify_group dir permission) |> List.iter pl
      ; close_page_with_margin ()
      }
  | Web_corpus.Sections sections →
      do
      { center_begin |> pl
      ; section_selection_form dir permission sections |> pl
      ; html_break |> pl
      ; if permission = Web_corpus.Manager then
          new_section_form dir permission |> pl
        else ()
      ; center_end |> pl
      }
  ]
;
value mk_page dir permission =
  let title_str =
    "Sanskrit␣Corpus␣" ^
    (permission |> Web_corpus.string_of_permission |> String.capitalize_ascii)
  in
  let clickable_title =
    let query =
      Cgi.query_of_env [ (Params.corpus_permission, Web_corpus.string_of_permission permission)
    in
    title_str
    —> anchor_ref (Cgi.url corpus_manager_cgi ~query)
    —> h1_title
  in
  do
  { maybe_http_header ()
  ; page_begin (title title_str)
  ; body_begin Chamois_back |> pl
  ; open_page_with_margin 15
  ; clickable_title |> print_title (Some default_language)
```

```
  ; body dir permission
  ; close_page_with_margin ()
  ; page_end default_language True
  }
;
```

Module Corpus_manager_cgi

CGI script *manager* for corpus management, i.e. for listing and adding sentences of the corpus.

```
value main =
  let env = Cgi.create_env (Cgi.query_string ()) in
  let corpdir = Cgi.decoded_get Params.corpus_dir "" env
  and corpperm = Cgi.decoded_get Params.corpus_permission "" env in
  let permission = Web_corpus.permission_of_string corpperm in
  let lang = Html.default_language in
  try
    Corpus_manager.mk_page corpdir permission
  with
  [ Sys_error msg → Web.abort lang Control.sys_err_mess msg
  | _ → Web.abort lang Control.fatal_err_mess "Unexpected anomaly"
  ]
;
```

Interface for module Save_corpus_params

```
value state : string
;
value force : string
;
value nb_sols : string
;
```

Module Save_corpus_params

```
value state = "state"
;
value force = "force"
```

```
;
value nb_sols = "nbsols"
;
```

Module Save_corpus_cgi

CGI script *save_corpus* for saving a sentence into the corpus.

```
open Html;
open Web;

value confirmation_page query =
  let title_str = "Sanskrit Corpus" in
  let env = Cgi.create_env query in
  let corpdir = Cgi.decoded_get Params.corpus_dir "" env in
  let corppermission = Cgi.decoded_get Params.corpus_permission "" env in
  let sentno = Cgi.decoded_get Params.sentence_no "" env in
  let confirmation_msg =
    Printf.sprintf "Confirm changes for sentence no. %s of %s ?" sentno corpdir
  in
  let specific_url path = Cgi.url path ~fragment : sentno in
  do
  { maybe_http_header ()
  ; page_begin (title title_str)
  ; body_begin Chamois_back |> pl
  ; open_page_with_margin 15
  ; h1_title title_str |> print_title (Some default_language)
  ; center_begin |> pl
  ; div Latin16 confirmation_msg |> pl
  ; html_break |> pl
  ; cgi_begin (specific_url save_corpus_cgi) "" —> pl
  ; hidden_input Save_corpus_params.state (escape query) |> pl
  ; hidden_input Save_corpus_params.force (string_of_bool True) |> pl
  ; submit_input "Yes" —> pl
  ; cgi_end |> pl
  ; html_break |> pl
  ; cgi_begin (specific_url corpus_manager_cgi) "" —> pl
  ; hidden_input Params.corpus_dir corpdir |> pl
  ; hidden_input Params.corpus_permission corppermission |> pl
  ; submit_input "No" —> pl
  ; cgi_end |> pl
```

```
  ; center_end |> pl
  ; close_page_with_margin ()
  ; page_end default_language True
  }
;
value analysis_of_env env =
  let lang =
    env
    —> Cgi.decoded_get "lex" Paths.default_lexicon
    —> Html.language_of
  in
  let cpts =
    env
    —> Cgi.decoded_get "cpts" ""
    (* —> Checkpoints.parse_cpts *)
  in
  let nb_sols =
    env
    —> Cgi.decoded_get Save_corpus_params.nb_sols "0"
    —> int_of_string
  in
  Corpus.Analysis.make Corpus.Analyzer.Graph lang cpts nb_sols
;
value error_page = error_page "Corpus␣Manager"
;
(**************)
(* Entry point *)
(**************)
value main =
  let query = Cgi.query_string () in
  let env = Cgi.create_env query in
  let query = Cgi.decoded_get Save_corpus_params.state "" env in
  try
    let force =
      env
      —> Cgi.decoded_get Save_corpus_params.force (string_of_bool False)
      —> bool_of_string
    in
    let env = Cgi.create_env query in
    let corpdir = Cgi.decoded_get Params.corpus_dir "" env in
```

