Homework 4. Morse code data recovery

Motivation

You are working for the Library of Congress and are helping to analyze some ancient audio recordings of Morse code operators in action. Someone else has written a program that takes the analog signals from these recordings and turns them into a digital representation of signal-on/signal-off.

Morse code represents each letter using a string of `.` (dih) and `-` (dah) symbols. At the signal level, 1 (a signal-on of duration 1) represents dih, 111 (a signal-on of duration 3) represents dah, 0 (a signal-off of duration 1) separates dihs from dahs within a letter, 000 (signal-off of duration 3) represents the boundary between two letters, and 0000000 (signal-off of duration 7) represent the space between words.

For example, if the message is `MORSE CO`, its Morse code is `-- --- .-. ... .` followed by a word space followed by `-.-. ---` and its signal representation is `1110111000111011101110001011101000101010001000000011101011101000111011101110111`; this last is the output of your colleague's program and the input to your program.

Unfortunately, the audio recordings are of poor quality, and their low-level analysis sometimes produces instances of 11 (two adjacent 1s only); these are errors, and the original signals could have been either 1 or 111. Similarly, the input to your program sometimes contains instances of 00, which could have been either 0 or 000 in the original data; of 0000, which must have been 000; and of 00000, which could be either 000 or 0000000. When the input is ambiguous, you need to evaluate either possibility, though you'll prefer to guess the shorter alternative first. You should treat any sequence of more than five 0s as if it were 0000000, and any sequence of more than three 1s as if it were 111.

For example, if the input is `11100111001111` you want to first guess that the actual data were `11101110111`; if this doesn't work you'll then fall back on `1110111000111`, `1110001110111`, and `111000111000111`, respectively.

Assignment

First, write a Prolog predicate `signal_morse/2` that converts a list of 1s and 0s to the corresponding list of `.`s, `-`s, `^`s, and `#`s. For example, the goal:

```
?- signal_morse(
     [1,1,1,0,1,1,1,
      0,0,0,1,1,1,0,1,1,1,0,1,1,1,0,0,0,1,0,1,1,1,
      0,1,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,
      1,1,1,0,1,0,1,1,1,0,1,0,0,0,1,1,1,0,1,1,1,
      0,1,1,1],
     M).
```

should succeed with:

```
M = [-,-,^,-,-,-,^,'.',-,'.',^,'.','.','.',^,'.',#,-,'.',-,'.',^,-,-,-]
```

Here `-` and `.` stand for themselves; `^` stands for a boundary between letters, and `#` stands for a space between words. In this representation, a word is any nonempty sequence of tokens other than `#`. (The GNU Prolog interpreter prints the atom `.` as `'.'`.)

If `signal_morse/2` gets an ambiguous signal list, it should generate all possible solutions, preferring solutions with shorter leftmost alternatives, as described above.

Second, write a Prolog predicate `signal_message/2` that converts a list of 1s and 0s to the corresponding list of letters, interpreted according to the following Morse code table, derived from Recommendation ITU-R M.1677-1 (2009).

```
morse(a, [.,-]).           % A
morse(b, [-,.,.,.]).       % B
morse(c, [-,.,-,.]).       % C
morse(d, [-,.,.]).         % D
morse(e, [.]).             % E
morse('e''', [.,.,-,.,.]). % É (accented E)
morse(f, [.,.,-,.]).       % F
morse(g, [-,-,.]).         % G
morse(h, [.,.,.,.]).       % H
morse(i, [.,.]).           % I
morse(j, [.,-,-,-]).       % J
morse(k, [-,.,-]).         % K or invitation to transmit
morse(l, [.,-,.,.]).       % L
morse(m, [-,-]).           % M
morse(n, [-,.]).           % N
morse(o, [-,-,-]).         % O
morse(p, [.,-,-,.]).       % P
morse(q, [-,-,.,-]).       % Q
morse(r, [.,-,.]).         % R
morse(s, [.,.,.]).         % S
morse(t, [-]).             % T
morse(u, [.,.,-]).         % U
morse(v, [.,.,.,-]).       % V
morse(w, [.,-,-]).         % W
morse(x, [-,.,.,-]).       % X or multiplication sign
morse(y, [-,.,-,-]).       % Y
morse(z, [-,-,.,.]).       % Z
morse(0, [-,-,-,-,-]).     % 0
morse(1, [.,-,-,-,-]).     % 1
morse(2, [.,.,-,-,-]).     % 2
morse(3, [.,.,.,-,-]).     % 3
morse(4, [.,.,.,.,-]).     % 4
morse(5, [.,.,.,.,.]).     % 5
morse(6, [-,.,.,.,.]).     % 6
morse(7, [-,-,.,.,.]).     % 7
morse(8, [-,-,-,.,.]).     % 8
morse(9, [-,-,-,-,.]).     % 9
morse(., [.,-,.,-,.,-]).   % . (period)
morse(',', [-,-,.,.,-,-]). % , (comma)
morse(:, [-,-,-,.,.,.]).   % : (colon or division sign)
morse(?, [.,.,-,-,.,.]).   % ? (question mark)
morse('''',[.,-,-,-,-,.]). % ' (apostrophe)
morse(-, [-,.,.,.,.,-]).   % - (hyphen or dash or subtraction sign)
morse(/, [-,.,.,-,.]).     % / (fraction bar or division sign)
morse('(', [-,.,-,-,.]).   % ( (left-hand bracket or parenthesis)
morse(')', [-,.,-,-,.,-]). % ) (right-hand bracket or parenthesis)
morse('"', [.,-,.,.,-,.]). % " (inverted commas or quotation marks)
morse(=, [-,.,.,.,-]).     % = (double hyphen)
morse(+, [.,-,.,-,.]).     % + (cross or addition sign)
morse(@, [.,-,-,.,-,.]).   % @ (commercial at)

% Error.
morse(error, [.,.,.,.,.,.,.,.,.]). % error - see below

% Prosigns.
morse(as, [.,-,.,.,.]).          % AS (wait A Second)
morse(ct, [-,.,-,.,-]).          % CT (starting signal, Copy This)
morse(sk, [.,.,.,-,.,-]).        % SK (end of work, Silent Key)
morse(sn, [.,.,.,-,.]).          % SN (understood, Sho' 'Nuff)
```

