KTH Information and Communication Technology

ID1019
Johan Montelius

Programming II (ID1019)
2020-03-09

Instructions

- All answers should be written in these pages, use the space allocated after each question to write down your answer (not on the back side).
- Answers should be written in English.
- You should hand in the whole exam.
- No additional pages should be handed in.

Grade

The exam consists of two parts. The first five questions are about basic functional programming: pattern matching, recursion, immutable data structures etc. The first part is the basic requirement to pass the course:

- FX: 7 points
- E: 8 points

The second part, questions 6-9, are about: semantics, higher-order functions, complexity, processes etc. The higher grades are based only on these questions but is only given (and only corrected) if the basic part has been answered satisfyingly (8 out of 10 point).

- D: one question correctly answered
- C: two questions correctly answered
- B: three questions correctly answered
- A: all questions correctly answered

Namn:________________________________ Persnr:________________________________

1 toggle [2p]

Implement a function, `toggle/1`, that takes a list and returns a list where the elements have changed place two-by-two. If the number of elements is odd the last element keeps its position.

Example: `toggle([:a,:b,:c,:d,:e,:f,:g])` should give the answer `[:b,:a,:d,:c,:f,:e,:g]`.

Answer:

```
def toggle([a,b|rest]) do [b, a | toggle(rest)] end
def toggle(rest) do rest end
```

2 push and pop [2p]

Implement a stack: propose a data structure and implement the two functions `push/2` and `pop/1`. The function `push/2` should return the updated stack and the function `pop/1` should return either: `{:ok, any(), stack()}` if there is an element on the stack or `:no` if the stack is empty.

Answer:

```
 @type stack() :: [any()]

 def push(stack, elem) do [elem|stack] end

 def pop([]) do :no end
 def pop([elem|rest]) do {:ok, elem, rest} end
```

Namn:________________________________ Persnr:________________________________

3 flatten [2p]

Implement a function `flatten/1` that takes a list of lists, that in turn can be lists of lists etc, and returns a flat list with all the elements of the original list in the same order. You can use `++` to append two lists.

Example, the call: `flatten([1,[2],[[3,[4,5]], 6]])` returns: `[1,2,3,4,5,6]`

Answer:

```
def flatten([]) do [] end
def flatten([head|tail]) do flatten(head) ++ flatten(tail) end
def flatten(item) do  [item] end
```

Namn:________________________________ Persnr:________________________________

4 h-index [2p]

An h-index can be used to describe how much and how far one runs. Your h-index is the highest value *h* such that you have run at least *h* km at least *h* times. Your h-index is for example 12 if you have done 12 km or more 16 times but have not done 13 km or more 13 times. It is fairly simple to have a h-index of 10 but a bit tougher to come up to 30.

Implement a function `index/1` that, given a list of values that describes runs in km, calculates the h-index. The list is ordered with the longest runs first. You will be able to calculate the value by going through the list only once. The algorithm is simple:

- The initial estimate of the h-index is 0.
- Traverse the list and if the first element is greater than the current estimate,
  - then increment the estaimate by 1 and continue otherwise,
  - you have found the correct h-index.

Example: `index([12,10,8,8,6,4,4,4,2])`

should return: `5`

Answer:

```
def index(runs) do index(runs, 0) end

def index([k|rest], n) when k > n do
  index(rest, n+1)
end
def index(_, n) do n end
```

Namn:______________________________ Persnr:______________________________

5 more compact [2p]

Implement a function `compact/1` that takes a tree on the form below and returns a tree where each node with the same key is both its leafs (or if there is only one leaf) had been replaced with a single leaf. The function should be applied recursivly so that changes propagate towords the root.

Trees are represented as follows, note that only the leaves have values:

```
@type tree() :: :nil | {:node, tree(), tree()} | {:leaf, any()}
```

Example:

```
compact({:node, {:leaf, 4}, {:leaf, 4}})
```

should return: `{:leaf, 4}`

```
compact({:node, {:leaf, 5}, {:node, :nil, {:leaf, 4}}})
```

should return: `{:node, {:leaf, 5}, {:leaf, 4}}`

Answer:

```
def compact(:nil) do :nil end
def compact({:leaf, value}) do {:leaf,value} end
def compact({:node, left, right}) do
  cl = compact(left)
  cr = compact(right)
  combine(cl, cr)
end


def combine(:nil, {:leaf, value}) do
  {:leaf, value}
end
def combine({:leaf, value}, :nil) do
  {:leaf, value}
end
def combine({:leaf, value}, {:leaf, value}) do
  {:leaf, value}
end
def combine(left, right) do
  {:node, left, right}
end
```

Namn:________________________________ Persnr:________________________________

6 Formal semantics [P/F]

In the formal semantic that we have discussed we have used a rule call the S-rule or *scoping rule*. The rule is used when we describe how a sequence of pattern matching expressions are evaluated.

