Homework #3: Interpreter for The Register Language

Out: Tuesday, May 7, 2019, Due: ~~Wednesday, May 22~~, Friday, May 24, 2019, 19:00

Administrative

The language for this homework is:

```
#lang pl 03
```

The homework is basically a completion of the implementation of the simple ROL language that we have started to implement in the previous assignment. In addition it handles the topic of free variables in a WAE program.

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

Important: the grading process requires certain bound names to be present. These names need to be global definitions.

This homework is for individual work and submission.

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.

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 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.

Note: 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 code for all the following questions should appear in a single .rkt file named <your ID>_3 (e.g., 333333333_3 for a student whose ID number is 333333333).

1. Adding with expressions to the ROL language

- In the practical session, you have started to come up with the language “**ROL**” – a simple language for “Register Operation Expressions”. You have already implemented a simple parser for this language. In this part you will extend your parser (as well as BNF and RegE abstract syntax tree) to also support "if" expression, two Boolean function “geq?” and “maj?” and Boolean types. “geq “return true if given two registers, the first register's (binary) value is greater or equal to the value of the second register, “maj?” returns true if the majority of bits in the register are ones. Following examples include some tests for the complete interpreter of this language (including the evaluation part – which we will only complete in subsequent sections).

```
;; 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))
```

```
;; 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 {with {x {1 1 1 1}} {shl y}}}")
        =error> "free identifier: y")
(test (run "{ reg-len = 2
                 { with {x { or {and {shl {1 0}}
                                    {1 0}}
                                {1 0}}}
                      {shl x}}}") => '(0 1))
(test (run "{ reg-len = 4  {or {1 1 1 1} {0 1 1}}}") =error>
            "wrong number of bits in (0 1 1)")
(test (run "{ reg-len = 0 {}}") =error>
            "Register length must be at least 1")
(test (run "{ reg-len = 3
               {if {geq? {1 0 1} {1 1 1}}
                   {0 0 1}
                   {1 1 0}}}") => '(1 1 0))
(test (run "{ reg-len = 4
                 {if {maj? {0 0 1 1}}
                     {shl {1 0 1 1}}
                     {1 1 0 1}}}")  => '(0 1 1 1))
(test (run "{ reg-len = 4
                 {if false {shl {1 0 1 1}} {1 1 0 1}}}") =>
          '(1 1 0 1))
```

- Complete the BNF grammar for the ROL language in the following partial code provided to you as a skeleton (copy this code to your solution file. In later sections, you will need to augment it with additional functions). In writing your BNF, you can use <num> the same way that it is used in the languages we wrote in class. 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> ::=
  |#


;; Defining two new types
(define-type BIT = (U 0 1))

(define-type Bit-List = (Listof BIT))


;; RegE abstract syntax trees
(define-type RegE
   [Reg  <--fill in -->]
   [And  <--fill in -->]
   [Or   <--fill in -->]
   [Shl  <--fill in -->]
   [Id   <--fill in -->]
   [With <--fill in -->]
   [Bool <--fill in -->]
   [Geq <--fill in -->]
   [Maj <--fill in -->]
   [If <--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)))
```

2. Supported Types

Note that, our new language allows for two types of outputs. Boolean and register (unlike the WAE language, which only allowed the output to be a number). Define a new type RES that has two variants, one for each output type. Complete the code…

```
(define-type RES
   [<--fill in-->]
   [<--fill in-->])
```

3. Substitutions

Using the following formal specifications (and the example of our WAE interpreter), write a function

(: subst : RegE Symbol RegE -> RegE)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
….

```
  #| Formal specs for `subst':

     (`BL' is a Bit-List, `E1', `E2' are <RegE>s, `x' is some
<id>, `y' is a *different* <id>, 'BOOL' is Boolean )

        BL[v/x]                 = BL

        BOOL[v/x]                 = BOOL

        {and E1 E2}[v/x]      = {and E1[v/x] E2[v/x]}

        {or E1 E2}[v/x]       = {or E1[v/x] E2[v/x]}

        {shl E}[v/x]          = {shl E[v/x]}

        y[v/x]                = y

        x[v/x]                = v

        {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}

        {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}

       {if {E1} E2 E3}[v/x] = {if {E1[v/x]} E2[v/x] E3[v/x]}

       {maj? E1}[v/x] = {maj? E1[v/x] }

       {geq? E1 E2}[v/x] = {geq? E1[v/x] E2[v/x]}

  |#
```

4. Evaluation

Using the following formal specifications (and the example of our WAE interpreter), write a function

(: eval : RegE -> RES)
;; evaluates RegE expressions by reducing them to bit-lists
(define (eval expr)
(cases expr
[< --fill in-- >]
[< --fill in-- >]
…
[< --fill in-- >]))

