Comp301 Project 2

Ahmet Hakan Hafif (64092)
Kerem Aksoy (64243)

Problem A)

- Part 1: 5 components of a language:
  - Syntax and Datatypes
  - Values
  - Environment
  - Behavior Specification
  - Behavior Implementation
    - Scanning
    - Parsing
    - Evaluation
- Part 2:
  - Syntax and Datatypes are defined in **lang.rkt** and **data-structures.rkt** respectively.
  - Values → For the values, we need an interface to make use of the definitions. (num-val,bool-val,expval-->num,expval-->bool) **data-structures.rkt**
  - Environment: environment is defined and handled in **environment.rkt to provide an initial to value-of to work on.**
  - Behavior Specification is defined in interp.rkt
  - Behavior Implementation
    - Lang.rkt scans and parses given procedures via its grammar.
    - Evaluation happens with the help of interp.rkt
    - Interp.rkt basically have the behavior implementation

Problem B)

- Part 1:

```
(define init-env
  (lambda ()
    (extend-env
     'z (num-val 6)
     (extend-env
      'y (num-val 3)
      (extend-env
       'x (num-val 4)
       (empty-env))))))
```

- Step by Step:
  - Init-env = (empty-env)
  - Init-env = ([x = 4] , ‘empty-env)

- Init-env = ([y = 3] , [x = 4] , ‘empty-env)
- Init-env = ([z = 6] , [y = 3] , [x = 4] , ‘empty-env)

Problem C)

- Expressed Values = Possible values of expressions
- Denoted Values = Possible values of variables
- Both of them are declared as **Int + Bool + String** in our language.
- Therefore, the expressed and denoted values for our language are both integers, booleans or strings

Problem D)

PART 5

We implemented the grammar of my cond in lang.rkt, however we could not handle its implementation inside of interp.rkt. So, we extracted the test cases which included my-cond. Also, We had to commented out the grammar of it since interp.rkt gives and error as missing case if there was an grammar inside lang.rkt

Grammar for my-cond

```
;; grammar for my-cond-exp
(expression
 ("my-cond" expression "then" expression "," (arbno expression "then" expression ",") "else" expression)
 my-cond-exp)
```

PART 6 : Custom Expression

- **6.1 The Mod Algorithm**
  - As a group, we decided to add a mod procedure with the help of usage of helper functions that we learned. This procedure:
    - Takes 2 expressions as an input namely divider (n) and number (a)
    - Checks if given number is negative
      - If negative, recursively sums divider and number until number becomes greater than 0.
    - If number is greater than divider, recursively calls mod-exp with divider and number - divider.
    - After recursive calls, checks if divider and number is equal
      - If equal, returns 0
    - If the number is less than the divider, it returns the number itself.

6.2 The Syntax of the expression for Mod

We implemented this mod operation as an expression as requested. In the syntax, it takes two expressions which are basically two numbers at the core. First one is the number whose mod would be taken in the second number. Since we define the syntax in a way that it takes two expressions, we can give any expressions as inputs whose output that returned from value-of are expressed numbers.

```
;; Grammar for custom data type mod-exp
(expression
 ("mod" "(" expression "," expression ")")
 mod-exp)
```

Example syntaxes : “mod(8,3)”, “mod(op(6,'add',2),op(3,'mult',1))”

6.3 The mod-helper Function

```
(define (mod-helper x n)
  (cond ((= x n) 0)
        ((< x 0) (mod-helper (+ x n) n))
        ((< x n) x)
        ((> x n) (mod-helper (- x n) n))))
```

This helper function is a scheme procedure which takes two scheme numbers(not expressed numbers) and applies the algorithm steps which were explained in the algorithm part (6.1).

6.4 Implementing mod-exp case of value-of

```
;; Implementing mod-exp
(mod-exp (exp1 exp2)
         (let ((a (expval->num (value-of exp1 env)))
               (n (expval->num (value-of exp2 env))))
           (num-val (mod-helper a n)))))
```

In this part, we first need to take the values of two input expressions. To do that, we simply call the value-of function on both of the expressions in the initial environment called env. Since the contract is that the returned values of these expressions need to be expressed numbers, we need to call the expval->num function to convert the expressed numbers into scheme numbers to give them as inputs to the mod-helper function which expects two scheme numbers. After calling mod-helper and taking the returned value as a scheme number, we need to call the num-val procedure to convert it back to an expressed number since value-of returns expressed values.

Part 7 Test cases of Custom Expression : mod-exp

```
;; Test cases for custom expression mod-exp
(mod-simple "mod(8,3)" 2) ;; Simple one
(mod-complex1 "mod(op(6,'add',7),op(2,'mult',2))" 1) ;Checking when x is positive
(mod-complex2 "mod(op(0,'sub',5),op(2,'mult',2))" 3) ;;Checking when x is negative
(mod-complex3 "mod(op(25,'div',5),op(15,'sub',4))" 5) ;;Checking when x is smaller than n
```

Tests were added into the tests.rkt file.

Workload
While doing this project, we simply met face to face and did this project as a whole together.