GENERIC OPERATORS AND RACKET1.RKT 6

COMPUTER SCIENCE 61AS

Racket-1 Practice

1. Exercise 1 (Long): Show how racket1 evaluates the following expression. Show all of the calls to eval-1, apply-1 and substitute. Dont show recursive calls to substitute though. (In essence, if we traced all 3 of these procedures, but ignored recursive calls to substitute, what would be the output?) Note: If you want to understand how substitute works, you should trace the recursive calls to substitute as well.

```
(((lambda (x)
     ((lambda (y)
        (lambda (x) (+ x y x)))
     x)) 5) 10
```

Solution: Eval-1 is called with

```
(((lambda(x) ((lambda(y) (lambda(x) (+ x y x))) x)) 5) 10);
```

it sees that it is a procedure call, naturally a list of two elements. Whats the procedure? Well, to find out, call eval-1 on it!

- eval-1 is called with

  ```
  (((lambda(x) ((lambda(y) (lambda(x) (+ x y x))) x)) 5) 10)
  ```

  . This is just a lambda expression, it evaluates to a procedure represented by the same lambda expression, so we will just return itself.

  – eval-1 returns

    ```
    (lambda(x) ((lambda(y) (lambda(x) (+ x y x))) x))
    ```

    . Next, we need to map eval-1 over the arguments, which is just the list of a single argument - (5). Mapping eval-1 over that will return (5) since 5 is self-evaluating.

Now its time to apply:

– apply-1 called with proc

  `(lambda(x) ((lambda(y) (lambda(x) (+ x y x))) x))`

  and argument list (5). apply-1 sees that this is a compound procedure, so it tries to substitute 5 for x in the body of the expression:

  * substitute called exp as

    `(lambda(x) ((lambda(y) (lambda(x) (+ x y x))) x))`

    , params as (x), args as (5), and bound as (). We'll try tracing through substitute later, but we already know it is supposed to return the following:

  * Substitute returns

    `((lambda(y) (lambda(x) (+ x y x))) 5)`

    . Evaluate the body —¿

  * eval-1 is called with

    `((lambda(y) (lambda(x) (+ x y x))) 5)`

    . Looks like a procedure call! The procedure will evaluate to

    `(lambda(y) (lambda(x) (+ x y x))`

    with argument list (5). Lets apply!

    · apply-1 called with proc

      `(lambda(y) (lambda(x) (+ x y x))`

      and argument list (5). The procedure is a compound, so we call substitute on its body. Substitute is called with

      `(lambda(y) (lambda(x) (+ x y x))`

      with params (y), args (5), and bound (). It does the same as above:

    · substitute returns

      `(lambda(x) (+ x 5 x))`

      ; Now, eval the body:

    · eval-1 is called with

      `(lambda(x) (+ x 5 x))`

      . Thats another lambda expression! we're just going to return the procedure, represented as itself:

      · eval-1 returns

        `(lambda(x) (+ x 5 x))`

    * apply-1 returns

      `(lambda(x) (+ x 5 x))`

  - eval-1 returns

    `(lambda(x)(+ x 5 x))`

  - apply-1 returns

    `(lambda(x)(+ x 5 x))`

- eval-1 returns

  `(lambda(x) (+ x 5 x))`

  , and weve finally found out what the procedure of our original procedure call is! What are the arguments? Well, its just (10). Time to apply!

- apply-1 called with

  `(lambda(x) (+ x 5 x))`

  and argument list (10). Of course, this is a compound procedure, so we, as usual, call substitute on the body:

  - substitute called with exp as

    `(+ x 5 x)`

    , params (x), args (10), and bound ().

  - substitute returns

    `(+ 10 5 10)`

    . Now we can evaluate the body.

  - eval-1 called with

    `(+ 10 5 10)`

    . A procedure call! The procedure is evaluated as the Racket + procedure, and the argument list evaluated as (10 5 10). We call apply-1:

    * apply-1 called with Rackets + procedure and args (10 5 10). We will use Rackets apply to apply the primitive procedure, getting 25.

    * apply-1 returns 25

  - eval-1 returns 25

- apply-1 returns 25

eval-1 returns 25

For the first call to substitute: Substitute called with exp as

```
((lambda(y) (lambda(x) (+ x y x))) x)
```

, params as (x), args as (5), and bound as (). This is a procedure call, so were going to map substitute over every element of this list:

- substitute called with exp as

  ```
  (lambda(y) (lambda(x) (+ x y x)))
  ```

  . This is a lambda expression, so were going to create a new lambda expression with the same formal parameters (y), and well call substitute on the body, adding the parameters to the bound list.

  – substitute called with exp as

    ```
    (lambda(x) (+ x y x))
    ```

    , params as (x), args as (5), and bound as (y). This is a lambda expression, so were going to create a new lambda expression with the same formal parameters (x), and well call substitute on the body, adding the parameters to the bound list.

