CmpE 260 - Principles of Programming Languages
Spring 2023 - Project 2

Deadline: 25 May 2023 17:00

Assistants: Alper Ahmetoğlu, Ali Nasra
ahmetoglu.alper@gmail.com, ali.nasra@boun.edu.tr

1 Introduction

In this project, you are going to implement an interpreter for an extension of Racket language. The main functionality of this language is that you can set values to variables and use them in expressions in a procedural fashion. Variables that are bound to values are hold in a state (a hash), and the state changes at each iteration of the program. Below are some examples from the language:

Example 1

```
(:= a 5)
(:= b (cdr (list 1 2 3)))
(:= c a)
(:= d (* 3 4))
(:= e (* 2 d))
(:= f 3)
(:= f (/ f e))
(:= pi-num 3.14)
(:= area
    (func (x type) (
            (:= square (* x x))
            (if: (eq? type "circle")
                (
                    (:= a (* pi-num square))
                )
                (
                    (if: (eq? type "triangle")
                        (
                            (:= k (/ (sqrt 3) 4))
                            (:= a (* k square))
                        )
                        ( ; else it is a square
                            (:= a square)
```

```
                        )
                    )
                )
            )
        )
    )
)

(printf "a = ~a\n" a)
(printf "b = ~a\n" b)
(printf "c = ~a\n" c)
(printf "d = ~a\n" d)
(printf "e = ~a\n" e)
(printf "f = ~a\n" f)
(printf "pi-num = ~a\n" pi-num)
(printf "circle area = ~a\n" (area 5 "circle"))
(printf "triangle area = ~a\n" (area 5 "triangle"))
(printf "square area = ~a\n" (area 5 "square"))
```

The output of this program is:

```
a = 5
b = (2 3)
c = 5
d = 12
e = 24
f = 1/8
pi-num = 3.14
circle area = 78.5
triangle area = 10.825317547305483
square area = 25
```

Example 2

```
(:= fact 1)
(:= n 5)
(while: (> n 1)
    (
        (:= fact (* fact n))
        (:= n (- n 1))
    )
)
(printf "~a" fact)
```

The output of this program is:

```
120
```

The state of the program

The state is a hash consisting of variable names and their values. The values can be numbers, strings, booleans, Racket functions, or functions defined in this language. The state is updated at each evaluation of an expression. There is a special variable named `-r` that holds the result of the last expression. For example, for the first 9 expressions in the first example above, the state is updated as follows:

```
'#hash()
'#hash((-r . 5) (a . 5))
'#hash((-r . (2 3)) (a . 5) (b . (2 3)))
'#hash((-r . 5) (a . 5) (b . (2 3)) (c . 5))
'#hash((-r . 12) (a . 5) (b . (2 3)) (c . 5) (d . 12))
'#hash((-r . 24) (a . 5) (b . (2 3)) (c . 5) (d . 12) (e . 24))
'#hash((-r . 3) (a . 5) (b . (2 3)) (c . 5) (d . 12) (e . 24) (f . 3))
'#hash((-r . 1/8) (a . 5) (b . (2 3)) (c . 5) (d . 12) (e . 24) (f . 1/8))
'#hash((-r . 3.14) (a . 5) (b . (2 3)) (c . 5) (d . 12) (e . 24) (f . 1/8)
       (pi-num . 3.14))
'#hash((-r . #<procedure:...>) (a . 5) (area . #<procedure:...>) (b . (2 3))
       (c . 5) (d . 12) (e . 24) (f . 1/8) (pi-num . 3.14))
```

In this way, the result of the last computation can be always accessed by the variable `-r` (useful for test expressions, function results).

2 Functions

You are going to implement 11 functions that will help you to parse and evaluate statements in this language.

2.1 `empty-state` (5 points)

This is going to be an empty hash representing the state of the program.

Example:

```
> empty-state
'#hash()
```

2.2 `(get state var)` (5 points)

This function takes a state `state` and a variable name `var`, and returns the value of the variable in the state. It should also return the value of the variable if it is a valid Racket function, a number, a string, or a boolean. (Hint: evaluate the variable if it is not in the state.)

Examples:

```
> (get (hash 'a 5) 'a)
5
```

```
> (get (hash 'a 5) '+)
#<procedure:+>
> (get (hash 'a 5) '3)
3
> (get (hash 'a 5) ''a)
'a
> (get (hash 'a 5) 'b)
Error:  variable b is not defined
```
(you don't need to print this error)

2.3 (put state var val) (5 points)

This function takes a state `state`, a variable name `var`, and a value `val`, and returns a new state that is the same as the old state except that the variable is bound to the value.

Examples:

```
> (put (hash 'a 5) 'a 6)
'#hash((a .  6))
> (put (hash 'a 5) 'b 6)
'#hash((a .  5) (b .  6))
```

2.4 (:= var val-expr state) (15 points)

This function takes a variable name `var`, a value expression `val-expr`, and a state `state`, and returns a new state that is the same as the old state except that the variable is bound to the value of the value expression. You should evaluate the value expression with `eval-expr` function that you will implement since it may be a complex expression. The result of the value expression should be stored in the variable `-r`.

