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

Out: Sunday, April 26, Due: Wednesday, May 13 , 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 the 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):

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>_2 (e.g., 333333333_2.rkt for a student whose ID number is 333333333).

1. BNF and Parsing

In class we have started to come up with the language AE — a simple language for "Arithmetic Expressions". We have already seen the grammar for AE and the appropriate parser (including a new type that we defined). Here we will write a BNF and parser for a new language "**ROL**": a similarly simple language of "Register Operation Expressions". Following examples include some tests for the complete interpreter of this language (including the evaluation part – which we will not do in this assignment). In this question, you will be required to fill in the missing parts in the ROL code **provided at the bottom of the assignment**.

```
;; tests

 (test (run "{ reg-len =  4  {1 0 0 0}}") => '(1 0 0 0))

 (test (run "{ reg-len = 4  {shl {1 0 0 0}}}") => '(0 0 0 1))

 (test (run "{ reg-len = 4  {and {shl {1 0 1 0}}{shl {1 0 1
0}}}}") => '(0 1 0 1))

 (test (run "{ reg-len = 4  { or {and {shl {1 0 1 0}}  {shl
{1 0 0 1}}} {1 0 1 0}}}") => '(1 0 1 1))

 (test (run "{ reg-len = 2  { or {and {shl {1 0}} {1 0}} {1
0}}}") => '(1 0))

 (test (run "{ reg-len = 4  {or {1 1 1 1} {0 1 1}}}") =error>
"wrong number of bits in")

(test (run "{ reg-len =  0  {}}") =error> "Register length must be
at least 1")
```

1.1 BNF for the Register Operation Language ROL

Write a BNF for "**ROL**": a similarly simple language of "Register Operation Expressions". Valid 'programs' (i.e., words in the **ROL** language) in this language should correspond to Racket expressions that evaluate to S-expressions holding lists of zero and ones, and symbols. The valid operation names that can be used in these expressions are **`and`**, **`or`** (which refer to binary operations), and **`shl`** (which refers to unary operation). Plain values are registers, formed as sequences of bits (zeros or ones) rapped with curly parentheses. Any non-zero length of sequences is allowed, however, this length must be appropriately specified in the first part of the program code. Specifically, all programs must be of the form:

"{ reg-len = `len` B}"

where `len` is a natural number, and B is the sequence of Register operations to be performed (it has to be the case that all registers are of length `len`). The BNF code should not impose the bit sequences to be al of length len (this will be later imposed by the parsing function). The BNF should, still, verify that bit sequences are not empty. For example, some **valid** expressions in this language are:

```
"{ reg-len =  4  {1 0 0 0}}"
"{ reg-len = 4  {shl {1 0 0 0}}}"
"{ reg-len = 4  {and {shl {1 0 1 0}}{shl {1 0 1 0}}}}"
"{ reg-len = 4  { or {and {shl {1 0 1 0}}  {shl {1 0 0 1}}} {1 0 1 0}}}"
"{ reg-len = 2  { or {and {shl {1 0}} {1 0}} {1 0}}}"

"{ reg-len = 4  {or {1 1 1 1} {0 1 1}}}"
```

Note that the last expression is not a good code (and should not even pass parsing), but this should not be specified by the BNF. However, the following are **invalid** expressions:

```
"{1 0 0 0}"
"{ reg-len = 3  {and {2 2 1} {0 1 1}}}"
"{ reg-len = 3  {+ {1 1 1} {0 1 1}}}"
"{ reg-len = 4  {shl {1 1 1 1} {0 1 1 1}}}"
"{ reg-len =  0  {}}"
```

a. Write a BNF grammar for the ROL language. In writing your BNF, you can use <num> the same way that it was used. Needless to say – don't use <num> for 0 and 1. You might want to use the following structure:

   #| BNF for the ROL language:

      <ROL> ::=

      <RegE> ::=

      <Bits> ::=

   |#

b. **Important remark:** Your solution here 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).

c. Give 3 examples for ROL program codes and add the appropriate three derivation processes. .

1.2 Parser for the Register Operation Language ROL

a. Introduce a new type definition called RegL which will play the role of the abstract syntax tree for your program. Note that tree should only reflect the structure of the B part in a code of the form "{ reg-len = `len` B}". Therefore, it should be of the form:

```
(define-type RegE

   [Reg  < --fill in-- >]

   [And  < --fill in-- >]

   [Or   < --fill in-- >]

   [Shl  < --fill in-- >] )
```

To see what the variant Reg should be applied to, see the typdef at the top of the provided code.

b. Finally, write the full parser by filling in the missing parts in ROL code provided at the bottom of the assigment. The tests provided within this code do not cover all the code. Make sure that your final code does cover all cases.

