Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2020-2021

Programming Exam 2

Due date: 15 April 2021, Thursday, 23:59

1 Problem Definition

For this second programming examination composed of two parts, you will be working with lists. In the first part, you will be doing some basic manipulations on infinite lists. In the second part, you will be writing a prettifier for a simplified object format. All functions will be tested independently, but are designed to *build up*. Thus, using previous functions when defining your new functions will make life easier.

1.1 General Specifications

- The signatures of the functions, their explanations and specifications are given in the following section. Read them carefully.
- Make sure that your implementations comply with the function signatures.
- You may define any number of helper function(s) as you need.
- You can (and sometimes even should) use previous functions inside your new function definitions. The exercises are designed to build up!
- You are allowed to import `Data.List` for this exam, if you want to. However, what the `Prelude` provides is already more than enough.

1.2 Quick VPL Tips

- Evaluation is fast. If evaluation seems to hang for more than a few seconds, your code is entering an infinite loop or has an abnormal algorithmic complexity. Or you've lost your connection, which is *much* less likely!
- Although the run console does not support keyboard shortcuts, it should still be possible to copy and paste using the right-click menu (Tested on latest versions of Firefox and Chrome).
- Get familiar with the environment. Press the plus shapes button on the top-left corner to see all the options. You can download/upload files, change your font and theme, switch to fullscreen etc. Useful stuff!

2 Part I - Time to ∞ up!

2.1 naturals (10 points)

Define an infinite list of type `[Integer]` containing all natural numbers in ascending order.

To explain a little more: the head of the list should be 0, the next value should be 1, followed by 2 etc. That's it!

The formal description for the ith element of the list n_i (starting from n_0) is the following:

$$n_i = i$$

Here is the signature and a few examples showcasing how `naturals` should behave, even though it's kind of obvious:

```
naturals :: [Integer]
```

```
*PE2> naturals !! 0
0
*PE2> naturals !! 1
1
*PE2> take 10 naturals
[0,1,2,3,4,5,6,7,8,9]
*PE2> take 25 naturals
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25]
*PE2> naturals !! 3333333
3333333
```

Hint: Yep, this is as easy as it looks. Remember `take` is used to get the first n elements of a given list.

2.2 interleave (20 pts)

Now, we're moving on to the next step to prepare for an even more awesome infinite list. You will implement a function named `interleave` for *interleaving* two lists of the same type; as the name implies.

The resulting list should alternate between elements from the first list and elements from the second list, starting from the first. The interleaving should **stop** as soon as **any** of the lists become empty. This implies that the length of the resulting list will always be even for finite lists: twice the length of the shorter input. Here is the signature and some sample runs to clarify (observe that the result always cuts off at the shorter list, but also works with infinite lists -in the final example- thanks to laziness):

```
interleave :: [a] -> [a] -> [a]
```

```
*PE2> interleave [1, 2, 3] [4, 5, 6]
[1,4,2,5,3,6]
*PE2> interleave [4, 5, 6] [1, 2, 3]
[4,1,5,2,6,3]
*PE2> interleave "hlool" "elwrd"
"helloworld"
*PE2> interleave [] [1, 2, 3]
[]
*PE2> interleave [3, 66, 7] []
[]
*PE2> interleave [5, 4, 3, 2] [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[5,1,4,1,3,1,2,1]
*PE2> interleave [5, 4, 3, 2] $ repeat 1
[5,1,4,1,3,1,2,1]
*PE2> take 10 $ interleave (repeat 2) (repeat 4)
[2,4,2,4,2,4,2,4,2,4]
```

Hint: No hints here! Remember that `repeat` creates infinite lists by repeating the given value.

2.3 integers (10 pts)

As the final step of this part, you will implement another infinite list of type `[Integer]`. This time it will contain all the integers, starting from zero and then alternating between the negative and positive values of the next natural number. The first value is 0, followed by -1 and 1, then -2 and 2, then -3 and 3 etc. Here's the signature and some example interactions:

```
integers :: [Integer]
```

```
*PE2> integers !! 0
0
*PE2> take 5 integers
[0,-1,1,-2,2]
*PE2> take 25 integers
[0,-1,1,-2,2,-3,3,-4,4,-5,5,-6,6,-7,7,-8,8,-9,9,-10,10,-11,11,-12,12]
*PE2> integers !! 312455
-156228
*PE2> integers !! 312456
156228
```

And finally, a more formal definition for the ith element n_i (starting from n_0):

$$n_i = \begin{cases} \dfrac{i}{2} & i \equiv 0 \mod 2 \\ -\dfrac{i+1}{2} & i \equiv 1 \mod 2 \end{cases}$$

Hint: Hey, maybe your previous definitions can be useful?

