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

Out: Wednesday, April 10, Due: Monday, April 29 , 23:55

Administrative

This is another introductory homework, and again it is for **individual 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 individual 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.

Comments: Submitted code for each question should include at least two lines of comments with your personal description of the solution, the function and its type. In addition, you should comment on the process of solving this question – what were the main difficulties, how you solved them, how much time did you invest in solving it, did you need to consult others. **A solution without proper comments may be graded 0.**

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.

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. Reminder: this means that most of the focus of this homework is put on the contract and purpose statements, good style (indentation, comments, etc), and good tests.

General note: Code quality will be graded. Write clean and tidy code. Consult the Style Guide, and if something is unclear, ask questions on the course forum.

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 <your ID>_1 (e.g., 333333333_1 for a student whose ID number is 333333333).

1. BNF (SE)

a. In class we have seen the grammar for AE – a simple language for "Arithmetic Expressions".

   Write a BNF for "**SE**": a similarly simple language of "String Expressions". Valid 'programs' (i.e., word in the **SE** language) in this language go along the lines of pl expressions for Strings, with two exceptions: 1. Only digits 0,…,9 are allowed as valid characters within strings; 2. We will have two types of expressions that are not available in the pl language (see below 'string-insert' and 'number->string' type expressions). The valid operators that can be used in these expressions are **`string`**, **`string-length`**, and **`string-append`**, and also **`string-insert`** and **`number->string`**. It also possible to have expressions of the form **`"<D>"`**, where **`<D>`** stands for a (finite) sequence of digits. Plain values in the language are characters (of digits) and natural numbers, thus the following are also valid expressions: a sequence of numbers (such as, **`347226`**) and an expression of the form **`#\v`**, where v is a digit.

   Here, we do not care about implementing anything – neither the parser nor the evaluator. Thus, your code should be commented out. Still, it may be helpful to consider some future semantics that will help you understand the requirements. The operations **`" "`**, **`string`**, **`string-append`**, **`string-insert`**, and **`number->string`** are considered expressions that represent a string (they would return a string). The operation **`string-length`**, and digit sequences are considered expressions that

represent a natural number (they would return a natural number). Expression of the form **`#\v`** represent a character.

Note the following requirements for the grammar:

`string` is allowed with a sequence of any number of characters. **`string-append`** is allowed with a sequence of any number of expressions that represent strings. **`string-insert`** is allowed with an expression that represents a string, a character, and a natural number. **`number->string`** is allowed with a natural number.

For example, some **valid** expressions in this language are:

```
"12344"

12

( string #\1 #\2 #\4 )

( string-append ( string #\1 #\2 #\4 ) "12" )

( string-insert "1357" #\4 66 )

( number->string 156879 )

( number->string ( string-length "0033344" ) )

( string-append "45" ( number->string ( string-length
"0033344" ) ))

( string-append )

( string-append "" ( string-insert "1357" #\4 66 ) "" )

#\3
```

but the following are **invalid** expressions:

```
"a2b"

12 13 4 67

( string 124 )

( string-append ( string-length "44" ) "12" )

( string-insert "1357" 4 66 )

( number->string "156879" )

( string-append 33 44 66)

#\3 #\4

#\32
```

```
#\q
```

NOTE: The use of ellipsis ('`...`') or '*' is ***not*** allowed here (find ways within the BNF framework to specify zero-or-more occurrences of a previous piece of syntax). Use λ to specify the empty string.

Important remark: Your solution should only be a BNF and not a code in Racket (or in any other language). You cannot test your code!!! Indeed, your answer should appear inside a comment block (write the grammar in a #|---|# comment).

b. Add to your BNF a derivation process for 3 different **SE** expressions, such that every operator (e.g., **`string-append, string-length, and number->string`**) appears in at least one of these expressions. You may either provide a derivation tree or a series of replacements starting with <SE> and ending with your string. Mark each derivation rule by an index (use "$\overset{(i)}{\Rightarrow}$" to state that in a certain step, you have used rule number i of your BNF).