Tip: for starters, you can assume that the value expression is a number, a string, a boolean. Then, you can extend your implementation to handle function calls and function definitions. Especially the last examples below will not be complete until you implement the `eval-expr` function.

Examples:

```
> (:= 'a 5 (hash 'b 6))
'#hash((-r .  5) (a .  5) (b .  6))
> (:= 'a '(+ 2 3) (hash 'b 6))
'#hash((-r .  5) (a .  5) (b .  6))
> (:= 'a 'b (hash 'b 6))
'#hash((-r .  6) (a .  6) (b .  6))
> (:= 'a '(+ 2 b) (hash 'b 6))
'#hash((-r .  8) (a .  8) (b .  6))
> (:= 'b '(* b b) (hash 'b 6))
'#hash((-r .  36) (b .  36))
> (:= 'a '+ (hash 'b 6))
```

```
'#hash((-r .  #<procedure:+>) (a .  #<procedure:+>) (b .  6))
> ((get (:= 'square '(lambda (x) (* x x)) (hash 'b 6)) 'square) 5)
25
> ((get (:= 'my-func '(func (x) ((:= x (* 5 x)) (:= y (+ x x)))) (hash 'b 6))
'my-func) 5)
50
```

2.5 (if: test-expr then-exprs else-exprs state) (15 points)

This function takes a test expression `test-expr`, a list of then expressions `then-exprs`, a list of else expressions `else-exprs`, and a state `state`, and returns the result of then expressions if the test expression evaluates to true, and the result of else expressions otherwise. You will evaluate the test expression with `eval-expr` function and evaluate `then-exprs` if `-r` is true in the resulting state after evaluating `test-expr`, and evaluate `else-exprs` otherwise.

Tip: It is again highly suggested that you first implement this function for primitive types, and then extend it to handle function calls and function definitions.

Examples:

```
> (if:  '#t '(1) '(2) empty-state)
'#hash((-r .  1))
> (if:  '#f '(1) '(2) empty-state)
'#hash((-r .  2))
> (if:  '#t '(a) '(b) (hash 'a 5 'b 6))
'#hash((-r .  5) (a .  5) (b .  6))
> (if:  '#f '(a) '((:= a (* a b))) (hash 'a 5 'b 6))
'#hash((-r .  30) (a .  30) (b .  6))
> (if:  '(> a b) '((:= a (* a b))) '((:= b (* a b))) (hash 'a 5 'b 6))
'#hash((-r .  30) (a .  5) (b .  30))
```

2.6 (while: test-expr body-exprs state) (15 points)

This function takes a test expression `test-expr`, a list of body expressions `body-exprs`, and a state `state`. It evaluates the test expression with `eval-expr` function and evaluates the body expressions if `-r` is true in the resulting state after evaluating `test-expr`. It repeats this process until `-r` is false in the resulting state after evaluating `test-expr`. It returns the resulting state after evaluating the last test expression (since it will be the last expression to be evaluated).

Examples:

```
> (while:  '(> a 0) '((:= a (- a 1))) (hash 'a 5))
'#hash((-r .  #f) (a .  0))


> (while: '(< a 10)
    '((printf "a=~a\n" a) (:= a (+ a 1)))
```

```
    (hash 'a 0))
a=0
a=1
a=2
a=3
a=4
a=5
a=6
a=7
a=8
a=9
'#hash((-r . #f) (a . 10))
```

2.7 (func params body-exprs state) (15 points)

This function takes a list of parameters `params`, a list of body expressions `body-exprs`, and a state `state`. It creates **a function** that takes a list of arguments `args` and a state `state` and evaluates `body-exprs` with `args` and `state`, and **returns the value of the `-r` variable** in the resulting state after evaluating body expressions (does not return a state unlike others). The resulting function is bound to the `-r` variable in the resulting state.

Disclaimer: The state should not be modified by the function. However, the function should be able to access to variables in the state. The only modification to the state should be the binding of the function to the `-r` variable.

Examples:

```
> (func '(x y) '((:= x (* x x)) (:= y (- y x))) empty-state)
'#hash((-r .  #<procedure:...>) (now, we can get the function from the result)
> ((get (func '(x y) '((:= x (* x x)) (:= y (- y x))) empty-state) '-r) 5 10)
-15 (see that the function only returns a value, not a state)
> ((get (func '(x) '((:= x (* x x)) (:= y (- y x))) (hash 'y 100)) '-r) 5)
75 (see that the function can access the variables in the state)
```

(map-eval lst state) (suggested for eval-expr, not mandatory)

Given a list of expressions `lst` and a state `state`, this function evaluates each expression in `lst` with `eval-expr` function and returns a list of the results together with the resulting state after evaluating the last expression in `lst`. This function is not mandatory, but it is highly suggested to use this function in `eval-expr` function.