3 Part II - Prettifying SJSON

In this section we will be working on prettifying a simplified object format that we will call SJSON (a very restricted subset of actual JSON). The rules of this format are simple:

- Each SJSON strings describes an object; objects are delimited by braces `'{'` and `'}'`.
- Each object contains **one or more** *fields*, which are key-value pairs, where the key and value are separated by a colon `':'`
- Keys are always strings, delimited by single quotes `''`
- Values can be either strings or other objects
- If there are multiple key-value pairs, they are separated by a comma `','`
- For simplicity, strings only contain alphanumeric characters and spaces `' '`. There are no escape sequences or weird rules.
- Parts of the SJSON representation may be separated by zero or more characters of *whitespace*, which in this instance means spaces `' '`, tabs `'\t'` and newlines `'\n'`. Removing these spaces or adding more does not change the representation. Obviously, space characters `' '` inside strings should not be removed.

Here are some example strings representing SJSON objects:

- A basic object containing one key-value pair: `"{'key':'value'}"`
- An object containing multiple key-value pairs, with some tabs and spaces:
  `" { 'name': 'Simon Peyton', 'surname': 'Jones','age':'63' } "`
- An object containing another object broken over multiple lines:

```
" {
      'firstName': 'John', 'lastName': 'Smith',
      'isAlive': 'true',
      'age': '27',
      'address': { 'streetAddress': '21 2nd Street',
                   'city': 'New York', 'state': 'NY',
                   'postalCode': '10021-3100' },
      'spouse': 'noone' }"
```

3.1 splitOn (20 pts)

Now that we've defined SJSON, it's time to get to work! As a first step, you will define a simple function for splitting a string on the first occurence of a character. The function returns a tuple, with the first element containing the left side of the split string, and the second element containing the right side. Please note that the character split on is removed. If the character does not exist, the whole input string should be returned as the first element (left), with an empty string as the second element. Here is the signature and some sample I/O:

```haskell
splitOn :: Char -- character to split the string on
        -> String -- string to split
        -> (String, String) -- left and right pieces of the split string
```

```
*PE2> splitOn '.' "point.x"
("point","x")
*PE2> splitOn ',' "11,12,13,14"
("11","12,13,14")
*PE2> splitOn 'a' ""
("","")
*PE2> splitOn 'a' "a"
("","")
*PE2> splitOn 'a' "bab"
("b","b")
*PE2> splitOn '-' "no minuses here my friend"
("no minuses here my friend","")
*PE2> splitOn 'z' "zzzzz"
("","zzzz")
*PE2> splitOn '\'' "get'separated"
("get","separated")
```

3.2 tokenizeS (20 pts)

Directly working on characters when trying to process structured text can be difficult, so your next job is writing a *tokenizer* that will convert the input into a list of tokens (strings) that will be easier to work with in your prettifier.

Here's how your tokenizer should work:

- Special characters `'{'`, `'}'`, `':'` and `','` must be converted into tokens as-is. e.g. the character `':'` will become the string `":"`.
- Strings should be converted into tokens without their quotes. e.g. `'key'` becomes the string `"key"`, `'Chicken Egg 12'` becomes the string `"Chicken Egg 24"`.
- Whitespace should be completely ignored and not tokenized, **except for space characters present inside strings** which should clearly remain as-is. Remember that we defined whitespace as spaces, tabs and newlines.

The inputs to your function are guaranteed to be valid SJSON representations, as we have defined previously. There will be no erroneous input with extra characters or not conforming to the SJSON syntax rules. Here is the signature and a couple of example runs to make things clearer:

```
tokenizeS :: String -- valid SJSON string to tokenize
          -> [String] -- list of tokens resulting from the tokenization
```

```
*PE2> tokenizeS "{'alpha':'77'}"
["{","alpha",":","77","}"]
*PE2> tokenizeS "{'key0':'val0','key1':'val1'}"
["{","key0",":","val0",",","key1",":","val1","}"]
*PE2> tokenizeS "    { 'howabout' : 'somewhitespace'   } "
["{","howabout",":","somewhitespace","}"]
*PE2> tokenizeS "{\n\t 'spaces inside a string' : 'true'\n   }\n"
["{","spaces inside a string",":","true","}"]
*PE2> tokenizeS "{{{'nested':'sth'}},'not nested':'val'}"
["{","{","{","nested",":","sth","}","}",",","not nested",":","val","}"]
*PE2> tokenizeS "{ 'cat': { 'name':   'Boncuk',  'age': '2' }, 'dog':{'name':'Charl
ie', 'age': '7'}, \n 'cow': {'name': 'Tinker Bell', 'age': '8'} }"
["{","cat",":","{","name",":","Boncuk",",","age",":","2","}",",","dog",":","{","nam
e",":","Charlie",",","age",":","7","}",",","cow",":","{","name",":","Tinker Bell","
,","age",":","8","}","}"]
```

Hint: What you need to deal with spaces inside strings *is a simple algorithm, but quite unbreakable.* While going through the input, ignore any whitespace you come across. But, when you find a single quote `''`, simply take everything until the next single quote. `splitOn` could help with this. With this approach, you can safely include spaces inside strings while ignoring whitespace in the rest of the input. Remember that you need an *escape sequence* for the single quote *character literal*: `'\''`.

