Task 9, version 2. Morse coder

Adrian Jonsson Sjödin

Spring Term 2023

What has changed

In the previous report there was a problem in the handling of encoding input strings into Morse code. I was using a tail recursive method with an accumulator that after closer inspection ended up giving me a time complexity of $\mathcal{O}(n^2)$. This lead me to having to redo the whole encoding functionality in the program. However the decoding functionality worked as intended and have thus been left unchanged. The things that have been updated in the report is thus the first parts in the Method section that concerns the encoding functionality and the Discussion section. The rest have been left unchanged since the end result output was still the same and the decoding was left unchanged.

Introduction

In this task we implement a Morse coder that can take a text input and encode it into morse code. We also implement a decoder that can take a morse signal and output the message it contains.

The requirement on the encoder is that it should have a time complexity of $\mathcal{O}(n \cdot m)$, where n is the length of the message and m is the length of the morse codes. The lookup function in the encoder should have a time complexity of $\mathcal{O}(log(k))$, where k is the number of letters in the alphabet, or $\mathcal{O}(1)$

The requirement on the decoder is that it should use a data structure that will give an $\mathcal{O}(m)$ lookup operation. In other words it should be proportional to the length of the morse code.

Method

For the encoding function we are provided with a private function that returns a list of tuples with character codes and their morse code representation (see fig 1). We will later use this when we encode a message into morse code.

Code Overview 1: Char codes and morse representation

```
defp char_codes do
  [
    {32, '..--'},
    .
    .
    .
    {122, '--..'}
    ]
end
```

Now we can start on the main encoding function `encode/1` (see fig. 2). This is the function that will be called to convert a message into morse code. It take an input string and first convert everything into lower case and then into a list of character codes. We then build the encoding table by calling the private `build_encode_table/0` function (see fig. 3). This function creates a tuple with morse code representations for ASCII characters. It uses the `char_codes/0` function to get the list of tuples, fills in any missing codes with `:na` using the `fill_codes/2` function (see fig. 4), and then converts the list to a tuple. The `fill_codes/2` function is a recursive function that fills in missing codes in the encoding table with `:na`. It takes a list of tuples and an index and does the following:

1. If the list is empty, return an empty list.
2. If the first element of the tuple matches the index, add the Morse code to the list and call itself recursively with the remaining tuples and an incremented index.
3. If the first element of the tuple doesn't match the index, add :na to the list and call itself recursively with the same tuples and an incremented index.

Code Overview 2: The `encode/1` function

```
def encode(text) do
  input = String.to_charlist(String.downcase(text))
  table = build_encoding_table()
  process_encode(input, [], table)
end
```

Code Overview 3: The `build_encoding_table/3` function

```
defp build_encoding_table() do
  char_codes()
  |> fill_codes(0)
  |> List.to_tuple()
end
```

Code Overview 4: The `fill_codes/2` function

```
defp fill_codes([], _), do: []
defp fill_codes([{n, code} | codes], n), do: [code | fill_codes(codes, n + 1)]
defp fill_codes(codes, n), do: [:na | fill_codes(codes, n + 1)]
```

Having build the encoding table we now call the private `process_encode/3` function (see fig. 5) that is a tail recursive function that recursively encodes the input list of characters into morse code using the provided encoding table. This function process each character code in the char list one by one and looks up the corresponding morse representation with `find_code/2` (see fig. 6), and then call itself recursively with the remaining character codes and the found morse code saved. When this has gone through the whole input message it will call the function `combine_codes/2` (see fig. 6) that takes the saved morse codes and combines them into a single list, adding a space between each morse code, before presenting the result.

Code Overview 5: The `process_encode/3` function

```
defp process_encode([], all, _), do: combine_codes(all, [])
defp process_encode([char | rest], sofar, table) do
  code = find_code(char, table)
  process_encode(rest, [code | sofar], table)
end
```

Code Overview 6: The `find_codes/2` and `combine_codes/2` functions

```
defp find_code(char, table), do: elem(table, char)
defp combine_codes([], done), do: done
defp combine_codes([code | rest], sofar) do
  combine_codes(rest, code ++ [?\s | sofar])
end
```

Next we need to create the decoding functions. The input to this function will be a list of characters representing the morse code signal (dots, dashes, and spaces). The plan is to make use of the provided decoding tree to recursively decode the signal. To do this we first generate the provided tree and then pass it, the morse signal and an empty list as arguments to a private

`decode/3` function (see fig. 7) which will process the signal recursively by calling the `decode_char/2` (see fig. 8) function. This function matches the morse code signal with either a dot or dash (. or -) and a corresponding sub-tree in the binary tree. It then recursively calls `decode_char/2` with the remaining signal and the corresponding sub-tree. When the end of the signal is reached, it returns a tuple containing the decoded character and the remaining signal. If the signal cannot be matched, the function returns `:no`. The decode function then appends the decoded character to a list that is initially empty, until the end of the signal is reached. If the signal cannot be decoded, an error message is printed to the console.

