CIS 425, Fall 2019, HW 3– Racket and SML

Turn in: Sunday, October 20

October 15, 2019

The submission of assignments is optional. Submitted assignments will not be graded. If your solution differs from the posted solution, then you can submit it and indicate clearly why you think your solution is correct.

Interpreter in Racket

Consider the following simple language E :

```
E ::= num | E + E | E * E
```

where num is any integer. How do we represent expressions defined by this grammar in Racket? We use an integer to represent itself and a list to represent a compound expression. The first element in the list will be a symbol, either + or *, and the other two will be the operands. In other words, we can think of the grammar in terms of Racket expressions:

```
E ::= num | (+ E E) | (* E E)
```

In order to form a list in Racket without treating it as a function call, you prefix it with "*quote*" [1]. So the expression $3 + (4 * 5) + 6$ might be written as

```
(quote (+ 3 (+ (* 4 5) 6)))
```

[1] A more concise idiom used for "quote" is to use the quotation mark itself. For example the expression '(+ 3 2) corresponds to (quote (+ 2 3)). For the rest of the assignment, these will be used interchangeably

An example of a function on E is as follows:

```
(define (desc e)
  (cond
    ((number? e) "Number")
    ((eq? (car e) '+) "Plus")
    ((eq? (car e) '*) "Times")
    (else (error "Error: Not a valid E term"))))
```

What is the result of the following function calls:

- `(desc 1)`
- `(desc '(+ 1 2))`

- `(desc '(* 1 2))`

- `(desc '(/ 1 2))`

Write a function *interp* that accepts as input an expression E, interprets it, and returns the integer result. On ill-formed input (that is, input that does not conform to the grammar for E), *interp* must raise an error. For example:

```
> (interp 1)
1
> (interp '(+ 1 2))
3
> (interp '(+ (+ 1 2) 3))
6
> (interp '(+ (+ 1 2) (* 3 4)))
15
> (interp '(cons 1 2))
Error: Not a valid E term
```

(Important: You may not use eval to write *interp*.)

You may use this skeleton code to get started [2]:

[2] Notice that one branch of the recursion is done for you.

```
(define (interp e)
  (cond
    ((number? e) ________)
    ((eq? (car e) '+) ________)
    ((eq? (car e) '*) (* (interp (car (cdr e))) _______))
    (else (error "Error: Not a valid E term"))))
```

Syntax Manipulation

We have now seen Racket's "quotation" mechanism. Consider the following example:

```
(quote (+ (+ 2 3) 1))
```

From what we have already seen, we expect the *cadr* of this expression to be $(+\ 2\ 3)$. Now suppose we would like to return the list $(+\ 5\ 1)$. That is, we want to evaluate only the *cadr* of our expression. Racket provides us a mechanism, "quasiquote" [3] allows us to do just that:

[3] Again, Racket provides concise notation for quasiquotes: '(+ ,(+ 2 3) 1), corresponds to (quasiquote (+ (unquote (+ 2 3)) 1)). Notice that this is not the 'normal' quotation mark but a backtick (usually located above the tab key on your keyboard.) instead. Again, for the rest of the assignment, the two notations will be used interchangeably

```
(quasiquote (+ (unquote (+ 2 3)) 1))
```

That is, as intended, this will return the expression

```
'(+ 5 1).
```

Write a translator function called TR, with one argument E, such that

```
(TR E)
```

produces a new expression E' such that every occurrence of the symbol + in E is replaced with the symbol *, and every occurrence of the symbol * in E is replaced with the symbol +.

For example,

```
(TR '(+ 1 2))
```

returns the expression:

```
'(* 1 2)
```

and

```
(TR '(+ (* 2 3) (* (+ 1 2) (* 8 8))))
```

produces

```
'(* (+ 2 3) (+ (* 1 2) (+ 8 8)))
```

Use Racket's quasiquote mechanism in returning the result.

