(*Adapted to Source of AY2019/2020 Semester 1*)

Programming Tasks for Practical Assessment (PE)

CS1101S — Programming Methodology
National University of Singapore
School of Computing

Semester 1 AY2016/2017

10 November 2016 Time Allowed: **1 Hour 45 Minutes**

Instructions (please read carefully):

1. This question booklet comprises **11 printed pages** and has **3 questions** with a total of **60 marks**. Answer all questions.

2. This is an **open book** assessment. Any written or printed material, or programs stored on the Source Academy may be used as reference material.

3. The internet must not be used, except the site [deleted].

4. The questions in this paper should be answered and submitted on the Source Academy at [deleted]. Please go to **Assessments −> PE 2016** to attempt the questions.

5. Remember to **submit** your solutions at the end of the practical assessment. However, programs **cannot be resubmitted** after they have been submitted the first time.

6. All programs should be written in ~~**Source Week 10**~~ **Source §4**, unless otherwise stated.

7. The `assert` function calls in the solution templates provide some sample test cases to check the correctness of your solutions. You may view the test results in the 'Interpreter' tab. Note that the given test cases are not exhaustive; passing them does not mean your solution is correct. You are strongly encouraged to add your own test cases.

8. The solutions of some sub-questions require correct solutions of previous questions and sub-questions. In the environment that we set up for you, **you can program and test your solutions without dependencies**. This is achieved by the `assert` function. Each time `assert` is called, our correct implementation of the solutions to all relevant previous questions and sub-questions is installed in the global environment. We therefore strongly encourage you to **test your programs only using** `assert`.

GOOD LUCK!

Question 1: It's All in Your Genes [25 marks]

In the human body, proteins are directing or regulating or at least influencing most phenomena of any significance. The information used for making proteins is encoded in ***genes***, which are specific segments of strands of ***DNA*** (deoxyribonucleic acid).

In this question, no prior knowledge of microbiology is required. The examiners took the liberty to highlight only a small selection of the fascinating processes that make our bodies work.

1A. Nucleobase Test [2 marks]

DNA strands are very long molecules that contain ***nucleobases***. There are four such nucleobases in a DNA strand, which we represent by the strings

- `"A"` (adenine),
- `"C"` (cytosine),
- `"G"` (guanine), and
- `"T"` (thymine).

Write a function `is_nucleobase` that takes a string as argument and returns true if the string represents a nucleobase.

Examples:

```
is_nucleobase("Otto"); // false
is_nucleobase("G");    // true
is_nucleobase("B");    // false
is_nucleobase("A");    // true
```

1B. DNA Strand Test [3 marks]

DNA strands are "directed"; they are always read in a particular direction. We therefore represent them as lists. Write a function `is_dna_strand` that takes a list of strings as argument and returns true if every element string represents a nucleobase.

Examples:

```
is_dna_strand(list("A", "G", "A"));      // true
is_dna_strand(list("A", "B", "B", "A")); // false
is_dna_strand(list("T", "G", "C"));      // true
is_dna_strand(list("T", "G", "Otto"));   // false
```

1C. Combining Strands [2 marks]

The DNA strands in human cells contain up to 250 million nucleobases. In order to identify a DNA strand, researchers work with smaller sequences, which they then put together into longer ones.

Write a function `combine` that takes a list of DNA strands and combines them into a single DNA strand without changing their order.

Examples:

```
combine(list(list("A", "G", "A"),
             list("G", "C", "T", "A"), list("C")));
// returns list("A", "G", "A", "G", "C", "T", "A", "C")

combine(list(list("G"), list("T"),
             list("C", "A", "A", "A"), list("G")));
// returns list("G", "T", "C", "A", "A", "A", "G")
```

1D. DNA Repair [2 marks]

Ionizing radiation can cause guanine `"G"` to be transformed into the 8-oxoguanine, which we represent by the string `"8"`. In this exercise, we mimic the process of repairing such DNA damage.

Write a function `oxoguanine_repair`, which takes a list of `"A"`, `"C"`, `"G"`, `"8"` and `"T"` and returns a list in which every `"8"` is replaced by `"G"`.

Example:

```
oxoguanine_repair(
           list("A", "8", "A", "8", "C", "T", "A", "C"));
// returns list("A", "G", "A", "G", "C", "T", "A", "C")
```

1E. Finding Gene Start [5 marks]

A ***gene*** is a segment in a DNA strand that can be used to make a protein. In the human body, all genes start with a sequence ATG (called the ***start codon***).

Write a function `find_gene_start` that takes a DNA strand as argument and finds the strand *after* the *first occurrence* of ATG. If your function `find_gene_start` finds an ATG sequence, it returns the following sequence in a one-element list. If there is no ATG sequence, `find_gene_start` returns the empty list `null`.

Examples:

```
find_gene_start(list("A", "C", "A", "T", "G", "T", "A", "C");
// returns list(list("T", "A", "C"))

find_gene_start(list("A", "T", "A", "G", "T", "A", "T", "G");
// returns list(null)

find_gene_start(list("A", "T", "A", "G", "T", "A", "C", "G");
// returns null
```

1F. Finding Gene End [6 marks]

The sequences TAG, TAA and TGA are called ***stop codons***. Genes start with the start codon ATG and end with the *closest following* stop codon. As a result, a gene never contains a stop codon.