    ∗ substitute called with exp as (+ x y x), params as (x), args as (5), and bound as (y x). This is a procedure call, so map substitute over everything.

      · substitute called with exp as +, params as (x), args as (5), bound as (y x). + is a symbol, so we call lookup. lookup wont find + in the params list, so itll leave it alone.

      · substitute returns +

      · substitute called with exp as x, params as (x), args as (5), bound as (y x). x is a symbol, but its also in the bound list, so well ignore it.

      · substitute returns x

      · substitute called with exp as y, params as (x) , args as (5), bound as (y x). y is in the bound list, so well ignore it.

      · substitute returns y

      · substitute called with exp as x, params as (x), args as (5), bound as (y x). x is a symbol, but its also in the bound list, so well ignore it.

      · substitute returns x

        * substitute returns

          ```
          (+ x y x)
          ```

      - substitute returns

        ```
        (lambda(x)(+ x y x))
        ```

- substitute returns

  ```
  (lambda(y) (lambda(x) (+ x y x)))
  ```

  ; now we keep mapping substitute to the next element:

- substitute called with exp as x, params as (x), args as (5), and bound as (). x is a symbol, and its not in the bound list, so we use lookup and replace x with 5.
- substitute returns 5

substitute returns

```
((lambda(y) (lambda(x) (+ x y x))) 5)
```

2. If I type this into Racket, I get an unbound variable error: (eval-1 ’x) Why didnt this just return x, unquoted? What should I have typed in instead? (Assume that I want to use eval-1 from Racket in order to get the symbol x, unquoted.)

Solution: The quote in front of x protects x from Racket, but not from Racket-1. Recall that eval-1 is just a procedure, and for Racket to make that procedure call, it first evaluates all its arguments including (quote x) before passing the argument to eval-1. Then, when eval-1 sees the symbol x, it tries to call eval on it, throwing an unbound variable error. The problem, then, is that the expression ’x is evaluated TWICE once by the Racket evaluator, and once by eval-1. Thus, to protect it twice, you need two quotes: (eval-1 ”x) Note that if you type just ’x into Racket-1, it works: Racket-1: ’x x Make sure you understand the difference, and why you dont need two quotes there (because Racket never evaluates ’x, unlike the previous case).

3. Hacking Racket-1: For some reason, this expression works:

   ```
   ('(lambda (x) (* x x)) 3)
   ```

   In Racket, this would cause an error, because of the quote in front of the lambda expression. Why does it work in Racket-1? What fact about Racket-1 does this exploit?

Solution: When eval-1 sees the procedure call and tries to evaluate the procedure, it sees that it is a quoted expression, and unquotes it. Then, the procedure is passed to apply-1, which sees that it is a lambda expression, and uses it as such. This exploits the fact that in Racket-1, a compound procedure is represented as a list that looks exactly like the lambda expression that created it. Thus, even though we never evaluated the lambda expression into a procedure value, Racket-1 is still fooled into thinking its a valid procedure.

```
(define (eval-1 exp)
    (cond ((constant? exp) exp)
          ((symbol? exp) (eval exp)) ; use underlying Racket's EVAL
          ((quote-exp? exp) (cadr exp))
          ((if-exp? exp)
              (if (eval-1 (cadr exp))
                  (eval-1 (caddr exp))
                  (eval-1 (cadddr exp))))
          ((lambda-exp? exp) exp)
          ((pair? exp) (apply-1 (eval-1 (car exp)) ; eval the operator
                                (map eval-1 (cdr exp))))
          (else (error "bad expr: " exp))))
(define (apply-1 proc args)
    (cond ((procedure? proc) ; use underlying Racket's APPLY
              (apply proc args))
          ((lambda-exp? proc)
              (eval-1 (substitute (caddr proc) ; the body
                                  (cadr proc) ; the formal parameters
                                  args ; the actual arguments
                                  '()))) ; bound-vars, see below
          (else (error "bad proc: " proc))))
(define (substitute exp params args bound)
    (cond ((constant? exp) exp)
          ((symbol? exp)
              (if (memq exp bound)
                  exp
                  (lookup exp params args)))
          ((quote-exp? exp) exp)
          ((lambda-exp? exp)
              (list 'lambda
                    (cadr exp)
                    (substitute (caddr exp)
                                params
                                args
                                (append bound (cadr exp)))))
               (else (map (lambda (subexp)
                                  (substitute subexp
                                              params
                                              args
                                              bound))
                          exp))))
```

Data Directed Programing

Exercise 4: The TAs have broken out in a cold war; apparently, at the last midterm-grading session, someone ate the last potsticker and refused to admit it. It is near the end of the semester, and Professor Hilfinger really needs to enter the grades. Unfortunately, the TAs represent the grades of their students differently, and refuse to change their representation to someone elses. Professor Hilfinger is far too busy to work with five different sets of procedures and five sets of student data, so for educational purposes, you have been tasked to solve this problem for him. The TAs have agreed to type-tag each student record with their (the TAs) first name, conforming to the following standard:

```
(define type-tag car)
(define content cdr)
```

Its up to you to combine their representation into a single interface for Professor Hilfinger to use.

a. Write a procedure (make-tagged-record ta-name record) that takes in a TAs student record, and type-tags it so its consistent with the type-tag and content selector procedures defined above.