```
  #| Formal specs for `eval':
       eval(Reg)    = Reg
       eval(bl)     = bl
       eval(true)   = true
       eval(false)  = false
       eval({and E1 E2}) =
                  (<x1 bit-and y1> <x2 bit-and y2> ... <xk bit-
and yk>),
            where eval(E1) = (x1 x2 ... xk)
                  and eval(E2) = (y1 y2 ... yk)
        eval({or E1 E2}) =
            (<x1 bit-or y1> <x2 bit-or y2> ... <xk bit-or yk>,)
            where eval(E1) = (x1 x2 ... xk)
                  and eval(E2) = (y1 y2 ... yk)
        eval({shl E}) = (x2 ... xk x1), where eval(E) = (x1 x2
... xk)
     eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
     eval({if E1 E2 E3})
             = eval(E3)   if eval(E1) = false
             = eval(E2)   otherwise


    eval({maj? E}) = true if x1+x2+...+xk >= k/2, and false
otherwise,
                                        where eval(E) = (x1 x2
... xk)
    eval({geq? E1 E2}) = true if x_i >= y_i,
            where eval(E1) = (x1 x2 ... xk)
                  and eval(E2) = (y1 y2 ... yk)
                  and i is the first index s.t. x_i and y_i are
not equal
                  (or i =k if all are equal)
   eval({if Econd  Edo Eelse})
                       = eval(Edo) if eval(Econd) =/= false,
                       = eval(Eelse),    otherwise.
```

A REMARK ABOUT THE IMPLEMENTATION OF if EXPRESSIONS:

Note that if should behave very much like in Racket – where it is always the case that either eval(Edo) is performed or eval(Eelse) is performed – but never both!! In addition, the condition is considered not to hold only if it evaluates to false, and true otherwise (even if it evaluates to a non-Boolean).

Towards coming up with the implementation of the eval function, you need to complete the code for the following functions:

```
;; Defining functions for dealing with arithmetic operations
;; on the above types
  (: bit-and : BIT BIT -> BIT) ;; Arithmetic and
  (define(bit-and a b)
    < --fill in-- >)

  (: bit-or : BIT BIT -> BIT) ;; Aithmetic or
  (define(bit-or a b)
    < --fill in-- >)


(: reg-arith-op : (BIT BIT -> BIT)  RES  RES ->  RES)
    ;; Consumes two registers and some binary bit operation 'op',
    ;; and returns the register obtained by applying op on the
    ;; i'th bit of both registers for all i.
  (define(reg-arith-op op reg1 reg2)
    (: bit-arith-op : Bit-List Bit-List -> Bit-List)
     ;; Consumes two bit-lists and uses the binary bit operation 'op'.
     ;; It returns the bit-list obtained by applying op on the
     ;; i'th bit of both registers for all i.
    (define(bit-arith-op bl1 bl2)
       < --fill in-- >
    (RegV (bit-arith-op  (RegV->bit-list reg1) (RegV->bit-list reg2))))


 (: majority? : Bit-List -> Boolean)
    ;; Consumes a list of bits and checks whether the
    ;; number of 1's are at least as the number of 0's.
(define(majority? bl)
```

```
  < --fill in-- >)


(: geq-bitlists? : Bit-List Bit-List -> Boolean)
  ;; Consumes two bit-lists and compares them. It returns true if the
  ;; first bit-list is larger or equal to the second.
(define (geq-bitlists? bl1 bl2)
 < --fill in-- >)

(: shift-left : Bit-List -> Bit-List)
   ;; Shifts left a list of bits (once)
 (define(shift-left bl)
    < --fill in-- >)
(: RegV->bit-list : RES -> Bit-List)
   ;; extract a bit-list from RES type
 < --fill in-- >)
```

5. Interface

Wrap it all up by writing the function

```
(: run : String -> Bit-List)
 ;; evaluate a ROL program contained in a string
 ;; we will not allow to return a boolean type
….
```