ARIEL UNIVERSITY

Ariel University
Computer Sciences Department
Programming Languages Course
Assignment #2 – BNFs, Higher-Order Functions, Typed Racket

Out: *Wednesday, May 11, 2022*, Due: ***Thursday, May 26, 11:55 am***

Administrative

This is another introductory homework, and again it is for **work and submission in pairs**. In this homework you will be introduced to the course language and some of the additional class extensions.

In this homework (and in all future homeworks) 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 02
```

As in previous assignment, you need to use the special form for tests: `test`.

Reminders (this is more or less the same as the administrative instructions for the previous assignment):

This homework is for work and submission in pairs.

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

ARIEL UNIVERSITY

strict in any case of suspicion of plagiary. Among other thing, students may be asked to verbally present their assignment.

Important – If you consult any person (other than your partner) or any other source – You must indicate it within your personal comments. Also the contribution of each of you to the solution must be detailed.

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.

VERY IMPORTANT NOTE:

A solution without proper and elaborate PERSONAL describing your work process comments may be graded 0.

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 server 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 server 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: Code quality will be graded. Write clean and tidy code. Consult the Style Guide, and if something is unclear, ask questions on the course forum.

The test form can be used to test that an expression is true, that an expression evaluates to some given value, or that an expressions raises an error with some expected text message. For example, the three kinds of tests are used in this example:

```
#lang pl
(: smallest : (Listof Number) -> Number)
(define (smallest l)
  (match l
```

ARIEL UNIVERSITY

```
    [(list)      (error 'smallest "got an empty list")]
    [(list n)    n]
    [(cons n ns) (min n (smallest ns))]))
(test (smallest '(5 7 6 4 8 9)) => 4)
(test (zero? (smallest '(0 1 2 3 4))))
(test (smallest '()) =error> "got an empty list")
```

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:

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

The code for all the following questions should appear in a single .rkt file named <your ID>_1 (e.g., 222222222_333333333_1 for two students whose ID numbers are 222222222 and 333333333).

1. BNF

a. In class we have seen the grammar for AE — a simple language for "Arithmetic Expressions".

   Write a BNF for "**LE**": a similarly simple language of "List Expressions". Valid 'programs' (i.e., word in the **LE** language) in this language should correspond to Racket expressions that evaluate to S-expressions holding numbers and symbols. The valid functions that can be used in these expressions are **`cons`**, **`list`**, and **`append`**, and **`null`** is a valid expression too. Note that the grammar should allow only **`cons`** with an expression that represents a list (no numbers or symbols as the second part of a cons expression), **`list`** can have any number of arguments (including zero arguments), and **`append`** can have any number of *list* arguments. [A quick rule-of-thumb for the **`cons`** restriction: if an expression works fine in the course language, then it should be in **LE**.] Plain values are either numbers or quoted symbols — no quoted lists, and only the quote character can be used. You can use ⟨`num`⟩ and ⟨`sym`⟩ as known numbers and symbols terminals. (Note that we assume that ⟨`sym`⟩ corresponds to a Racket symbol, and that *does not* include the quote (`'`) character, i.e., the quote (`'`) character should

appear in your BNF.)
For example, some **valid** expressions in this language are:

```
null
12
'boo
(cons 1 (cons 'two null))
(list 1 2 3)
(list (list (list (list 1 2 3))))
(append (list 1 2) (list 3 4) (list 5 6))
(list)
(append)
(cons 1 (cons (append (cons 'x null) (list 'y 'z)) null))
```

but the following are **invalid** expressions:

```
(cons 1 2)
(list (3 4))
(quote boo)
(append 1 (list 2 3) 4)
(cons 1 2 null)
(cons 1 (2))
(cons 1 '())
'(1 2 3)
(cons '1 null)
(list ''a)
(car (list 1 2))
```

b. Note that some of these expressions, if taken as Racket expressions, evaluate to valid S-expression values — but this question specifies a *restricted* set of expressions.
For `list` and `append` you will need to use ellipsis (`...`) to specify zero-or-more occurrences of the previous piece of syntax.

ARIEL UNIVERSITY

c. **Important remark:** Your solution should only be a BNF and not a code in Racket (or in any other language). You cannot test your code!!! Indeed, your answer should appear inside a comment block (write the grammar in a #|---|# comment).

d. Add to your BNF a derivation process for 3 different **LE** expressions, in which all plain values are sub-words (of length 2 or 3) of either your name or ID number. E.g., if your name is Baba Ganush and your ID number is 123456789, then you might want to show how you derive the word **(cons 345 (cons (append (cons 'Bab null) (list 'Gan 'sh)) null))** from your BNF (make sure you use the full power of your BNF). To do so, you simply mark each derivation rule by an index .and use "=>(i)" to state that in a certain step, you have used rule number i of your BNF.

2. Simple Changes to the Syntax and Semantics of AE

2 Moving to Infix Syntax

In class, we have seen the implementation of the AE language. (The code is available as a single file in the Interpreters Section.) We mentioned the fact that we're using an intermediate S-expression format for input code, and that this helps in making the syntax and the semantics independent. To see this in action, modify the AE implementation in two ways, and notice how each change affects different parts of the code:

- **Make the language use *infix* syntax** (still fully parenthesized, for simplicity). You will need to change both the BNF definition (the topmost comment) and the code in the parser. Make sure you properly comment your change to the parser. (Hint: this is extremely easy; if you find yourself writing new code, then you're probably off track.)

- **Allowing division by zero.** As it stands, the AE evaluator reflects Racket's behavior for *all* arithmetics. For example, if you try to evaluate {5 / 0}, you get a Racket "division by zero" error. 'Fix' this by returning infinity from such divisions, and to make things easy, use 999 as the value of infinity.

  For this question, copy the AE evaluator code in one chunk, including the tests at the end (but get rid of the #lang line, since you already have one). Note that to be able to submit your solution, you will need to add enough test cases that cover the code completely.

ARIEL UNIVERSITY

3. Higher Order Functions

As you already know, lists are a fundamental part of Racket. They are often used as a generic container for compound data of any kind. It is therefore not surprising that Racket comes with plenty of useful functions that operate on lists. One of the most useful list functions is **`foldl`**: it consumes a *combiner* function, an *initial* value, and an input list. It returns a value that is created in the following way:

- For the empty list, the initial value is returned,
- For a list with one item, it uses the combiner function with this item and the initial value,
- For two items, it uses the combiner function with the first and the result of folding the rest (a one-item list),
- etc.

In the general case, the value of **`foldl`** is:

```
(foldl f init (list x1 x2 x3 ... xn))
  = (f xn (... (f x3 (f x2 (f x1 init)))))
```

Note that **`foldl`** is a *higher-order* function, like **`map`**. Its type is:

```
(: foldl : (All (A B) (A B -> B) B (Listof A) -> B))
```

Use **`foldl`** together with (or without) **`map`** to define a `sum-of-squares` function which takes a list of numbers as input, and produces a number which is the sum of the squares of all of the numbers in the list. A correct solution should be a one-liner. Remember to write a proper description and contract line, and to provide sufficient tests (using the `test` form). You will need to do this for a definition of `square` too, which you would need to write for your implementation of `sum-of-squares`.
A more detailed explanation on both functions can be found at the bottom of the assignment or here.

Here is an example of a test that you might want to perform:

```
(test (sum-of-squares '(1 2 3)) => 14)
```

ARIEL UNIVERSITY

4. Typed Racket (and more H.O. functions)

a. We define a *binary tree* as a something that is either a `Leaf` holding a number, or a `Node` that contains a binary tree on the left and one on the right. Write a `define-type` definition for `BINTREE`, with two variants called `Node` and `Leaf`.

b. Using this `BINTREE` definition, write a (higher-order) function called `tree-map` that takes in a numeric function `f` and a binary tree, and returns a tree with the same shape but using `f(n)` for values in its leaves. For example, here is a test case:

```
(test (tree-map add1 (Node (Leaf 1) (Node (Leaf 2) (Leaf
3))))
      => (Node (Leaf 2) (Node (Leaf 3) (Leaf 4))))
```

c. Remember: correct type, purpose statement, and tests.

d. Continue to implement `tree-fold`, which is the equivalent of the swiss-army-knife tool that `foldl` is for lists. There are two differences between `tree-fold` and `foldl`: **first**, `foldl` has a single `init` argument that determines the result for the single empty list value. But with our trees, the base case is a `Leaf` that holds some number — so we use a (numeric) *function* instead of a constant. **The second** difference is that `tree-fold`'s combiner function receives different inputs: the two results for the two subtrees. `tree-fold` should therefore consume three values — the combiner function (a function of two arguments), the leaf function (a function of a single number argument), and the `BINTREE` value to process. Note also that `tree-fold`'s type is slightly simpler than `foldl`'s type — because we have only trees of numbers — but don't confuse that with the output type, which *does not* have to be a number (or a list of numbers).
Note: `tree-fold` is a *polymorphic* function, so its type should be parameterized over "some input type `A`". The Typed Racket notation for this is

```
(: tree-fold : (All (A) ...type that uses A...))
```

Hint: The type A is also the type of the returned value of `tree-fold (as opposed to foldl, here, no extra polymorphic type B is required, since the 'inner type' of a Leaf is always Number).`

e. When your implementation is complete, you can use it, for example, to implement a `tree-flatten` function that consumes a `BINTREE` and returns a list of its values from left to right:

```
(: tree-flatten : BINTREE -> (Listof Number))
;; flattens a binary tree to a list of its values in
;; left-to-right order
(define (tree-flatten tree)
  (tree-fold (inst append Number) (inst list Number) tree))
```

f. You can use this function, as well as others you come up with, to test your code. Note the use of `(inst` *f* `Number)` — the Typed Racket inference has certain limitations that prevent it from inferring the correct type. We need to 'help' it in these cases, and say explicitly that we use the two polymorphic functions `append` and `list` instantiated with `Number`. (Think about an `(All (A) ...)` type as a kind of a function at the type world, and `inst` is similar to calling such a function with an argument for `A`.) You will not need to use this elsewhere in your answers.

g. Use `tree-fold` to define a `tree-reverse` function that consumes a tree and returns a tree that is its mirror image. That is, for any tree *t*, the following equation holds:

```
(equal? (reverse (tree-flatten t))
        (tree-flatten (tree-reverse t)))
```

You can use it in your tests (but use `test` for the test). If you do things right, this should be easy using a one-line helper definition (you may call it `switch-nodes`).

To recap: In this question you were asked to define the **`BINTREE`** type, and the following procedures **tree-map**, **tree-fold**, and **tree-reverse**. Don't forget to add comments and tests.

On the procedures map and fold-l

הפונקציה map:

ARIEL UNIVERSITY

קלט: פרוצדורה proc **ורשימה** lst

פלט: רשימה שמכילה אותו מספר איברים כמו ב- lst **– שנוצרה ע"י הפעלת הפרוצדורה** proc **על כל אחד מאיברי הרשימה** lst. **(ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה** map **יכולה לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)**

(map proc lst ...+) → list?
proc : procedure?
lst : list?

Applies proc to the elements of the lsts from the first elements to the last. The proc argument must accept the same number of arguments as the number of supplied lsts, and all lsts must have the same number of elements. The result is a list containing each result of proc in order.

דוגמאות:

```
> (map add1 (list 1 2 3 4))
'(2 3 4 5)
> (map (lambda (x) (list x))
     '(sym1 sym2 33))
'((sym1) (sym2) (33))
```

הפונקציה foldl:

קלט: פרוצדורה proc, **ערך התחלתי** init **ורשימה** lst

פלט: ערך סופי (מאותו טיפוס שמחזירה הפרוצדורה proc**) שנוצר ע"י הפעלת הפרוצדורה** proc **על כל אחד מאיברי הרשימה** lst **תוך שימוש במשתנה ששומר את הערך שחושב עד כה – משתנה זה מקבל כערך התחלתי את הערך של** init. **(ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה** foldl **יכולה לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)**

(foldl

proc init lst ...+) → any/c
proc : procedure?
init : any/c
lst : list?

Like map, foldl applies a procedure to the elements of one or more lists. Whereas map combines the return values into a list, foldl combines the return values in an arbitrary way that is determined by proc.

דוגמאות:

```
> (foldl + 0 '(1 2 3 4))
10
> (foldl cons '() '(1 2 3 4))
'(4 3 2 1)
```

Enjoy and Good Luck!