In Elixir we have the possibility to use pattern matching in the head of a function definition (see example below). How should we describe the semantics of a function call if we allow patternmatching in the head?

$$\frac{\sigma' = \sigma \setminus \{v/n \quad | \quad v/n \in \sigma \quad \wedge \quad v \quad \text{in} \quad p\}}{S(\sigma, p) \rightarrow \sigma'}$$

Example of how pattern matching in the head of a function definition:

```
def member(_, []) do  :no end
def member(x, [x|_]) do  :yes end
def member(x, [_|h]) do  member(x, h) end
```

This is how we defined a function call when the function was only allowed to have unique variables in its head.

$$\frac{E\sigma(e) \rightarrow < v_1, \ldots : \text{seq} : \theta > \qquad E\sigma(e_i) \rightarrow s_i \qquad E\{v_1/s_1, \ldots\} \cup \theta(\text{seq}) \rightarrow s}{E\sigma(e.(e_1, \ldots)) \rightarrow s}$$

How is this changed if we allow patterns in the head (we assume there is only one head)?

$$\frac{E\sigma(e) \rightarrow < p_1, \ldots : \text{seq} : \theta > \qquad\qquad\qquad\qquad\qquad\qquad}{E\sigma(e.(e_1, \ldots)) \rightarrow s}$$

Reply on the next page.

Namn:______________________________ Persnr:______________________________

Answer:

$$\frac{E\sigma(e) \rightarrow < p_1, \ldots p_n : \text{seq} : \theta > \quad E\sigma(e_i) \rightarrow s_i \quad \gamma_0 = \{\} \quad P\gamma_{i-1}(p_i, s_i) \rightarrow \gamma_i \quad E\gamma_n \cup \theta(seq) \rightarrow s}{E\sigma(e.(e_1, \ldots e_n)) \rightarrow s}$$

$$\frac{E\sigma(e) \rightarrow < p_1, \ldots p_n : \text{seq} : \theta > \quad E\sigma(e_i) \rightarrow s_i \quad \gamma_0 = \{\} \quad P\gamma_{i-1}(p_i, s_i) \rightarrow \text{fail}}{E\sigma(e.(e_1, \ldots e_n)) \rightarrow \text{fail}}$$

There is no need for a S-rule. We should however make sure that we accumulate the variable bindings. In the example with `member/2` the variable `x` in the second clause the same variable.

Namn:________________________________ Persnr:________________________________

7 Next prime number [P/F]

You should implement a function `primes/0` that returns a function that represents the endless sequence of prime numbers. The returned function should when applied to no argumnets, return a tuple `{:ok, prime, next}` where prime is the next prime number and next a function that will give us the following prime numbers.

```
@type next() :: {:ok, integer(), ( -> next())}

@spec primes() :: ( -> next())
```

One way to solve this is to implement sieve of Eratosthenes. When you have found a prime number p (the first one is 2) then you make sure that you do not return any number n that is a multiple of this prime ($rem(n,p) == 0$). Below is a good start of this algorithm, it is now up to you to implement `sieve/2`.

```
def primes() do
  fn() -> {:ok, 2, fn() -> sieve(2, fn() -> next(3) end) end} end
end

def next(n) do
  {:ok, n, fn() -> next(n+1) end}
end
```

Answer:

```
def sieve(p, f) do
  {:ok, n, f} = f.()
  if rem(n, p) == 0 do
    sieve(p, f)
  else
    {:ok, n, fn() -> sieve(n, fn() -> sieve(p, f) end) end}
  end
end
```

8 a better flatten [P/F]

If you implement `flatten/1` as simple as possible you will end up with an algorith with less than perfect time complexity. You will append longer and longer lists and the complexity will be quadratic to the number of elements in the list. You shall now implement `flatten/1` but make sure that the complexity is linear with respect to the number of elements in the lists.

Answer:

```
def improved([]) do [] end
def improved([[] | rest]) do
  improved(rest)
end
def improved([[head | tail] | rest]) do
  improved([head, tail | rest])
end
def improved([ elem | rest]) do
  [elem | improved(rest)]
end
```

Namn:________________________________ Persnr:________________________________

9 parallel map [P/F]

You should implementa funktion `pmap/` that works as regular `map/2` but where each element has been computed in a separat process thus allowing for parallel execution.

You should use the regular `map/2` when you spawn the processes and collect the result.

Example: the expression below

```
pmap([1,2,3,4], fn(x) -> x + 2 end)
```

should return: `[3,4,5,6]`

Answer:

```
def pmap(list, fun) do
  refs = Enum.map(list, parallel(fun))
  Enum.map(refs, collect())
end

def parallel(fun) do
  me = self()
  fn(x) ->
    ref = make_ref()
    spawn(fn() ->
      res = fun.(x)
      send(me, {:ok, ref, res})
    end)
    ref
  end
end

def collect() do
  fn(r) -> receive do {:ok, ^r, res} -> res end end
end
```