Solution:

```
(define make-tagged-record cons)
```

b. A student record consists of two things: a name item and a grade item. Each TA represents a student record differently. Marion uses a list, whose first element is a name item, and the second element the grade item. Jisoo uses a cons pair, whose car is the name item, and the cdr the grade item. Make calls to put and get, and write generic getname and get-grade procedures that take in a tagged student record and return the name or grade items, respectively.

Solution:

```
(put 'Marion 'get-name car)
(put 'Marion 'get-grade cadr)
(put 'Jisoo 'get-name car)
(put 'Jisoo 'get-grade cdr)

(define (get-name tagged-record)
    ((get (type-tag tagged-record) 'get-name)
    (content tagged-record)))
(define (get-grade tagged-record)
    ((get (type-tag tagged-record) 'get-grade)
    (content tagged-record)))
As you can see, the above two look very similar,
so we can define a generic procedure:
(define (operate op tagged-record)
    ((get (type-tag tagged-record) op)
    (content tagged-record)))
Then, we can just do:
(define (get-name tagged-record)
(operate 'get-name tagged-record))
(define (get-grade tagged-record)
(operate 'get-grade tagged-record))
```

c. Each TA represents names differently. Jisoo uses a cons pair, whose car is the last name and whose cdr is the first. Sam is so cool that a name is just a word of two letters, representing the initials of the student (so George Bush would be gb). Make calls to put and get to prepare the table, then write generic get-first-name and get-lastname procedures that take in a tagged student record and return the first or last name, respectively.

Solution:

```
There are a few ways to organize this. The obvious way is
to put into the table procedures that take in a full student
record and returns the first name:
(put 'Jisoo 'get-first-name
            (lambda(r) (cdr (get-name r))))
(put 'Jisoo 'get-last-name
            (lambda(r) (car (get-name r))))
(put 'Sam 'get-first-name
            (lambda(r) (first (get-name r))))
(put 'Sam 'get-last-name
            (lambda(r) (last (get-name r))))

(define (get-first-name tagged-record)
(operate 'get-first-name tagged-record))
(define (get-last-name tagged-record)
(operate 'get-last-name tagged-record))

Or, we can instead, put into the table procedures that take
in a name item rather than the whole record:
(put 'Jisoo 'get-first-name cdr)
(put 'Jisoo 'get-last-name car)
(put 'Sam 'get-first-name first)
(put 'Sam 'get-last-name last)
(define (get-first-name tagged-record)
    ((get (type-tag tagged-record) 'get-first-name)
        (get-name tagged-record)))
Unfortunately, this would mean we can no longer use operate.
So the first way of doing this is neater.
```

d. Each TA represents grades differently. Marion is lazy, so his grade item is just the total number of points for the student. Sam is more careful, so his grade item is an association list of pairs; each pair represents a grade entry for an assignment, so the car is the name of the assignment, and the cdr the number of points the student got. Make calls to put and get to prepare the table, and write a generic get-total-points procedure that take in a tagged student record and return the total number of points the student has.

Solution:

```
(put 'Marion 'get-total-points get-grade)
(put 'Sam 'get-total-points
(lambda(r) (apply + (map cdr (get-grade r)))))
We use map to get a list of points, and apply to add them up.
(define (get-total-points tagged-record)
(operate 'get-total-points tagged-record))
```

e. Now Professor Hilfinger wants you to convert all student records to the format he wants. He has supplied you with his record-constructor, (make-student-record name grade), which takes in a name item and a grade item, and returns a student record in the format Professor Hilfinger likes. He also gave you (make-name first last), which creates a name item, and (make-grade total-points), which takes in the total number of points the student has and creates a grade item. Write a procedure, (convert-to-hilfinger-format records), which takes in a list of student records, and returns a list of student records in Professor Hilfingers format, each record tagged with 'Hilfinger.

Solution:

```
(define (convert-to-harvey-format records)
    (map (lambda(r)
        (make-tagged-record 'hilfinger
            (make-student-record
                (make-name (get-first-name r)
                    (get-last-name r))
        (make-grade (get-total-points r)))))
    records))
```