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. Typed Racket (and more H.O. functions)

a. We define a *binary tree* as a something that is either a `Leaf` holding a number, or a `Node` that contains a binary tree on the left and one on the right. Write a `define-type` definition for `BINTREE`, with two variants called `Node` and `Leaf`.

b. Using this `BINTREE` definition, write a (higher-order) function called `tree-map` that takes in a numeric function `f` and a binary tree, and returns a

tree with the same shape but using `f(n)` for values in its leaves. For example, here is a test case:

```
(test (tree-map add1 (Node (Leaf 1) (Node (Leaf 2)
(Leaf 3))))

      => (Node (Leaf 2) (Node (Leaf 3) (Leaf 4))))
```

c. Remember: correct type, purpose statement, and tests.

d. Continue to implement `tree-fold`, which is the equivalent of the swiss-army-knife tool that `foldl` is for lists. There are two differences between `tree-fold` and `foldl`: **first**, `foldl` has a single `init` argument that determines the result for the single empty list value. But with our trees, the base case is a `Leaf` that holds some number — so we use a (numeric) *function* instead of a constant. **The second** difference is that `tree-fold`'s combiner function receives different inputs: the two results for the two subtrees. `tree-fold` should therefore consume three values — the combiner function (a function of two arguments), the leaf function (a function of a single number argument), and the `BINTREE` value to process. Note also that `tree-fold`'s type is slightly simpler than `foldl`'s type — because we have only trees of numbers — but don't confuse that with the output type, which *does not* have to be a number (or a list of numbers).
Note: `tree-fold` is a *polymorphic* function, so its type should be parameterized over "some input type `A`". The Typed Racket notation for this is

```
(: tree-fold : (All (A) ...type that uses A...))
```

Hint: The type A is also the type of the returned value of `tree-fold (as opposed to foldl, here, no extra polymorphic type B is required, since the 'inner type' of a Leaf is always Number).`

e. When your implementation is complete, you can use it, for example, to implement a `tree-flatten` function that consumes a `BINTREE` and returns a list of its values from left to right:

```
(: tree-flatten : BINTREE -> (Listof Number))

;; flattens a binary tree to a list of its values in
```

```
;; left-to-right order

(define (tree-flatten tree)

  (tree-fold (inst append Number) (inst list Number)
tree))
```

f. You can use this function, as well as others you come up with, to test your code.
Note the use of `(inst f Number)` — the Typed Racket inference has certain limitations that prevent it from inferring the correct type. We need to 'help' it in these cases, and say explicitly that we use the two polymorphic functions `append` and `list` instantiated with `Number`. (Think about an `(All (A) ...)` type as a kind of a function at the type world, and `inst` is similar to calling such a function with an argument for `A`.) You will not need to use this elsewhere in your answers.

g. Use `tree-fold` to define a `tree-reverse` function that consumes a tree and returns a tree that is its mirror image. That is, for any tree *t*, the following equation holds:

```
(equal? (reverse (tree-flatten t))

        (tree-flatten (tree-reverse t)))
```

You can use it in your tests (but use `test` for the test). If you do things right, this should be easy using a one-line helper definition (you may call it `switch-nodes`).

To recap: In this question you were asked to define the **`BINTREE`** type, and the following procedures **tree-map**, **tree-fold**, and **tree-reverse**. Don't forget to add comments and tests.

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)
```

The ROL BNF and Parsing (incomplete) code

```
; The ROL BNF and Parsing code:
#lang pl
;; Defining two new types
(define-type BIT = (U 0 1))
(define-type Bit-List = (Listof BIT))




;; The actual interpreter
#| BNF for the RegE language:
      <ROL>  ::= < --fill in-- >
      <RegE> ::= < --fill in-- >

      <Bits> ::= < --fill in-- >
  |#

;; RegE abstract syntax trees
(define-type RegE
    [Reg  < --fill in-- >]
    [And  < --fill in-- >]
    [Or   < --fill in-- >]
    [Shl  < --fill in-- >])

;; Next is a technical function that converts (casts)
;; (any) list into a bit-list. We use it in parse-sexpr.
  (: list->bit-list : (Listof Any) -> Bit-List)
  ;; to cast a list of bits as a bit-list
  (define (list->bit-list lst)
    (cond [(null? lst) null]
          [(eq? (first lst) 1)(cons 1 (list->bit-list (rest lst)))]
          [else (cons 0 (list->bit-list (rest lst)))]))

  (: parse-sexpr : Sexpr -> RegE)
  ;; to convert the main s-expression into ROL
  (define (parse-sexpr sexpr)
    (match sexpr
      < --fill in-- > ;; remember to make sure specified register length is at least 1
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

  (: parse-sexpr-RegL : Sexpr Number -> RegE)
  ;; to convert s-expressions into RegEs
```

```
(define (parse-sexpr-RegL sexpr reg-len)
  (match sexpr
    [(list (and a (or 1 0)) ... ) (< --fill in-- >
                                   (error 'parse-sexpr "wrong number of bits in ~s" a))]
    [< --fill in-- >]
    [< --fill in-- >]
    [< --fill in-- >]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> RegE)
;; parses a string containing a RegE expression to a RegE AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))




;; tests
(test (parse "{ reg-len =  4  {1 0 0 0}}") => (Reg '(1 0 0 0)))
(test (parse "{ reg-len = 4  {shl {1 0 0 0}}}") => (Shl (Reg '(1 0 0 0))))
(test (parse "{ reg-len = 4  {and {shl {1 0 1 0}} {shl {1 0 1 0}}}}") => (And (Shl (Reg
'(1 0 1 0))) (Shl (Reg '(1 0 1 0)))))
(test (parse "{ reg-len = 4  { or {and {shl {1 0 1 0}} {shl {1 0 0 1}}} {1 0 1 0}}}") =>
(Or (And (Shl (Reg '(1 0 1 0))) (Shl (Reg '(1 0 0 1)))) (Reg '(1 0 1 0))))
(test (parse "{ reg-len = 2  { or {and {shl {1 0}} {1 0}} {1 0}}}") => (Or (And (Shl
(Reg '(1 0))) (Reg '(1 0))) (Reg '(1 0))))
(test (parse "{ reg-len = 4  {or {1 1 1 1} {0 1 1}}}") =error> "wrong number of bits in")
```