Homework #2: BNFs, Parsing, and Higher-Order Functions

Out: Sunday, February 11, Due: Tuesday, February 20, 2024, 23:55

Administrative

This is another introductory homework, and now it is for **couples work and submission**. In this homework you will be introduced to the course language and some of the additional class extensions.

In this homework (and in all future homeworks) you should be working in the “Module” language, and use the appropriate language using a `#lang` line. You should also click the “Show Details” button in the language selection dialog, and check the “Syntactic test suite coverage” option to see parts of your code that are not covered by tests: after you click “run”, parts of the code that were covered will be colored in green, parts that were not covered will be colored in red, and if you have complete coverage, then the colors will stay the same. Note that you can also set the default language that is inserted into new programs to `#lang pl`, to make things more convenient. There are some variants for the `pl` language for various purposes — in particular, `#lang pl untyped` will ignore all type declarations, and will essentially run your code in an untyped Racket.

The language for this homework is:

```
#lang pl 02
```

As in previous assignment, you need to use the special form for tests: `test`.

Reminders (this is more or less the same as the administrative instructions for the previous assignment):

This homework is for couples work and submission.

Integrity: Please do not cheat. You may consult your friend regarding the solution for the assignment. However, you must do the actual programming and commenting on your own!! This includes roommates, marital couples, best friends, etc… I will be very strict in any case of suspicion of plagiary. Among other thing, students may be asked to verbally present their assignment.

Tests: For each question, you should have enough test cases for complete coverage (DrRacket indicates covered expressions with colors for covered and uncovered source code, unless your code is completely covered). See below on the way to create tests. Note that your tests should **not only** cover the code, but also all end-cases and possible pitfalls.

Important: Your tests should cover your whole code; otherwise the server will heavily penalize your submission. You should not have any uncovered expressions after you hit "Run" — it should stay at the same color, indicating complete coverage. Furthermore, the server will run its own tests over your code, which means that you will not be able to submit code that does not work.

General note: Code quality will be graded. Write clean and tidy code.

The test form can be used to test that an expression is true, that an expression evaluates to some given value, or that an expressions raises an error with some expected text message. For example, the three kinds of tests are used in this example:

```
#lang pl
(: smallest : (Listof Number) -> Number)
(define (smallest l)
  (match l
    [(list)      (error 'smallest "got an empty list")]
    [(list n)    n]
    [(cons n ns) (min n (smallest ns))]))
(test (smallest '(5 7 6 4 8 9)) => 4)
(test (zero? (smallest '(0 1 2 3 4))))
(test (smallest '()) =error> "got an empty list")
```

In case of an expected error, the string specifies a pattern to match against the error message. (Most text stands for itself, "`?`" matches a single character and "`*`" matches any sequence of characters.)
Note that the `=error>` facility checks only errors that *your* code throws, not Racket errors. For example, the following test will not succeed:

```
(test (/ 4 0) =error> "division by zero")
```

The code for all the following questions should appear in a single .rkt file named < ID1>_<ID2>_2 (e.g., 333333333_44444444_2 for two students whose ID numbers are 333333333 and 444444444).

1. Higher Order Functions

As you already know, lists are a fundamental part of Racket. They are often used as a generic container for compound data of any kind. It is therefore not surprising that Racket comes with plenty of useful functions that operate on lists. One of the most useful list functions is **`foldl`**: it consumes a *combiner* function, an *initial* value, and an input list. It returns a value that is created in the following way:

- For the empty list, the initial value is returned,
- For a list with one item, it uses the combiner function with this item and the initial value,
- For two items, it uses the combiner function with the first and the result of folding the rest (a one-item list),
- etc.

In the general case, the value of **`foldl`** is:

```
(foldl f init (list x1 x2 x3 ... xn))

  = (f xn (... (f x3 (f x2 (f x1 init)))))
```

Note that **`foldl`** is a *higher-order* function, like **`map`**. Its type is:

```
(: foldl : (All (A B) (A B -> B) B (Listof A) -> B))
```

Use **`foldl`** together with (or without) **`map`** to define a `sum-of-squares` function which takes a list of numbers as input, and produces a number which is the sum of the squares of all of the numbers in the list. A correct solution should be a oneliner. Remember to write a proper description and contract line, and to provide sufficient tests (using the `test` form). You will need to do this for a definition of `square` too, which you would need to write for your implementation of `sum-ofsquares`.
A more detailed explanation on both functions can be found at the bottom of the assignment or here.

