Homework #2: Two Basic Inerpreters

Out: Thursday, May 04, 2017, Due: Thursday, May 18, 23:55

Administrative

This is a basic implementation homework, and it is for **individual work and submission**. In this homework you will have a chance to use the ideas shown in class for the AE language and write the code for two additional languages.

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
```

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 work and submission in singles.

Integrity: Please do not cheat. You may consult your friends 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.

Important – If you consult any person or any other source – You must indicate it within your personal comments. Also the contribution of each of you to the solution must be detailed.

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.

VERY IMPORTANT NOTE:

A solution without proper and elaborate PERSONAL comments describing your work process 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>_3 (e.g., 222222222_3 for a student whose ID numbers is 222222222).

1. Extending the AE language

In class we have seen the interpreter for the AE language – a simple language for "Arithmetic Expressions". The interpreter appears at the end of this file for your convenience.

In this question you are asked to slightly change the syntax of the language and slightly extend it to also deal with exponentiations. Specifically, you are asked to change the syntax to be in postfix form, and in addition to add two operators: **`power`** and **`sqr`**.

As you are also expected to implement

The semantics for **`power`** and **`sqr`** expressions in the extended language are as follows:

- **`power`** should be evaluated to the binary (two argument) function, that takes the first argument and raises it to the power of the second argument. Thus, the second argument must be an integer. Note that in Racket, if you ask whether a

number is an integer and receive a positive answer, then the type checker will treat it an integer (hence, you will be able to send it to a function that expects an integer). You may want to write an auxiliary function *power* (in pl) that will take two number arguments, and will return #f if the second one cannot be cast as an integer and the first to the power of the second otherwise.

- **`sqr`** should be evaluated to the unary (a single argument) function, that take a number and raises it to the power of 2.

For example, the following tests should work:

```
(test (run "3") => 3)
  (test (run "{3 4 +}") => 7)
  (test (run "{{3 4 -} 7 +}") => 6)
  (test (run "{{3 4 power} 7 +}") => 88)
  (test (run "{{2 4 power} {5 sqr} +}") => 41)
 (test (run "{{2 4/5 power} {5 sqr} +}")
    =error> "eval: power expects an integer power, got")
```

2. Interpreter for the LE language.

In the previous homework, you have written a BNF for the **LE** language. In this work, you will complete the full interpreter for this language (this is yet another very basic interpreter).

2.a Parser for the LE language.

1. Define a new data type for the appropriate AST. You may use the following (incomplete) definitions:

```
;; LE abstract syntax trees
(define-type LE = (U LIST ATOM))

;; LIST abstract syntax trees
(define-type LIST
  <<--fill in-->>)

;; ATOM abstract syntax trees
(define-type ATOM
  <<--fill in-->>)
```

2. Complete the LE parser accordingly. The result should be a function `parseLE`. You may use the following (incomplete) definitions:

```
(: parse-sexpr->LEs : (Listof Sexpr) -> (Listof LE))
;; converts a list of s-expressions into a list of LEs
  (define (parse-sexpr->LEs sexprs)
    (map parse-sexprLE sexprs))


(: parse-sexpr->LISTs : (Listof Sexpr) -> (Listof
LIST))
;; converts a list of s-exprs into a list of LISTs
  (define (parse-sexpr->LISTs sexprs)
    <<--fill in-->>)
```

```
(: parse-sexpr->LIST : Sexpr -> LIST)
    (define (parse-sexpr->LIST sexpr)
      (let ([ast (parse-sexprLE sexpr)])
        (if (LIST? ast)
            ast
            (error 'parse-sexprLE "expected LIST; got
~s" ast))))

(: parse-sexprLE : Sexpr -> LE)
;; to convert s-expressions into LEs
(define (parse-sexprLE sexpr)
  (match sexpr
    [(number: n) <<--fill in-->>]
    ['null <<--fill in-->>]
    [(symbol: s) <<--fill in-->>]
    [<<--fill in-->>]
    [<<--fill in-->>]
    [<<--fill in-->>]
    [else <<--fill in-->>]))


(: parseLE : String -> LE)
  ;; parses a string containing a LE expression to a
  ;; LE AST
(define (parseLE str)
  (parse-sexprLE (string->sexpr str)))
```

Remark: You are not obliged to utilize the above skeleton, but you must write your own parser. Use procedure names that do not coincide with the names you used for the AE interpreter, as the two solutions need to appear in the same .rkt file.

2.a Evaluator for the LE language.

3. Define an appropriate `evalLE` procedure to complete the interpreter.

   Your code must comply with the following formal specifications:

```
#| Formal specs for `eval':
     eval(N)          = N ;; for numbers
     eval(Sym)        = 'Sym ;; for symbols
     eval({list E ...}) = (list eval(E) ...)
     eval({cons E1 E2}) = if eval(E2) = (list E), then
                            (cons eval(E1) eval(E2))
                          else error
     eval({append E ...}) =
         if eval(E) = (list E') for all expressions E, then
                              (append eval(E) ...)
|#
```

You may use the following (incomplete) definitions:

```
(: eval-append-args : (Listof LE) -> (Listof (Listof Any)))
  ;; evaluates LE expressions by reducing them to lists
  (define (eval-append-args exprs)
    (if (null? exprs)
        null
        (let ([fst-val (evalLE (first exprs))])
          (if <<--fill in-->>
              (cons <<--fill in-->>)
              (error 'evalLE "append argument: expected
                                  List got ~s" fst-val)))))



  (: evalLE : LE -> Any)
  ;; evaluates LE expressions by reducing them to numbers
  (define (evalLE expr)
    (if (LIST? expr)
        (cases <<--fill in-->>
          [<<--fill in-->> (map evalLE <<--fill in-->>)]
          [<<--fill in-->>]
          [<<--fill in-->> (apply append <<--fill in-->>)])
        (cases <<--fill in-->>
          [<<--fill in-->>]
          [<<--fill in-->>])))
```

```
(: runLE : String -> Any)
;; evaluate a WAE program contained in a string
(define (runLE str)
  (evalLE (parseLE str)))
```

Remark: You are not obliged to utilize the above skeleton, but you must write your own evaluator. Use procedure names that do not coincide with the names you used for the AE interpreter, as the two solutions need to appear in the same .rkt file.

In your code, you may wish to use the map and apply procedures of Racket, which are specified below.

The following tests should work for your code.

```
(test (runLE "null") => null)

(test (runLE "12") => 12)

(test (runLE "boo") => 'boo)

(test (runLE "{cons 1 {cons two null}}") => '(1 two))

(test (runLE "{list 1 2 3}") => '(1 2 3))


(test (runLE "{list {cons}}") =error> "parse-sexprLE:
bad syntax in (cons)")

(test (runLE "{list {cons 2 1}}") =error> "parse-
sexprLE: expected LIST; got")
```

The <AE> interpreter – for your convenience.

```
#lang pl

#| BNF for the AE language:
   <AE> ::= <num>
          | { + <AE> <AE> }
          | { - <AE> <AE> }
          | { * <AE> <AE> }
          | { / <AE> <AE> }
|#  ;; AE abstract syntax trees
```

```
(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 -> AE)
;; parses a string containing an AE expression to AE AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

(: eval : AE -> Number)
;; consumes an AE and computes the corresponding number
(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))]))

(: run : String -> Number)
;; evaluate an AE program contained in a string
(define (run str)
  (eval (parse str)))

;; tests
(test (run "3") => 3)
(test (run "{+ 3 4}") => 7)
(test (run "{+ {- 3 4} 7}") => 6)
```

procedure

(**apply** *proc v* ... *lst* #:<kw> *kw-arg* ...) → any
proc : procedure?
v : any/c
lst : list?
kw-arg : any/c

☞ The apply Function in **The Racket Guide** introduces apply.

Applies *proc* using the content of (list* *v* ... *lst*) as the (by-position) arguments. The #:<kw> *kw-arg* sequence is also supplied as keyword arguments to *proc*, where #:<kw> stands for any keyword.

The given *proc* must accept as many arguments as the number of *v*s plus length of *lst*, it must accept the supplied keyword arguments, and it must not require any other keyword arguments; otherwise, the exn:fail:contract exception is raised. The given *proc* is called in tail position with respect to the apply call.

Examples:

```
> (apply + '(1 2 3))
6
> (apply + 1 2 '(3))
6
> (apply + '())
0
> (apply sort (list (list '(2) '(1)) <) #:key car)
'((1) (2))
```

---

procedure

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

Applies *proc* to the elements of the *lst*s from the first elements to the last.

The *proc*argument must accept the same number of arguments as the number of supplied *lst*s, and all *lst*s must have the same number of elements. The result is a list containing each result of *proc* in order.

Examples:

```
> (map add1 '(1 2 3 4))


'(2 3 4 5)
> (map (lambda (number1 number2)
        (+ number1 number2))
      '(1 2 3 4)
      '(10 100 1000 10000))
'(11 102 1003 10004)
```