```
    let sentno =
      env
      —> Cgi.decoded_get Params.sentence_no ""
      —> float_of_string
      —> int_of_float
    in
    let text = Cgi.decoded_get "text" "" env in
    let unsandhied = Cgi.decoded_get "us" "f" env = "t" in
    let permission =
      Web_corpus.permission_of_string (Cgi.decoded_get Params.corpus_permission "" env)
    in
    match permission with
    [ Web_corpus.Annotator →
      let read_skt =
        if unsandhied then Sanskrit.read_raw_sanskrit else
          Sanskrit.read_sanskrit
      in
      let encode =
        Cgi.decoded_get "t" Paths.default_transliteration env
        —> Corpus.Encoding.of_string
        —> Corpus.Encoding.encode
      in
      do
      { Web_corpus.save_sentence force corpdir sentno
          (read_skt encode text) unsandhied (analysis_of_env env)
      ; Corpus_manager.mk_page corpdir permission
      }
    | Web_corpus.Reader | Web_corpus.Manager →
      let expected_permission = Web_corpus.(string_of_permission Annotator) in
      let current_permission = Web_corpus.string_of_permission permission in
      invalid_corpus_permission_page expected_permission current_permission
    ]
  with
  [ Web_corpus.Sentence_already_exists → confirmation_page query
  | Sys_error msg → error_page Control.sys_err_mess msg
  | Failure msg → error_page Control.fatal_err_mess msg
  | _ → abort default_language Control.fatal_err_mess "Unexpected␣anomaly"
  ]
;
```

Interface for module Mkdir_corpus_params

```
value dirname : string
;
value parent_dir : string
;
value permission : string
;
```

Module Mkdir_corpus_params

```
value dirname = "dirname"
;
value parent_dir = Params.corpus_dir
;
value permission = Params.corpus_permission
;
```

Module Mkdir_corpus_cgi

CGI script *mkdir_corpus* for creating a new corpus subdirectory.

```
open Web;

value main =
  let query = Cgi.query_string () in
  let env = Cgi.create_env query in
  let dirname = Cgi.decoded_get Mkdir_corpus_params.dirname "" env in
  let parent_dir = Cgi.decoded_get Mkdir_corpus_params.parent_dir "" env in
  let permission =
    Cgi.decoded_get Mkdir_corpus_params.permission "" env
    |> Web_corpus.permission_of_string
  in
  let error_page = error_page "Corpus Manager" in
  match permission with
  [ Web_corpus.Manager →
    try
      do
      { Web_corpus.mkdir (Filename.concat parent_dir dirname)
      ; Corpus_manager.mk_page parent_dir permission
```

```
    }
    with
    [ Web_corpus.Section_already_exists abbrev →
      error_page "Already␣existing␣section␣" abbrev
    | Unix.Unix_error (err, func, arg) →
      let submsg =
        Printf.sprintf "'%s'␣failed␣on␣'%s':␣%s"
          func arg (Unix.error_message err)
      in
      error_page Control.sys_err_mess submsg
    | _ →
      abort Html.default_language Control.fatal_err_mess "Unexpected␣anomaly"
    ]
  | Web_corpus.Reader | Web_corpus.Annotator →
    let expected_permission = Web_corpus.(string_of_permission Manager) in
    let current_permission = Web_corpus.string_of_permission permission in
    invalid_corpus_permission_page expected_permission current_permission
  ]
;
```

Module Mk_corpus

This is an unfinished attempt to fiter out citations from Heritage and make a corpus document from it - unused at present

```
value abort report_error status =
  do
  { report_error ()
  ; exit status
  }
;
value citation_regexp = Str.regexp "\\\\citation{\\(.*\\)}"
;
value extract_citation state save_sentence line line_no =
  try
    if Str.string_match citation_regexp line 0 then
      save_sentence [ ("text", Str.matched_group 1 line) :: state ]
    else
      raise Exit
  with
```

```
  [ _ →
    abort (fun () →
        Printf.eprintf
          "Line␣%d:␣\
 ␣␣␣␣␣␣␣␣␣␣␣Wrong␣input␣format␣(expect␣one␣citation␣macro␣per␣line)\n" line_no
      ) 1
  ]
;
value populate_corpus dirname file =
  if dirname.val ≠ "" then
    let ch = open_in file in
    let (corpus_location, dirname) =
      if Filename.is_relative dirname.val then
        ("", dirname.val)
      else
        (Filename.dirname dirname.val, Filename.basename dirname.val)
    in
    let module Corp = Corpus.Make (struct value path = corpus_location; end) in
    let dirname =
      if Filename.check_suffix dirname Filename.dir_sep then
        Filename.chop_suffix dirname Filename.dir_sep
      else
        dirname
    in
    let rec aux i =
      try
        (* let line = input_line ch in let state = [ (Params.corpus_dir, dirname) ; (Params.senten
*)
      failwith "TODO"
        (* do { extract_citation state (Corp.save_sentence True Web.graph_cgi) line i ; aux (i + 1) }
*)
      with
      [ End_of_file → () ]
    in
    do
    { Corp.mkdir dirname
    ; aux 1
    ; close_in ch
    }
  else
```

```
    abort (fun () →
        Printf.eprintf
          "Please␣specify␣the␣destination␣directory.␣␣\
␣␣␣␣␣␣␣␣␣␣␣See␣%s␣--help.\n" (Filename.basename Sys.argv.(0))
      ) 1
;
(*************)
(* Entry point *)
(*************)
value main =
  let dirname = ref "" in
  let opts =
    Arg.align
      [ ("-d", Arg.Set_string dirname,
          "␣Specify␣the␣destination␣directory") ]
  in
  let usage_msg =
    Filename.basename Sys.argv.(0) ^ "␣-d␣<dest_dir>␣<citation_file>"
  in
  Arg.parse opts (populate_corpus dirname) usage_msg
;
```

Index