2. Accommodating Memory Operations

We've talked about the limitation of AE to simple calculations; some calculators use a 'memory' cell to get more "power". Say that you wanted to add just that functionality to your language. A naive attempt at the syntax for this language, call it '**MAE**', is to add a `set` operator that sets the current memory value to some expression result, and a `get` operator to retrieve the current value:

```
<MAE> ::= <num>
        | { + <MAE> <MAE> }
        | { - <MAE> <MAE> }
        | { * <MAE> <MAE> }
        | { / <MAE> <MAE> }
        | { set <MAE> }
        | get
```

Here, the intended meaning for a **`{set E}`** is to evaluate ***E***, store the result in the memory cell, and return it.
There are, however, some problems with this approach.

1. The first problem can be seen if you consider evaluating the following expression:

   ```
   {* {+ {set 1} {set 2}} get}
   ```

   Describe the problem that is demonstrated by this, and suggest a feature to add to the MAE semantics that will solve it. Note that this feature is already present in our AE *implementation*, but it is only implicit there. (The solution is a *short* explanation. You do not need to implement anything. Write it in a single `#|---|#` comment.)

2. Even if the previous problem is fixed, the resulting language is not a good model for the way calculators are used. For example, we may want to extend the language to have a single memory cell for some limited level of abstraction — to compute the area of a 2-foot and 18-inches square in square-feet (recall that there are 12 inches in a foot) we would translate the obvious calculator operations to the following **MAE** syntax (assuming it is fixed as above):

   ```
   {* {set {+ 2 {/ 18 12}}} get}
   ```

   But if you were using a real calculator, you would more likely compute the **`{+ 2 {/ 18 12}}`** term first, store the result in memory, and then compute **`{* get get}`**. In other words, a computation is a non-empty sequence of sub-computations, each one gets stored in the memory and is available for the following sub-computation, except for the last one. Write a new **MAE** grammar that derives such programs. Use a toplevel **`seq`** for such **MAE** programs. To make it more interesting, make it impossible to use **`get`** in the first sub-computation in the sequence, force all expressions except for the last to be **`set`** expressions, and make **`set`** valid only around expressions (not inside them). Examples:

   ```
   ;; valid sequences
   {seq {set {+ 8 7}}
        {set {* get get}}
        {/ get 2}}
   ```

```
{seq {- 8 2}}

;; invalid sequences

{seq {* 8 get}           ; cannot begin with a
`get'

     24}

{seq {* 8 7}             ; must be a `set'

     24}

{seq {set {+ 1 2}}

     {set {- get 2}}}    ; cannot end with a
`set'

{seq {* 2 {set {+ 1 2}}} ; `set' must be outside

     {- get 2}}
```

Hint: you will need a toplevel **<MAE>** for a sequence of computations, then the usual <AE> and a new kind of <AE> (under a different name, of course) that includes a **get** operator.
Note: be careful with this question, you need to get the details right! Remember that a **BNF** grammar is a *formal* piece of text, and as in any code – good style is important here too.
Again, put your solution in a #|---|# comment.

1. **Important remark:** Here again -- your solution should only be a **BNF** and not a code in Racket (or in any other language). You cannot test your code!!! Indeed, your answer should appear inside a comment block (write the grammar in a #|---|# comment).

2. Add to your BNF a derivation process for 3 different **MAE** expressions, in which all plain values are sub-words (of length 2 or 3) of either your name or ID number. E.g., if your ID number is 123456789, then you might want to show how you derive the word

   {seq {set {+ 78 567}}

   {set {* get get}}

   {/ get 23}}

   from your BNF (make sure you use the full power of your BNF).

3. 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 one-liner. 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-of-squares`.
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)
```

4. 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 $a_0, \ldots, a_{k-1}$ 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 $\{\{\ \boldsymbol{poly}\ \boldsymbol{C_1}\ \boldsymbol{C_2}\ \ldots\ \boldsymbol{C_k}\}\ \{\boldsymbol{P_1}\ \boldsymbol{P_2}\ \ldots \boldsymbol{P_\ell}\}\}$ 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}}"
```

   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)))
```

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 לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)**

(map proc lst ...+) → list?
 proc : procedure?
 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)
```
