Springs 5

Simon Yllmark

February 14, 2024

1 Introduction

In this assignment I implemented a program that figures out the number of possibilities a sequence of functioning, damaged and unknown sequence can have. The pods can either be functioning or damaged and thus the program's purpose is to figure out the unknowns to see how many sequences there is where the unknown can be functional or damaged.

2 Sequences

Well let's begin with describing the sequences given to the program. A given sequences for a row of ponds can look like this:

- "???.### 1,1,3"

  That will the be translated by the program to a list that look like this:

- `['???.###', [1,1,3]]`

This is done row by row by a function called rec() and the return a list containing lists like the one above. Here is the functions that create the sample, transform the sample to something better for the program to read and the evaluate each row in the generated list with the evaluation function evaluation().

```
  def sample() do
  "???.### 1,1,3\r\n.??..??...?##. 1,1,3\r\n?#?#?#?#?#?#?#? 1,3,1,6\r\n????.#...#
end

def parse() do
  description = sample()
  des = String.split(description,"\r\n")
  list = rec(des)
  lista = evaluation(list)
  lista
end
```

3 Evaluation

The evaluation function evaluate() that evaluates the list of listed sequence split the sequences into a listed sequence which then get evaluated by the counting function counting(), here it is:

```
def counting([],[]) do 1 end
def counting([],_) do 0 end
def counting(_,[]) do 1 end
def counting(list1,list2) do
  [element|rest] = list1
  [num|rest2] = list2
  case element do
    ?. -> counting(rest,list2)
    ?# -> case broken(list1,num) do
      {:ok, back} -> counting(back,rest2)
      false -> 0
    end
    ?? -> case broken(list1,num) do
       {:ok, back} -> counting(back,rest2) + counting(rest,list2)
       false -> counting(rest,list2)
    end
  end
end
```

What the counting() does is that it counts the number of valid sequences, for example this sequence:

```
['???.###', [1,1,3]]
```

where the list to the right of the character sequence say how many # there should be and in what order. for example the sequence:

```
['??', [1]]
```

There are two outcome that are valid and those are

```
'#.' and '.#'
```

because the list to the right say that there must be one pond that is broken in that sequence. Thus, what the counting function does is that it calculates the number of valid end states because the function recursively iterate the different states like a tree like structure.

4 Broken

The previous section did not go into detail about the evaluation function evaluation() helper function broken(). That function handles cases where the element in the list is # or ? in order to calculate the deviating branches in the tree like structure that the evaluation function takes in it tree like search pattern for end states. Here is the broken function:

```
    def broken([],0) do {:ok,[]} end
  def broken([], _) do false end
  def broken(list,0) do
    [element|rest] = list
    case element do
       ?#-> false
        _ -> {:ok,rest}
    end
   end
  def broken(list,length) do
    [element|rest] = list
    case element do
      ?. -> false
      ?# -> broken(rest,length-1)
      ?? -> broken(rest,length-1)
    end
  end
```

What the function does is that it either return false when it reaches a invalid end state or or the tuple

```
    {:ok, depending}
```

when it reaches an valid state. The value of depending depend on the reach end state. It could be the rest of the list or an empty list.

5 Conclusion

This was an assignment where expertise about recursive programming would come in extra handy in comparison to the other assignments, because to solution of this assignment came from having a good understanding of how recursive programming work and following the recursive branches of the tree for trouble shooting.