3.3 prettifyS (20 pts)

As the final step, you will write a prettifier for the SJSON format. This means that we will be printing our SJSON objects using a nice, indented format. Should not be too difficult thanks to the tokenizer you just wrote!

Below are our rules for prettification. Make sure you follow them **carefully**. Your output will not be post-processed, it needs to match the expected output exactly.

- Opening braces `'{'` should be followed by a newline, and increase the indentation level by one for following lines.
- Closing braces `'}'` should be preceded by a newline, and be at the same indentation level as the matching opening brace. They should reduce the indentation level by one.
- Every indentation level adds exactly four (4) spaces to the start of the line
- Strings should be written enclosed in single quotes `''`
- Colons `':'` separating key-value pairs should be followed by a single (1) space
- Commas `','` separating fields should be followed by a newline

With this approach, we can get some pretty SJSON outputs! Below is the signature and some examples. Since `show` shows newlines and tabs with their escape sequences, I'm also going to use `putStrLn` to show the outputs in a more proper fashion. Please note that `putStrLn` adds an extra newline at the end of the string, the final closing brace is **not** followed by a newline:

```haskell
prettifyS :: String -> String -- valid SJSON string to prettified output string
```

```
*PE2> prettifyS "{'chicken':'egg'}"
"{\n    'chicken': 'egg'\n}"
*PE2> putStrLn $ prettifyS "{'chicken':'egg'}"
{
    'chicken': 'egg'
}
*PE2> prettifyS "{ 'urfa':    'bland',   'adana':'spicy' }"
"{\n    'urfa': 'bland',\n    'adana': 'spicy'\n}"
*PE2> putStrLn $ prettifyS "{ 'urfa':    'bland',   'adana':'spicy' }"
{
    'urfa': 'bland',
    'adana': 'spicy'
}
*PE2> putStrLn $ prettifyS "{'header': 'here', 'payload': { 'local time': '1750 hou
rs', 'time zone': 'utc plus 3'}}"
{
    'header': 'here',
    'payload': {
        'local time': '1750 hours',
        'time zone': 'utc plus 3'
    }
}
*PE2> putStrLn $ prettifyS "{'country':'Turkey', 'university': { 'name': 'METU', 'd
epartment': { 'name': 'CEng', 'course': { 'name': 'CENG242', 'semester':'20202'}}},
'city':'Ankara'}"
{
    'country': 'Turkey',
    'university': {
        'name': 'METU',
        'department': {
            'name': 'CEng',
            'course': {
                'name': 'CENG242',
                'semester': '20202'
            }
        }
    },
    'city': 'Ankara'
}
```

Hint: Since the SJSON format is recursive (objects can be nested arbitrarily), your prettifier should *probably* also use recursion. The exact output can be difficult to understand from the box above. Make sure to check out the given example I/O files directly if you need more clarification. Once again, all input strings are guaranteed to be valid SJSON objects.

4 Regulations

1. **Implementation and Submission:** The template file named "pe2.hs" is available in the Virtual Programming Lab (VPL) activity called "PE2" on `OdtuClass`. At this point, you have two options:
   - You can download the template file, complete the implementation and test it with the given sample I/O on your local machine. Then submit the same file through this activity.
   - You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submit a file.

   The second one is recommended. However, if you're more comfortable with working on your local machine, feel free to do it. Just make sure that your implementation can be compiled and tested in the VPL environment after you submit it.

   There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

3. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

   Important Note: The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official testcases to determine your ***actual*** grade after the deadline.

5 Appendix - SJSON Grammar

For the sake of completeness, here's the grammar defining SJSON strings in BNF notation (with an obvious ... extension), without accounting for extraneous, token separating whitespace.

⟨*sjson*⟩ ::= '`{`' ⟨*fields*⟩ '`}`'

⟨*fields*⟩ ::= ⟨*field*⟩ | ⟨*field*⟩ '`,`' ⟨*fields*⟩

⟨*field*⟩ ::= ⟨*string*⟩ '`:`' ⟨*value*⟩

⟨*value*⟩ ::= ⟨*string*⟩ | ⟨*sjson*⟩

⟨*string*⟩ ::= '"' ⟨*maybechars*⟩ '"'

⟨*maybechars*⟩ ::= ⟨*char*⟩⟨*maybechars*⟩ | ϵ

⟨*char*⟩ ::= ' ' | ⟨*uppercase*⟩ | ⟨*lowercase*⟩ | ⟨*digit*⟩

⟨*uppercase*⟩ ::= '`A`' | '`B`' | ... | '`Z`'

⟨*lowercase*⟩ ::= '`a`' | '`b`' | ... | '`z`'

⟨*digits*⟩ ::= '`0`' | '`1`' | ... | '`9`'