For example:

```
?- signal_message(
     [1,1,1,0,1,1,1,
      0,0,0,1,1,1,0,1,1,1,0,1,1,1,0,0,0,1,0,1,1,1,
      0,1,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,
      1,1,1,0,1,0,1,1,1,0,1,0,0,0,1,1,1,0,1,1,1,
      0,1,1,1],
     M).
```

should succeed with:

```
M = [m,o,r,s,e,#,c,o]
```

Again, `#` stands for the space between words, but there is no `^` at this level because the boundary between letters is understood.

If the signal contains noise, `signal_message/2` should backtrack among all possible messages that it can stand for, interpreted according to the above-mentioned disambiguation rules for noisy signals. However, `signal_message/2` should succeed only for disambiguated signals that correspond to a valid Morse code message.

As a special case, if your `signal_message/2` implementation finds a word, followed by zero or more spaces, followed by an `error` token, it should omit the word, the spaces, and the error token; it should then start scanning again after the omitted tokens, looking for further errors. For example:

```
?- signal_message(
     [1,1,1,0,1,1,1, % m
      0,0,0,
      1,1,1,0,1,1,1,0,1,1,1, % o
      0,0,0,
      1,0,1,1,1,0,1, % r
      0,0,0,
      1,0,1,0,1, % s
      0,0,0,
      1, % e
      0,0,0,0,0,0,0, % #
      1,1,1,0,1,0,1,1,1,0,1, % c
      0,0,0,
      1,1,1,0,1,1,1,0,1,1,1, % o
      0,0,0,
      1,0,1,0,1,0,1,0,1,0,1,0,1,0,1, % error
      0,0,0,
      1,0,1,0,1,0,1,0,1,0,1,0,1,0,1, % error
      0,0,0,
      1,0,1,0,1,0,1,0,1,0,1,0,1,0,1, % error
      0,0,0,
      1,0,1,1,1,0,1,0,1,0,1,0 % as
     ],
     M).
```

should succeed with `M=[m,o,r,s,e,#,error,error,as]`, because the partial word 'CO' was erased by the first error token. The later error tokens were not preceded by a word and optional spaces, so they are preserved in the output.

Your implementations of `signal_morse/2` and `signal_message/2` may assume that their first arguments are *ground terms*, that is, terms that contain no logical variables.

Your implementation should run on GNU Prolog 1.4.4 as installed on the SEASnet RHEL 7 hosts as `gprolog` in `/usr/local/cs/bin`.

Submit

To turn in your assignment, submit a file `hw4.pl` containing your definition of `signal_morse/2` and `signal_message/2` and any other auxiliary definitions. Your definitions must implement the same Morse code system that is specified in `morse/2` above (which you can include directly in your code). If any extra text information is needed, other than what's in the comments, please submit it as a separate text file `hw4.txt`. Please do not put your name, student ID, or other personally-identifying information in your files.

$Id: hw4.html,v 1.89 2016/10/25 20:48:36 eggert Exp $