Write a function `find_gene_end` that finds a gene, which is the sequence up to but not including the next stop codon. If the function `find_gene_end` finds a gene, it returns it in a one-element list. If it does not find a gene, it returns the empty list `null`.

Examples:

```
find_gene_end(list("A", "T", "A", "C", "T", "A", "G",
                   "A", "T", "A", "A"));
// returns list(list("A", "T", "A", "C"))

find_gene_end(list("T", "G", "A", "A", "T", "A", "C"));
// returns list(null)

find_gene_end(list("A", "T", "A", "C", "C", "A", "G",
                   "A", "T"));
// returns null
```

1G. Catching All Genes [5 marks]

Write a function `all_genes` that finds all genes in a given DNA strand. The function `all_genes` should return a list of DNA strands. You can assume that genes do not overlap in the input DNA strand. Note that we do not impose any restriction on the length of a gene.

Example:

```
all_genes(list("T", "A", "T", "G", "C", "A", "T",
               "A", "A", "G", "T", "A", "G", "A",
               "T", "G", "A", "T", "G", "A", "T"));
// returns list(list("C", "A"), list("A"))
```

Question 2: The Game of TOTO [15 marks]

TOTO is a lottery game in which players place bets by buying TOTO ***tickets***. Each ticket is a set of *n* different integers, where each integer is chosen from [*min*, *max*] (this denotes a range from *min* to *max*, inclusive of *min* and *max*).

On the Draw Day, a ***winning set*** of *n* different integers are drawn, where each integer is in [*min*, *max*]. An ***extra number*** is also drawn, which is an integer different from all the *n* numbers in the winning set, and is also in [*min*, *max*].

The prize won for a ticket depends on how many numbers on the ticket match the numbers in the wining set and the extra number.

2A. [3 marks]

We are representing a set of numbers as a list of numbers in Source. Write a function, `all_different(nums)`, that takes in a list of numbers, `nums`, and returns true if and only if all the numbers in the list are different from each other. Note that the numbers in the list may not be sorted.

Examples:

```
all_different(list(23));
// returns true

all_different(list(2, 5, 1, 6, 7, 4, 3));
// returns true

all_different(list(2, 6, 1, 7, 6, 4, 3));
// returns false
```

2B. [4 marks]

Write a function, `is_valid_toto_set(nums, n, min, max)`, that takes in a list of integers, `nums`, and the integers `n`, `min` and `max`, and returns true if and only if the list of integers forms a valid set of numbers for a TOTO ticket (i.e. there are exactly `n` numbers, each number is in the range `min` to `max`, and all the numbers are different from each other).

Examples:

```
const nums = list(5, 1, 8, 49);
const n = 6;
const min = 1;
const max = 49;
is_valid_toto_set(nums, n, min, max);
// returns false
// Reason: length(nums) !== n.
```

```
const nums = list(25, 13, 2, 31, 30, 3, 15);
const n = 7;
const min = 3;
const max = 30;
is_valid_toto_set(nums, n, min, max);
// returns false
// Reason: the element 2 of nums is smaller than min.


const nums = list(25, 13, 8, 14, 30, 3, 8);
const n = 7;
const min = 3;
const max = 30;
is_valid_toto_set(nums, n, min, max);
// returns false
// Reason: 8 appears twice in nums.


const nums = list(25, 13, 8, 14, 30, 3, 15);
const n = 7;
const min = 3;
const max = 30;
is_valid_toto_set(nums, n, min, max);
// returns true
```

2C. [4 marks]

Write a function, `num_of_matches(numsA, numsB)`, that takes in two lists of numbers, `numsA` and `numsB`, and returns the number of elements of `numsA` that are equal to the elements of `numsB`. In each of `numsA` and `numsB`, all its elements are different from each other. The length of `numsA` may not be equal to that of `numsB`.

Examples:

```
const numsA = list(23, 21, 30, 15, 40);
const numsB = list(3, 40, 15, 20 );
num_of_matches(numsA, numsB);
// returns 2

const numsA = list(23, 21);
const numsB = list(5, 4, 7);
num_of_matches(numsA, numsB);
// returns 0
```

2D. [4 marks]

The prize won (or not won) for a TOTO ticket is determined by its ***winning group number***. The winning group number is in turn determined by how many numbers on the ticket are equal to the numbers in the winning set and the extra number. Here are the rules:

- Winning Group 1 — n numbers on the ticket match the winning set.
- Winning Group 2 — $n - 1$ numbers on the ticket match the winning set, and one number on the ticket matches the extra number.
- Winning Group 3 — $n - 1$ numbers on the ticket match the winning set.
- Winning Group 4 — $n - 2$ numbers on the ticket match the winning set, and one number on the ticket matches the extra number.
- Winning Group 5 — $n - 2$ numbers on the ticket match the winning set.
- Winning Group 0 — otherwise.

