COMP 301 Project 2

Baran Berkay Hökelek, 0060673
Berkay Barlas 0054512
Utku Noyan 0060717

November 15, 2020

Workload Breakdown

In this project first we met and discuss each section about how we can implement code and answer questions in Part A, B, and C. Then we divided parts according to their grades. You can see workload breakdown:

- Utku Noyan : Part A, B, and C
- Baran Berkay Hökelek: D1, D2, D3
- Berkay Barlas: D1, D4 + Bonus

After finishing the all part, we all checked and discussed these parts.

A

This part will prepare you for the following parts of the project. (15 pts)

(1) Write the 5 components of the language :

- Syntax and datatypes
  - a-program → Program ::= Expression
  - Const-exp → Expression ::= Number
  - diff-exp → Expression ::= -(Expression, Expression)
  - zero?-exp → Expression ::= zero? (Expression)
  - if-exp → Expression ::= if Expression then Expression else Expression
  - var-exp → Expression ::= Identifier
  - let-exp → Expression ::= let Identifier = Expression in Expression

- Values
  - Expressed values: Possible values of expressions
  - Denoted values: Possible values of variables
- Environment
  - ρranges over environments
  - [] denotes empt environment
  - $[var\ =\ val]_\rho$denotes (extend-env var val ρ)
  - $[var1\ =\ val1,\ var2\ =\ val2]_\rho$abbreviates $[var1\ =\ val1]([\ var2\ =\ val2]_\rho)$, *etc*
  - $[var1\ =\ val1,\ var2\ =\ val2,\ ...]_\rho$ denotes the environment where the value of var1 is val1, etc.
- Behavior specification
  - We specify the behavior of programs, expressions, and Observer(such as value-of)
- Behavior implementation
  - Scanning
  - Parsing
  - Evaluation

(2) For each component, specify where or which racket file (if it applies) we define and handle them.

- Syntax and datatypes
  - a-program → In **interp.rkt** file defined, and handled.
  - Const-exp → In **interp.rkt** file defined, and handled.
  - zero?-exp → In **interp.rkt** file defined, and handled.
  - if-exp → In **interp.rkt** file defined, and handled.
  - var-exp → In **interp.rkt** file defined, and handled.
  - let-exp → In **interp.rkt** file defined, and handled.
- Values
  - Expressed values: In **datastructures.rkt** file we define expressed values and it is either a number, a boolean, a string.
  - Denoted values: Not applicable
- Environment

- In **environments.rkt** file first initial environment defined, then environment constructors and observerst implemented.

- Behavior specification
  - Grammatical specification is defined and handled in the **lang.rkt** file above the sllgen boilerplate comment.

- Behavior implementation
  - In the **lang.rkt** file , Scan&Parse implemented below the sllgen boilerplate comment.

  - Evaluation is implemented in **interp.rkt** file.

B

In this part, you will create an initial environment for programs to run. (10 pts)

(1) Create an initial environment that contains 3 different variables (x, y, and z).

It is defined in "environments.rkt" and x,y,z initialized as 2,3,7. (init-env) = [x=2, y=3, z=7]

(2) Using the environment abbreviation shown in the lectures, write how the environment changes at each variable addition.

- $[]_{\rho 0}$: after empty-env
- $[z = 7]_{\rho 1}$: after (extend-env 'z (num-val 7)(empty-env))
- $[z = 7,\ y = 3]_{\rho 2}$: after (extend-env 'y (num-val 3)(extend-env 'z (num-val 7)(empty-env)))
- $[z = 7,\ y = 3,\ x = 2]_{\rho 3}$: after (extend-env 'x (num-val 2)(extend-env 'y (num-val 3)(extend-env 'z (num-val 7)(empty-env))))
- $[z = 7,\ y = 3,\ x = 2]_{\rho 3}$: final environment

C

Specify expressed and denoted values for MYLET language. (5 pts)

The used expressed values in MYLET language are either a number, a boolean, a string.

The used denoted values in MYLET language are either a number, a boolean, a string.

D

D1

In the-lexical-spec, a string is defined as:

```
(string (#\' letter (arbno (not #\') letter) #\') string)
```

And in the-grammar, a string expression is defined as:

```
(expression (string) str-exp)
```

In the value-of function, a string expression is evaluated just like a number:

```
(str-exp (str) (str-val str))
```

which requires a definition of the datatype str-val:

```
(define-datatype expval expval?
  (str-val
   (string string?)))
```

which uses the EOPL language's built-in string? function.

Adding a string? check to the function sloppy->expval? was also required.

```
((string? sloppy-val) (str-val sloppy-val))
```

D2

In the-grammar, op-exp is defined as:

```
(expression
    ("op(" expression "," expression "," number ")")
    op-exp)
```

And the value of op-exp is evaluated with this function:

```
(op-exp (exp1 exp2 num)
              (let ((val1 (value-of exp1 env)) (val2 (value-of exp2 env)))
                (let ((num1 (expval->num val1)) (num2 (expval->num val2)))
                  (cond
                    ((= num 1) (num-val (+ num1 num2)))
                    ((= num 2) (num-val (* num1 num2)))
                    ((= num 3) (num-val (/ num1 num2)))
                    (else (num-val (- num1 num2)))
                    ))))
```

D3

In the-grammar, if-exp is defined as:

```
(expression
     ("if" expression "then" expression
     (arbno "elif" expression "then" expression) "else" expression)
     if-exp)
```

This expression is interpreted as:

```
      (if-exp (exp1 exp2 conds exps exp3)
                     (let ((val1 (value-of exp1 env)))
                       (cond
                         ((expval->bool val1) (value-of exp2 env))
                         ((null? conds) (value-of exp3 env))
                         ((null? (cdr conds)) (value-of exp3 env))
                         (else (value-of
                         (if-exp
                         (cadr conds)
                         (cadr exps)
                         (cddr conds)
                         (cddr exps)
                         exp3)
                         env))
                         )))
```

D4

We defined string expression as our custom expression choice. It can be used for string operations such as string concatenation. It takes to expression that should have value of string and a number for operation type. We defined default operation type as string concatenation which takes two strings and returns concatenated string.

In the-grammar, str-exp is defined as:

```
;;;; str-op ;;;;
(expression
 ("str-op(" expression "," expression "," number ")")
 str-op)
```

This expression is interpreted as:

```
 ;;;; str-op ;;;;
 (str-op (exp1 exp2 type) ;(str-val str)
         (let ((val1 (value-of exp1 env)) (val2 (value-of exp2 env)))
           (let ((str1 (expval->string val1)) (str2 (expval->string val2)))
             (cond
               ; concat operation
               (else (str-val
                      (string-append
                       (substring str1 0 (- (string-length str1) 1))
                       (substring str2 1))))
             )
           )
         )
```

E

We added following test cases for our custom expression

```
(str-op-concat-test "str-op('this','this',0) " "'thisthis'")
(str-op-concat-test2 "str-op('abc','def',0) " "'abcdef'")
```