ARIEL UNIVERSITY

Ariel University
School of Computer Sciences
Programming Languages Course
Lecturer: Asaly Saed
Assignment #1 – Introduction, **Due to: 04/02/2024, 23:55**

Administrative:

The purpose of this homework is to familiarize you with the various tools that will be used throughout the course, and to get a feeling of basic programming in Racket (pl). In this particular homework, you will generally be graded on contracts (declarations) and purpose statements, other comments, style, test quality, etc. Correctness will play a very small role here, since everyone is expected to be able to solve these questions. The first thing you will need to do is to download and install Racket and then the course plugin.

For this problem set, you are required to set the language level to the course's language, by beginning you file with `#lang pl`.

This homework is for individual work and submission.

The code for all the following questions should appear in a single .rkt file named <your ID>_1.rkt (e.g., 333333333_1.rkt for a student whose ID number is 333333333).

Do not submit multiple files. Do not compress your file.

Integrity: Please do not cheat. You may consult your friend regarding the solution for the assignment. However, you must do the actual programming and commenting on your own!! This includes roommates, marital couples, best friends, etc… I will be very strict in any case of suspicion of plagiary. Among other thing, students may be asked to verbally present their assignment.

Comments: Submitted code for each question should include **at least two lines of comments with your personal description of the solution**, the function and its type. In addition, **you should comment on the process of solving this question** – what were the main difficulties, how you solved them, how much time did you invest in solving it, did you need to consult others. **A solution without proper comments may be graded 0.**

ARIEL UNIVERSITY

Tests: For each question, you should have enough test cases for complete coverage (DrRacket indicates covered expressions with colors for covered and uncovered source code, unless your code is completely covered). See below on the way to create tests.

Important: Your tests should cover your whole code; otherwise the grading system will heavily penalize your submission. You should not have any uncovered expressions after you hit "Run" — it should stay at the same color, indicating complete coverage. Furthermore, the grading system will run its own tests over your code, which means that you will not be able to submit code that does not work. Reminder: this means that most of the focus of this homework is put on the contract and purpose statements, good style (indentation, comments, etc), and good tests.

General note: do not duplicate code! If there is an expression that is used in multiple places, then you should use `let`.

The language and how to form tests: In this homework (and in all future homework) you should be working in the "Module" language, and use the appropriate language using a `#lang` line. You should also click the "Show Details" button in the language selection dialog, and check the "Syntactic test suite coverage" option to see parts of your code that are not covered by tests: after you click "run", parts of the code that were covered will be colored in green, parts that were not covered will be colored in red, and if you have complete coverage, then the colors will stay the same. Note that you can also set the default language that is inserted into new programs to `#lang pl`, to make things more convenient. There are some variants for the `pl` language for various purposes — in particular, `#lang pl untyped` will ignore all type declarations, and will essentially run your code in an untyped Racket.

The language for this homework is:

```
#lang pl
```

This language has a new special form for tests: `test`. It can be used to test that an expression is true, that an expression evaluates to some given value, or that an

expression raises an error with some expected text message. For example, the three kinds of tests are used in this example:

```
(define (safe-length list)
  (cond [(null? list) 0]
        [(pair? list) (add1 (safe-length (rest list)))]
        [else (error 'safe-length "bad value: ~s" list)]))
(test (zero? (safe-length null)))
(test (safe-length '(1 2 3)) => 3)
(test (safe-length "three") =  error> "bad value")
```

In case of an *expected* error, the string specifies a pattern to match against the error message. (Most text stands for itself, "`?`" matches a single character and "`*`" matches any sequence of characters.)

Note that the `=error>` facility checks only errors **that *your* code throws**, not Racket errors. For example, the following test **will not** succeed (because it is an error thrown by Racket):

```
(test (/ 4 0) =error> "division by zero")
```

Reminder: code quality will be graded. Write clean and tidy code.

ARIEL UNIVERSITY

Question 1:

Define the function `create-fixed-length-lists` – that consumes that consumes a list of numbers and an Integer `'n'` returns a list of lists where each sublist contains **`n`** elements from the original list, in the same order they appear in the original list. If the total number of elements in the original list is not a multiple of **`n`**, the final sublist may contain fewer than **`n`** elements.