Write a function, `check_winning_group(bet_nums, draw_nums, extra_num)`, that takes in the list of numbers on the ticket, `bet_nums`, the list of numbers in the winning set, `draw_nums`, and the extra number, `extra_num`, and returns the winning group number of the ticket.

Examples:

```
const bet_nums = list(40, 30, 1, 49, 23, 15);
const draw_nums = list(23, 1, 30, 15, 40, 49);
const extra_num = 27;
check_winning_group(bet_nums, draw_nums, extra_num);
// returns 1

const bet_nums = list(40, 30, 1, 49, 27, 15);
const draw_nums = list(23, 1, 30, 15, 40, 49);
const extra_num = 27;
check_winning_group(bet_nums, draw_nums, extra_num);
// returns 2
```

Question 3: Binary Arithmetic Expressions [20 marks]

A ***Binary Arithmetic Expression (BAE)*** is either a *number* or the expression `(` ⟨*bae*⟩ ⟨*op*⟩ ⟨*bae*⟩ `)`, where each ⟨*bae*⟩ is a BAE and ⟨*op*⟩ is the binary operator `+`, `-`, `*`, or `/`. The followings are examples of BAE:

- `123`
- `( 56 + 23 )`
- `( ( 2 + 5 ) * 100 )`
- `( ( 10 / 2 ) - ( 3 * 4 ) )`

BAEs represent arithmetic expressions that we are all familiar with, except that in BAEs, a pair of parentheses is always used to surround every binary arithmetic operation. As a result, we do not need to be concerned with operator precedence and associativity.

3A. [6 marks]

We want to represent BAEs as ***BAE-trees*** in Source programs. A BAE-tree is either a *number* or a list that has 3 elements where the first element is a BAE-tree, the second element is a string `"+"`, `"-"`, `"*"` or `"/"`, and the third element is a BAE-tree. The first and third elements are the left and right operands of the binary arithmetic operation, respectively.

For example, the BAE `( ( 2 + 5 ) * 100 )` has the following BAE-tree:

```
list( list(2, "+", 5), "*", 100 );
```

Write a function, `evaluate_BAE_tree(bae_tree)`, that takes in a valid BAE-tree, `bae_tree`, and evaluates it to a single numeric value. You can assume that division by 0 will not occur for the given input.

Examples:

```
const bae_tree = 123;
evaluate_BAE_tree(bae_tree);
// returns 123

const bae_tree = list( list(2, "+", 5), "*", 100 );
evaluate_BAE_tree(bae_tree);
// returns 700
```

3B. [7 marks]

We want to have a function to construct BAE-trees for BAEs. A BAE is first represented as a ***BAE-list***, which is simply a list of lexical tokens in the BAE, in the same order as they appear in the BAE. For example, the BAE `( ( 2 + 5 ) * 100 )` has the following BAE-list:

```
list( "(", "(", 2, "+", 5, ")", "*", 100, ")" );
```

Write a function, `build_BAE_tree(bae_list)`, that takes in a valid BAE-list, `bae_list`, and returns the corresponding BAE-tree.

Examples:

```
const bae_list = list(123);
build_BAE_tree(bae_list);
// returns 123

const bae_list = list("(", "(", 2, "+", 5, ")", "*", 100, ")");
build_BAE_tree(bae_list);
// returns a result equal to
// list( list(2, "+", 5), "*", 100 )
```

3C. [1 mark]

Write a function, `evaluate_BAE(bae_list)`, that takes in a valid BAE-list, `bae_list`, and evaluates the corresponding BAE to a single numeric value.

Examples:

```
const bae_list = list(123);
evaluate_BAE(bae_list);
// returns 123

const bae_list = list("(", "(", 2, "+", 5, ")", "*", 100, ")");
evaluate_BAE(bae_tree);
// returns 700
```

3D. [6 mark]

This question is not about BAE; it is about matching parentheses.

A ***parenthesis expression*** is an expression made of opening parenthesis "(" and closing parenthesis ")". For example, `(()(())())()` is a parenthesis expression. In a ***valid parenthesis expression***, every "(" must have a matching ")" on its right, and every ")" must have a matching "(" on its left. Each "(" must match only one ")", and each ")" must match only one "(".

For example, here are some ***valid*** parenthesis expressions:

- `()`
- `()()`
- `(())`
- `(()(()))()()`
- ← (*empty parenthesis expression is considered valid*)

Here are some ***invalid*** parenthesis expressions:

- `)(`
- `(()(`
- `(()))`

In Source, we represent a parenthesis expression simply as a list of `"("` and `")"`, in the same order as they appear in the parenthesis expression. For example, the parenthesis expression `(())` has the list representation `list("(", "(", ")", ")")`.

Write a function, `check_parentheses(paren_list)`, that takes in the list representation of a parenthesis expression, and returns true if the parenthesis expression is valid, otherwise it returns false.

Examples:

```
const paren_list = list();
check_parentheses(paren_list);
// returns true

const paren_list = list("(", "(", ")", ")");
check_parentheses(paren_list);
// returns true

const paren_list = list("(", "(", ")", "(");
check_parentheses(paren_list);
// returns false
```

——— END OF QUESTIONS ———