Assignment 3—version 0.0

Date of Submission: **15th April, Wednesday**

1 Introduction

This assignment is about implementing the environmental model of execution. At the end of the assignment, you will learn how a language with several interesting features is implemented. Let us give this language the name *myracket*. The implementation is not efficient—efficiency will be the subject of the course CS 302. However, it will teach you the principles behind the implementation of *myracket*.

2 Description of *myracket*

First let us see how the language being implemented will be represented. Remember that you are about to write a program for which the data is a program in *myracket*. Therefore we have to choose a suitable representation for programs in *myracket*. This is done through a series of `structs` that you will find in the file called `defs.rkt`.

```
(struct pgm (deflist) #:transparent)
(struct def (var/fun exp) #:transparent)
```

A program is a list of definitions, of which one of the definitions defines a special symbol `main`. You should write you *myracket* programs in such a way that, at the end of execution, `main` will have the result of the program. Each `def` binds a variable or a function name to an expression. A `deflist` is simply a list of `defs` so that nothing needs to be defined for it. However one needs to define `exp`. Going further:

```
;A variable is an expression
;A constant is an expression
;A string is an expression.
;A list is an expression
(struct uexp (op exp) #:transparent); op = car, cdr
(struct bexp (op exp1 exp2) #:transparent); op = cons, +, -, *, <, =, <=
(struct iff (cond exp1 exp2) #:transparent)
(struct app (fun explist) #:transparent)
```

```
(struct lam (varlist exp) #:transparent)
(struct sett (var exp) #:transparent)
(struct lett (deflist exp2) #:transparent)
(struct lets (deflist exp2) #:transparent)
(struct beginexp (explist) #:transparent)
(struct defexp (deflist exp) #:transparent)
(struct debugexp () #:transparent)
```

As you can see, expressions consist of *symbols*, *constants*, *strings*, *lists*, *unary* and *binary expressions* with built-in operators, *ifs*, *applications*, *lambdas*, *lets*, *begin expressions* and *define expressions*. In addition, there is a debug expression (`debug`), that we shall talk about later. Note that there are certain kinds of expressions that are in the intersection of *myracket* and *drracket*. These are strings, numbers, booleans, strings and operators. This is to make the assignment easy.

Consider the program:

```
(define (f g x) (g (* x x)))
(define x 4)
(define (h y) (+ x y))
(define main (f h 5))
```

Using the `structs` defined above, this program would be represented as follows:

```
(define prog1
  (pgm (list
    (def 'f (lam (list 'g 'x) (app 'g (list (bexp * 'x 'x)))))
    (def 'x 4)
    (def 'h (lam (list 'y) (bexp + 'x 'y)))
    (def 'main (app  'f (list 'h 5))))))
```

The file `examples.rkt` contains ten examples that cover all those discussed in the class.

3 Representation of stack, frames, and closures

From the discussion on the *environment model* of execution, we know that as the program is running, it needs to create *closures* and *frames*. We also need a *stack* to keep track of the current environment. We thus need the following definitions:

```
(struct frame (number bindings parent)  #:transparent)
(struct closure (lambda frame) #:transparent)
```

Each frame represents a local environment. It consists of a frame number, a sequence of bindings of symbols to their values, and the frame number of its parent frame. The parent frame is the

begining of a chain of framcreateframees representing the global environment. The bindings should be implemented through a mutable hash table (Note: Read mutable hash tables). A closure is a lambda along with with an environment that helps in finding the values of the global symbols in the lambda. In this case also the environment is represented by the frame number of the beginning of a chain of frames. Finally, a stack is a list of frames, the top of the stack holds the frame beginning the current environment.

If you see the file `defs.rkt`, you will notice that the `structs`'s for closure and frame don't look like what has been shown above. Instead, they seem to contain a lot of things after the `#:transparent`. The extra code is to ensure that when you display frames (by a `display` or `displayln`), the output is easy to read and understand. For example, in my implementation, just before the execution of the body of `h` (i.e. just before the execution of `(+ x y)`), ... the environment would look like this

```
  @@@@@@@@@@@@@@@@@@@@@@@@@
#<Frame: Number: 2
 Bindings:
   y --> 25
 Parent:  0>
@@@@@@@@@@@@@@@@@@@@@@@@@
#<Frame: Number: 0
 Bindings:
   x --> 4
   f --> #<: #<lam: (g x) #<app: g (#<+: x x>)>> Environment:  0>
   h --> #<: #<lam: (y) #<beginexp: (#<debugexp> #<+: x y>)>> Environment:  0>
 Parent:  #<emptyframe: >>
@@@@@@@@@@@@@@@@@@@@@@@@@
```

The environment starts with the frame numbered 2. This is the frame of `h` with the parameter `y` which has a value of 25. The parent of this frame is frame 0. This has bindings for `x`, `f` and `h` but not `main`. To display the current environment, I have a special expression called `(debugexp)`. If you insert `(debugexp)` at a certain point in the program, it will show you the complete environment at that point. This, I hope, will be useful for you in debugging your program.

4 A summary of the implementation rules

Here I shall summarize the rules of the implementation that we have discussed many times in the class.

1. `defs` extend the current frame by the new things being defined. Each definition binds the symbol being defined by its value.
2. The value of a symbol is whatever it is bound to.

3. The value of a constant is its value in *drracket*.
4. The value of a unary or a binary operator is the corresponding racket operator applied to the values of its arguments.
5. The value of a `lam`bda is a `closure`.
6. To find value of an `app`lication of a function to a list of arguments a new frame is created with the parameters of the function bound to the actual arguments. The new frame becomes the current environment after the parent of the frame is set to the environment packed in the closure of the function being applied.
7. A `lett` also creates a new frame with the current frame as parent. A `lets` can be translated to individual `lett`s.
8. A `sett` is an assignment.
9. You know what `beginexp` and `defexp` do.
10. `debugexp` is being given to you in an implemented form.

5 What you have to do

As usual, there will be the following files:

1. Files called `defs.rkt` and `examples.rkt` mentioned before.
2. A signature file call `my-interpreter.rkt` which will have unfilled functions with brief descriptions of what they do. This is the file that you have to fill.
3. A trial file called `try.rkt`, in which you can try out different functions in my model implementation. Everything in the model implementation is made visible to you in `try.rkt`.
4. What you have to submit is the file `my-interpreter.rkt`, with each definition completed.