To summarize the data structure used is the one that was provided, i.e. a decoding tree, which is built using a combination of tuples and atoms. The tuples represent nodes in the tree and the atoms are used to represent special states in the tree, such as the "no match" state (`:na`). The lookup function is `decode_char/2`, which takes a Morse signal and a decoding tree as input and recursively traverses the tree to decode the signal. The function returns a tuple `{char, rest}` if a character is successfully decoded from the signal, or the atom `:no` if the signal cannot be decoded.

Code Overview 7: The `decode/1` and `decode/3` functions

```
def decode(morse_signal) do
    # Build the decoding tree
    tree = decode_table()
    # Call the private decode function with the initial accumulator
    # set to an empty list
    decode(morse_signal, tree, [])
end

defp decode([], _, acc), do: acc

defp decode(morse_signal, tree, acc) do
  # Call the private decode_char function with the current Morse
  # signal and the decoding tree
  case decode_char(morse_signal, tree) do
    # If the decode_char function returns :no, it means the Morse
    # signal cannot be decoded
    :no ->
      :io.format("error: ~w\n", [Enum.take(morse_signal, 10)])
      acc

    # If the decode_char function returns a tuple {char, rest}, it
    # means a character was successfully decoded
    {char, rest} ->
```

```
      # Call the decode function recursively with the remaining Morse
      # signal and the updated accumulator
      decode(rest, tree, acc ++ [char])
  end
end
```

Code Overview 8: The `decode_char/2` function

```
defp decode_char([], {_, char, _, _}), do: {char, []}
defp decode_char([?- | morse_signal], {_, _, long, _}) do
  # If the next signal is a dash (-), follow the long path in the decoding tree
  decode_char(morse_signal, long)
end

defp decode_char([?. | morse_signal], {_, _, _, short}) do
  # If the next signal is a dot (.), follow the short path in the decoding tree
  decode_char(morse_signal, short)
end

# If the Morse signal is empty or the decoding tree is nil, return :no
defp decode_char(_morse_signal, nil), do: :no
# If the next signal is a space and the decoding tree is in the "no match" state,
# return :no
defp decode_char([?\s | _morse_signal], {_, :na, _, _}), do: :no
# If the next signal is a space and the decoding tree is in a character state,
# return the decoded character and the remaining Morse signal
defp decode_char([?\s | morse_signal], {_, char, _, _}), do: {char, morse_signal}
```

Result

In fig. 9 you see the result from encoding my name and decoding it as well as the two secret messages we were asked to decode.

Code Overview 9: Secret message and encoding and decoding of my name

```
iex> name = Morse.encode("Adrian Jonsson Sjodin")
'.- -.. .-. .. .- -. ..-- .--- --- -. ... ... --- -. ..-- ... .--- --- -.. .. -.'
iex> Morse.decode(name)
'adrian jonsson sjodin'
iex> secret1 = Morse.secret1()
'.- .-.. .-.. ..-- -.-- --- ..- .-. ..-- -... .- ... . ..-- .- .-. . ..-- -... .
.-.. --- -. --. ..-- - --- ..-- ..- ...'
iex> secret2 = Morse.secret2()
'.... - - .--. ... ---... .----- .----- .-- .-- .-- .-.-.- -.-- --- ..- - ..- -...
```

```
 . .-.-.- -.-. --- -- .----- .-- .- - -.-. .... ..--.. ...- .----. -.. .--.-- .....
  .---- .-- ....- .-- ----. .--.-- ..... --... --. .--.-- ..... ---.. -.-. .--.--
  ..... .----'
iex> Morse.decode(secret1)
'all your base are belong to us'
iex> Morse.decode(secret2)
'https://www.youtube.com/watch?v=d%51w4w9%57g%58c%51'
```

Discussion

The time complexity for the decoding should be $\mathcal{O}(n)$, where n is the length of the morse signal string. This because `decode` calls upon the function `decode_char` on each element of the string until it is fully decoded, or an error occur. Since each character is only visited once and the `decode_char` function is $\mathcal{O}(1)$, the time complexity of the decode function is $\mathcal{O}(n)$. The `decode_char` function is a constant time operation since it is a simple pattern-matching operation that does not depend on the length of the input. This means that the decoding satisfies the requirements.

When it comes to the encoding let us break it down and first look at each functions time complexity before looking at the overall time complexity.

The functions `char_codes/0`, `build_encoding_table/0`, `find_code/2` and `fill_codes` have a time complexity of $\mathcal{O}(1)$. This is because the size of `char_codes/0` is fixed and each of the following functions iterates over it once while doing its thing.

The function `encode/1` has a time complexity of $\mathcal{O}(n)$, where n is the number of characters in the input string. This because each character is processed once in the `process_encode/3` function, making that the dominating time operation in `encode/1`.

`process_encode/3` have a time complexity of $\mathcal{O}(n)$, where n is the number of characters in the input list passed to it. It iterates over this list once from start to end and swaps out the character codes with the morse code recursively.

Lastly we have the function `combine_codes/2` that is called when `process_encode/2` have gone through its list. This function will go through morse list and combine the codes and will thus have a time complexity of $\mathcal{O}(m)$, where m is the length of the list of morse codes.

This means that overall `encode/1` will have a time complexity of $\mathcal{O}(n + m) = \mathcal{O}(k)$, and thus we satisfy the conditions given for the encode functionality.

The full code can be found on my GitHub under the branch "new-morse".