Examples:

```
> (map-eval '((+ 1 2) (* 3 4)) empty-state)
('3 12 #hash((-r .  12)))
> (map-eval '(a b c d) (hash 'a 1 'b 2 'c 3 'd 4))
('1 2 3 4 #hash((-r .  4) (a .  1) (b .  2) (c .  3) (d .  4)))
```

2.8 (eval-expr expr state) (20 points)

This function takes an expression `expr` and a state `state`, and evaluates the expression with the given state. It puts the resulting value of the expression to the `-r` variable in the resulting state, and returns the new state. The expression can be a literal, a variable, a function call, a conditional expression, a while loop, or a function definition. You can assume that the expression is syntactically correct, and you do not need to check for errors. You can use `map-eval` function in this function.

Hint: Given the expression,

- If it is a list, you should check its first element to determine the operation. If the operation is one of `:=`, `if:`, `while:`, or `func`, you can basically append the state into the list and call `eval` function on the list.
- If the operation is `lambda`, just `eval` the list and put the resulting value into the `-r` variable in the resulting state.
- If the operation is a symbol (e.g., `'+`, `'*`, or some other symbol that), `map-eval` all of the elements in the list and apply the operation to the resulting values.
- Else, if the operation is not a list, then it is either a literal or a variable. If it is a literal, just put the literal into the `-r` variable in the resulting state. If it is a variable, get the value of the variable from the state and put it into the `-r` variable in the resulting state.

Examples:

```
> (eval-expr '5 empty-state)
'#hash((-r .  5))
> (eval-expr 'a (hash 'a 5))
'#hash((-r .  5) (a .  5))
> (eval-expr '(+ 1 2) empty-state)
'#hash((-r .  3))
> (eval-expr '(+ a b) (hash 'a 1 'b 2))
'#hash((-r .  3) (a .  1) (b .  2))
> (eval-expr '(:= a 5) empty-state)
'#hash((-r .  5) (a .  5))
> (eval-expr '(:= a (+ 1 2)) empty-state)
'#hash((-r .  3) (a .  3))
> (eval-expr '(if:  (< 1 2) (5) (10)) empty-state)
'#hash((-r .  5))
> (eval-expr '(if:  (< 2 1) (5) (10)) empty-state)
'#hash((-r .  10))
> (eval-expr '(while:  (< a 10) ((:= a (+ a 1)))) (hash 'a 0))
'#hash((-r .  #f) (a .  10))
> (eval-expr '(func (x y) (+ x y)) empty-state)
'#hash((-r .  #<procedure:...>))
```

2.9 (eval-exprs exprs state) (5 points)

Given a list of expressions `exprs`, this function evaluates each expression in `exprs` with `eval-expr` function and returns the resulting state after evaluating the last expression in `exprs`. You need to call each expression with the resulting state of the previous expression.

Hint: `foldl`.

Examples:

```
> (eval-exprs '((:= a 5) (+ a 1)) empty-state)
'#hash((-r .  6) (a .  5))
> (eval-exprs '((:= a 5) (+ a 1) (:= b 10) (+ a b)) empty-state)
'#hash((-r .  15) (a .  5) (b .  10))
> (eval-exprs (parse "example1.pi") empty-state)
```

```
a = 5
b = (2 3)
c = 5
d = 12
e = 24
f = 1/8
pi-num = 3.14
circle area = 78.5
triangle area = 10.825317547305483
square area = 25
'#hash((-r . #<void>)
       (a . 5)
       (area . #<procedure:...>)
       (b . (2 3))
       (c . 5)
       (d . 12)
       (e . 24)
       (f . 1/8)
       (pi-num . 3.14))
```

3 Submission

You will submit two files: `main.rkt`, `feedback.txt`. Your code should be in one file named `main.rkt`. First five lines of your `main.rkt` file must have exactly the lines below since it will be used for compiling and testing your code automatically:

```
; name surname
; student id
; compiling: yes
; complete: yes
#lang racket
```

The third line denotes whether your code compiles correctly, and the fourth line denotes whether you completed all of the project, which must be `no` if you are doing a partial submission. This whole part must be lower case and include only the English alphabet. Example:

```
; alper ahmetoglu
; 2012400147
; compiling: yes
; complete: yes
#lang racket
```

We are interested in your feedback about the project. In the `feedback` file, please write your feedback. This is optional, you may leave it empty and omit its submission.

4 Prohibited Constructs

The following language constructs are ***explicitly prohibited***. You *will not get any points* if you use them:

- Any function or language element that ends with an `!`.
- Any of these constructs: `begin`, `begin0`, `when`, `unless`, `for`, `for*`, `do`, `set!-values`.
- Any language construct that starts with `for/` or `for*/`.

5 Tips and Tricks

- You can use higher-order functions `apply`, `map`, `foldl`, `foldr`. You are also encouraged to use anonymous functions with the help of `lambda`.
- You can use Racket reference, either from DrRacket's menu: Help, Racket Documentation, or from the following link `https://docs.racket-lang.org/reference/index.html`.
- A useful link for the difference between `let`, `let*`, `letrec`, and `define`: `https://stackoverflow.com/questions/53637079/when-to-use-define-and-when-to-use-let-in-racket`.
- There will be many test cases from basic ones to more complicated ones. Do not get discouraged if you cannot pass the more complicated ones. You can still get a good grade by passing the basic ones.