You can run the result of your translator using the eval function:

```
> (eval (TR '(+ (* 2 3) (* (+ 1 2) (* 8 8)))))
90
```

Again, you may use the following skeleton [4]:

[4] again, notice that one branch of recursion has been solved for you.

```
(define (interp e)
  (cond
    ((number? e) ________)
    ((eq? (car e) '+) ________)
    ((eq? (car e) '*) (quasiquote (+  (______) (unquote (TR (car (cdr (cdr e))))))))
    (else (error "Error: Not a valid E term"))))
```

Map in Racket

As in most functional languages (including SML and Haskell), map is a built-in higher-order function which takes two arguments, a function F and a list L, and returns a similar list with F applied to each element in L. For example:

```
> (define (square x) (* x x))
> (define L '(1 2 3 4 5))
> (map square L)
'(1 4 9 16 25)
> (map square empty)
```

Write a *map* function that takes a function and a list and maps the function onto each of the terminal elements of that list. You may use provided skeleton [5].

[5] In Racket, the function *cons* can be used to build a new list from its head (*car*) and its tail (*cdr*). Moreover, the empty list can be represented by *empty* or the expression '()

```
> (define (square x) (* x x))
> (define L '(1 2 3 4 5))
> (map square L)
'(1 4 9 16 25)

(define (map f x)
  (cond
    ((null? x) ________)
    ((pair? x) (cons ____________))))
```

Interpreter in SML

Consider the following simple language E:

```
E ::= num| E + E | E * E
```

where num is any integer. We represent terms of E by using a corresponding datatype:

```
datatype E = NUM of int | PLUS of E * E | TIMES of E * E
```

Specifically, we use the NUM constructor to represent an integer and PLUS or TIMES to represent a compound expression. In other words, we can think of the grammar in terms of SML expressions [6]:

[6] Notice how SML notation differs for product **types** as opposed to product **values**. As you can check at the SML prompt, value pairs are separated by a comma. For example: $(1, 2)$. The types of such values on the other hand are separated by *. That is the value $(1, 2)$ has the type int * int, not (int, int).

```
 E ::= NUM num | PLUS (E, E) | TIMES (E, E)
```

So the expression $3 + (4 * 5) + 6$ might be written as

```
PLUS (NUM 3, PLUS (TIMES (NUM 4, NUM 5), NUM 6)
```

Write a function interp that accepts as input a program written in E, interprets it, and returns the integer result. For example:

```
- interp (NUM 1);
val it = 1 : int
- interp (PLUS (NUM 1, NUM 2));
val it = 3 : int
- interp (PLUS (PLUS (NUM 1, NUM 2), NUM 3));
val it = 6 : int
- interp (PLUS (PLUS (NUM 1, NUM 2), (TIMES (NUM 3, NUM 4))));
val it = 15 : int
```

Map on Lists and Trees

As in most functional languages (including Javascript and Haskell), map is a built-in higher-order function which takes two arguments, a function F and a list L, and returns a similar list with F applied to each element in L. For example :

```
- fun square x = x * x;
val square = fn : int -> int
- val L = [1,2,3,4,5];
val L = [1,2,3,4,5] : int list
- map square L;
val it = [1,4,9,16,25] : int list
```

1. Define map to work as expected on lists.
2. Suppose we have this datatype for ML-style nested lists (i.e. trees) of integers:

   ```
   datatype tree = NIL | CONS of (tree * tree) | LEAF of int;
   ```

   Write a treemap function that takes a function and a tree and maps the function onto each of the terminal elements of that list.

   ```
   - fun square x = x * x;
   val square = fn : int -> int
   - Control.Print.printDepth := 100; (* do this or the next output will be garbled *)
   val it = () : unit
   - val L = CONS (CONS (LEAF 1, LEAF 2), CONS (CONS (LEAF 3, LEAF 4), LEAF 5));
   val L = CONS (CONS (LEAF 1,LEAF 2),CONS (CONS (LEAF 3,LEAF 4),LEAF 5)) : tree
   - treemap square L;
   val it = CONS (CONS (LEAF 1,LEAF 4),CONS (CONS (LEAF 9,LEAF 16),LEAF 25)) : tree
   ```

ML Reduce for Lists and Trees

- Define the reduce function in ML to work as expected on lists
- *John C. Mitchell*, problem 5.5