Here is an example of a test that you might want to perform:

```
(test (sum-of-squares '(1 2 3)) => 14)
```

2. PAE (and more H.O. functions)

a. In this question, you are asked to write a function `createPolynomial` that takes as arguments a list of k numbers $a_0, \dots, a_{k-1}$ and returns as output a function. The returned function takes a number x_0 and return the value of the polynomial $a_0 \cdot x^0 + \cdots + a_{k-1} \cdot x^{n-1}$ at x_0. To this end, you can use the built-in pl `expt` function taking two numbers a and b, and returning a^b.

The following should help you understand the task at hand:

```
> (createPolynomial '(1 2 4 2))
- : (Number -> Number)
#<procedure:polyX>


(define p2345 (createPolynomial '(2 3 4 5)))
(test (p2345 0) =>
   (+ (* 2 (expt 0 0)) (* 3 (expt 0 1)) (* 4 (expt 0 2)) (* 5
(expt 0 3))))
(test (p2345 4) =>
   (+ (* 2 (expt 4 0)) (* 3 (expt 4 1)) (* 4 (expt 4 2)) (* 5
(expt 4 3))))
(test (p2345 11) => (+ (* 2 (expt 11 0)) (* 3 (expt 11 1)) (* 4
(expt 11 2)) (* 5 (expt 11 3))))



(define p536 (createPolynomial '(5 3 6)))
(test (p536 11) => (+ (* 5 (expt 11 0)) (* 3 (expt 11 1)) (* 6
(expt 11 2))))
```

```
(define p_0 (createPolynomial '()))
(test (p_0 4) => 0)
```

Remark: all recursive calls should be in tail recursion.

You are given the following partial code. Use it as a basis for your full code. Don't forget to add comments and tests.

```
(: createPolynomial : (Listof Number) -> <-fill in->)
(define (createPolynomial coeffs)
  (: poly : (Listof Number) Number Integer Number ->
Number)
  (define (poly argsL x power accum)
     (if <-fill in->
        <-fill in->
        <-fill in-> )
  (: polyX : Number -> Number)
  (define (polyX x)
     ←fill in→)
←fill in→)
```

b. We now move on to define a language PLANG that supports evaluating a polynomial on a sequence of points (numbers). You should base your solution on the interpreter we have written for the AE language. Specifically, your code should keep most of the definitions therein. The changes you do need to make are described next.

i. Write the BNF for the new language to allow for expressions of the form $\{\{\,\boldsymbol{poly}\ C_1\ C_2 \ldots C_k\}\ \{P_1\ P_2 \ldots P_\ell\}\}$ where all C_i and all P_j are valid AE expressions (and both $k \geq 1$ and $\ell \geq 1$).
See examples for **valid** expressions:

```
"{{poly 1 2 3} {1 2 3}}"
"{{poly 4/5 } {1/2 2/3 3}}"
"{{poly 2 3} {4}}"
"{{poly 1 1 0} {-1 3 3}}"
"{{poly {/ 4 2}  {- 4 1}} {{- 8 4}}}"
"{{poly {+ 0 1} 1 {* 0 9}} {{- 4 5} 3 {/ 27 9}}}"
```

Also see examples for **invalid** expressions:

```
"{{poly } {1 2 3} }"
"{{poly 4/5 } {1/2 2/3 3} {poly 1 2 4} {1 2}}"
"{{poly 2 3} {}}"
"{{poly 1 1 3} }"
```

You may use the following skeleton for your BNF:

```
#|
  The grammar:
    <PLANG> ::=
    <AEs>   ::=
    <AE>    ::=
  |#
```

ii. Write the parser for the new language. Use the following partial code as well the test examples provided below.

```
(define-type PLANG
    [Poly (Listof AE) <-fill in->])

  (define-type AE
    [Num  Number]
    [Add  AE AE]
    [Sub  AE AE]
    [Mul  AE AE]
    [Div  AE AE])

  (: parse-sexpr : Sexpr -> AE)
  ;; to convert s-expressions into AEs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)    (Num n)]
      [(list '+ lhs rhs) (Add (parse-sexpr lhs)
                              (parse-sexpr rhs))]
      [(list '- lhs rhs) (Sub (parse-sexpr lhs)
              (parse-sexpr rhs))]
      [(list '* lhs rhs) (Mul (parse-sexpr lhs)
                              (parse-sexpr rhs))]
      [(list '/ lhs rhs) (Div (parse-sexpr lhs)
```

```
                                   (parse-sexpr rhs))]
       [else (error 'parse-sexpr "bad syntax in ~s"
      sexpr)]))

   (: parse : String -> PLANG)
   ;; parses a string containing a PLANG expression
to a PLANG AST    (define (parse str)
     (let ([code (string->sexpr str)])
       <-fill in->))

(test (parse "{{poly 1 2 3} {1 2 3}}")
      => (Poly (list (Num 1) (Num 2) (Num 3))
               (list (Num 1) (Num 2) (Num 3))))
(test (parse "{{poly } {1 2} }")
      =error> "parse: at least one coefficient is
required in ((poly) (1 2))")
(test (parse "{{poly 1 2} {} }")
=error> "parse: at least one point is
                       required in ((poly 1 2) ())")
```