Here is another example, written in a form of a test that you can use:

```
(test (create-fixed-length-lists '(1 2 3 4 5 6 7 8 9) 3) => '((1 2
3) (4 5 6) (7 8 9)))

(test (create-fixed-length-lists '(1 2 3 4 5 6 7 8 9) 9) => '((1 2 3
4 5 6 7 8 9)))
```

Question 2:

2.a. Define a function `nested-list-depth` that calculates the maximum depth of nesting in a list that may contain any type. The function should use recursion to traverse the list and determine the depth of the deepest nested list. Ensure to include a clear contract, purpose statement, and comprehensive tests.

For example, written in a form of a test that you can use:

```
(test (nested-list-depth '(1 (2 3) ((4)) (5 (6)))) => 3)

(test (nested-list-depth '(1 2 3)) => 1)

(test (nested-list-depth '()) => 0)
```

ARIEL UNIVERSITY

2.b. Define a `min&max-lists` function that consumes a list of lists (where the type of the elements in the inner list may be any type). The function returns a list of lists – such that for each inner list `lst` (in the original list) the following is done – 1. If `lst` contains at least one number, then `lst` is replaced with a list of size two, containing the minimum and maximum in `lst`, and 2. Otherwise, `lst` is replaced with a `null`.

For example, written in a form of a test that you can use:

```
(test (min&max-lists '((any "Benny" 10 OP 8) (any "Benny" OP (2
3)))) => '((8 10) ()))
(test (min&max-lists '((2 5 1 5 L) (4 5 6 7 3 2 1) ()))
        => '((1 5) (1 7) ()))
```

Hints: 1. Use the most exact type for the declaration of the function. 2. You may want to define a helper function that deals with a single inner list. 3. You may want to use the built in Racket map, apply, and min.

Question 3:

Create a data structure called `TaggedQueue`. Each element in this queue is tagged with a unique identifier (ID). You are required to implement the following operations, with guidance provided below:

1. Implement Empty Queue (`EmptyTQ`): Define a constructor for an empty `TaggedQueue`.

2. Implement Enqueue Operation (`Enqueue`): This function should accept an ID (symbol), a value (any type), and an existing `TaggedQueue`, returning a new `TaggedQueue` with the element added.

ARIEL UNIVERSITY

3.Implement Search Operation (`search-queue`): This function should take an ID and a `TaggedQueue` as inputs, returning the value associated with the first occurrence of the ID in the queue, or `#f` if not found.

4. Implement Dequeue Operation (`dequeue-queue`): This function should remove the first element from the `TaggedQueue` and return the modified queue. If the queue is empty, it should return `#f`.

The implementation should include proper type definitions and test cases for each operation. For example, written in a form of a test that you can use:

```
(test (EmptyTQ) => (EmptyTQ))

(test (Enqueue 'x 42 (EmptyTQ)) => (Enqueue 'x 42 (EmptyTQ)))

(test (search-queue 'x (Enqueue 'x 42 (EmptyTQ))) => 42)

(test (dequeue-queue (Enqueue 'x 42 (EmptyTQ))) => (EmptyTQ))
(test (dequeue-queue (EmptyTQ)) => #f)
```

Question 4:

Design a BNF grammar for a small programming language that supports string concatenation and variable assignments. Your BNF should include the following components:

1. Variable Assignment: Variables can be assigned string values.
2. String Concatenation: Strings can be concatenated using a specific operator.
3. Variables and Strings: Define what constitutes a valid variable name and string format.
- Additionally, provide examples of valid statements in your language.

For example:

- "x = "hello"" - Assigns the string "hello" to variable x.
- "y = "world"" - Assigns the string "world" to variable y.
- "z = x ++ y" - Concatenates the strings in variables x and y and assigns the result to z.

ARIEL UNIVERSITY

Your task is to formalize these rules into a coherent BNF structure, ensuring clarity and unambiguity in the grammar. Consider how variables are declared, how strings are defined, and how concatenation operations are handled in terms of precedence and associativity.

Good Luck!