Problem Set 4
COMP301 Fall 2021
Week 6: 01.11.2021 - 05.11.2021

Instructions:

- Submit your answers to the Blackboard PS4 assignment until November 6th Saturday, at 23.59.
- Please use the code boilerplate, which includes several tests for you to see if your code is correct.
- Submit your code to BlackBoard as *yourIDno_username.rkt* (scm extension is also fine).(Example: *123456_fbulgur17.rkt*)

Problem 1: In this problem, you will re-implement the environment with a slightly different representation called a-list or association-list representation.

Part A. In this representation, variable and value pairs are stored as actual pairs. The environment looks like figure 1. In the code given, implement the missing parts of the procedures **extend-env**, **apply-env**, **empty-env?** and **has-binding?** using a-list representation.

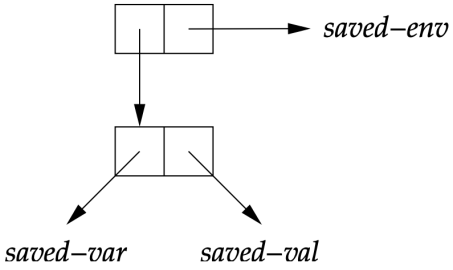

FIGURE 1. a-list representation of an environment

- **empty-env?** : Returns true if the given environment is an empty one, otherwise returns false.
- **has-binding?** : Returns true if the given search-var has an associated value in the environment, otherwise returns false.

Part B. In this part, you will implement a new constructor **extend-env*** using a-list representation which takes list of variables, list of values of the same length, an environment and extends the environment with variable value pairs. Ex:

```
(extend-env* '(a b) (list 1 2) (empty-env))
; returns '(extend-env (b 2) (extend-env (a 1) (empty-env)))
```

Part C. This new procedure **extend-env*** is good but it requires time proportional to given association amount. With a small change in the a-list representation this procedure can be run in constant time. In this new representation which is called ribcage, instead of extending the environment with one variable and one value pair, we extend it with list of variables and list of associated values pair which looks like figure 2.

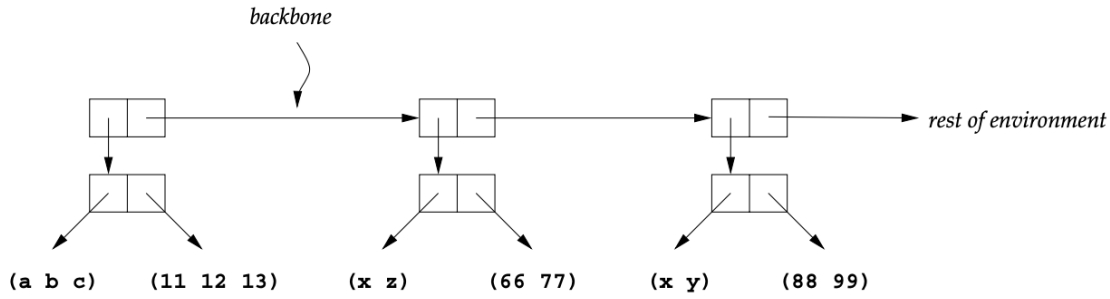

FIGURE 2. ribcage representation of an environment

- The procedure **empty-rib-env** is already given in the code.
- Implement procedures **extend-rib-env*** and **apply-rib-env**. Ex:

```
(extend-rib-env* '(c) (list 6)
   (extend-rib-env* '(a b) (list 1 2) (empty-rib-env)))
; returns '(extend-rib-env ((c)(6))(extend-rib-env ((a b)(1 2))
                                    ;(empty-rib-env)))
```

Hint: It is not necessary but knowing how to use let* and/or letrec may help in your implementation of this part.

Problem 2: In this section, you will define a recursive data type by using define-datatype and use cases expression to manipulate them.

Part A. Implement the missing parts of the stack data type. Stack is given by the following grammar.

Stack ::= () | (Int *Stack*)

Part B. Implement **add**, **top**, **pop**, **empty?** procedures.

- **add** : Takes a value and stack and returns a new stack with new value added.
- **top** : Takes a stack and returns the value at the top. Does not pop the value.
- **pop** : Takes a stack and returns the stack with the value at the top is popped.
- **empty?** : Takes a stack and checks if it is empty.

Part C. Given the grammar of a bintree implement the missing parts in the define-datatype in the code.

Bintree ::= (Int) | (Symbol *Bintree* *Bintree*)

Part D. Implement two procedures **bintree-to-list** and **sum-leafs**.

- **bintree-to-list** : Takes a bintree data type and returns its list version. Ex:

```
(bintree-to-list (interior-node 'a (leaf-node 3) (leaf-node 4)))
; returns '(interior-node a (leaf-node 3) (leaf-node 4))
```

- **sum-leafs** : Takes a bintree and sums all of the values of its leaves. In the case of the example above, it is 7.