iii. Write the evaluation process. In order to leave the AE eval unchanged (for the sake of keeping your work as simple as possible), we wrap it with an eval-poly function (which will be the core of the evaluator). We start with presenting the formal specification of the semantics:

eval-poly($\{\{\boldsymbol{poly}\ \boldsymbol{C_1}\ \boldsymbol{C_2} \ldots \boldsymbol{C_k}\}\ \{\boldsymbol{P_1}\ \boldsymbol{P_2} \ldots \boldsymbol{P_\ell}\}\}$) =
$'(p(eval(P_1), \ldots, eval(P_\ell)))$ where
p is the polynomial defined by coefficients
$(eval(C_1), \ldots, eval(C_k))$. That is, the expressions $\boldsymbol{C_1}\ \boldsymbol{C_2} \ldots \boldsymbol{C_k}$ are evaluated to coefficients and the expressions $\boldsymbol{P_1}\ \boldsymbol{P_2} \ldots \boldsymbol{P_k}$ are evaluated to numbers on which we evaluate the polynomial.
Here are some possible tests:

```
(test (run "{{poly 1 2 3} {1 2 3}}")
               => '(6 17 34))
(test (run "{{poly 4 2 7} {1 4 9}}")   =>
                '(13 124 589))
(test (run "{{poly 1 2 3} {1 2 3}}")    =>
                '(6 17 34))
(test (run "{{poly 4/5 } {1/2 2/3 3}}")
               => '(4/5 4/5 4/5))
```

```
(test (run "{{poly 2 3} {4}}")  =>
                  '(14))
(test (run "{{poly 1 1 0} {-1 3 3}}")  =>
                  '(0 4 4)))
(test (run "{{poly {/ 4 2} {- 4 1}} {{- 8 4}}}")
                  => '(14))
(test (run "{{poly {+ 0 1} 1 {* 0 9}} {{- 4 5} 3
{/ 27 9}}}")
                  => '(0 4 4))
```

Use the following partial code as a basis for your code.

```
;; evaluates AE expressions to numbers
  (define (eval expr)
    (cases expr
      [(Num n)  n]
      [(Add l r) (+ (eval l) (eval r))]
      [(Sub l r) (- (eval l) (eval r))]
      [(Mul l r) (* (eval l) (eval r))]
      [(Div l r) (/ (eval l) (eval r))]))
  (: eval-poly : PLANG ->  <-fill in-> )
  (define (eval-poly p-expr)
    <-fill in-> )

  (: run : String -> (Listof Number))
  ;; evaluate a FLANG program contained in a
string
  (define (run str)
    (eval-poly (parse str)))
```

HINT: You may want to use the procedure map (twice). See more about map below.

On the procedures map and fold-l

הפונקציה map:

קלט: פרוצדורה proc **ורשימה** lst **פלט: רשימה שמכילה אותו מספר איברים כמו ב-** lst **– שנוצרה ע"י הפעלת הפרוצדורה** proc **על כל אחד מאיברי הרשימה** lst. **(ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה** map **יכולה**

map proc lst ...+) → **לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)**
proc : procedure? (list?
lst : list?
Applies proc to the elements of the lsts from the first elements to the last. The proc argument must accept the same number of arguments as the number of supplied lsts, and all lsts must have the same number of elements. The result is a list containing each result of proc in order. **דוגמאות :**> (map add1 (list 1 2 3 4))
'(2 3 4 5)
> (map (lambda (x) (list x))
'(sym1 sym2 33)) '((sym1) (sym2) (33)) **הפונקציה foldl:**
קלט: פרוצדורה proc, ערך התחלתי init ורשימה lst פלט: ערך סופי (מאותו טיפוס שמחזירה הפרוצדורה proc) שנוצר ע"י הפעלת הפרוצדורה proc על כל אחד מאיברי הרשימה lst תוך שימוש במשתנה ששומר את הערך שחושב עד כה – משתנה זה מקבל כערך התחלתי את הערך של init. (ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה foldl יכולה לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)

(foldl

proc init lst ...+) → any/c
proc : procedure?
init : any/c
lst : list?
Like map, foldl applies a procedure to the elements of one or more lists. Whereas map combines the return values into a list, foldl combines the return values in an arbitrary way that is determined by proc. **דוגמאות :**> (foldl + 0 '(1 2 3 4))
10
> (foldl cons '() '(1 2 3 4))
'(4 3 2 1)