Homework #4: Interpreter for The Register Operation Language (ROL)

Out: Saturday, May 23, 2017, Due: Tuesday, June 6, 2017, 23:55

Administrative

The language for this homework is:

```
#lang pl 04
```

The homework is about implementing a new and simple language – ROL – that allows for basic bit-vector manipulations.

Important: the grading process requires certain bound names to be present. These names need to be global definitions. The grading process *cannot* see names of locally defined functions.

This homework is for work and submission in pairs (individual submissions are also fine). If you choose to work in pairs:

1. **Make sure to specify the ID number of both partners**
2. **Submit the assignment only once.**
3. **Make sure to write in your comments what was the role of each partner in each solution.**

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 (as pairs, if you so choose)!! 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.**

If you choose to consult any other person or source of information, this **must be** clearly declared in your comments.

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

Introduction – Register Operation Language ROL

In this work, you will implement a new, very simple language. This language should be written in the substitution mode – as a point of reference, you should use the FLANG interpreter that appears at the bottom of this file (and also at the end of "Lecture 08 - First Class Function"). To understand what programs in this language do, you should think of the equivalent of FLANG, where instead of numbers – we now work with bit-vectors. In addition, instead of basic binary arithmetic operations on numbers, we now work with bit-vector logic operations, such as the binary "and" and "or", and the unary "shl" (shift-left, or rotate-left). We will also allow "with" expressions, as well as "fun" and "call" expressions.

Following are tests that should work for your final code:

```
;; 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")

  (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")

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

(test (run "{ reg-len = 3

                 {with {identity {fun {x} x}}

                    {with {foo {fun {x} {or x {1 1 0}}}}

                       {call {call identity foo} {0 1 0}}}}}")
```

1. BNF for the Register Operation Language ROL

Write a BNF for "**ROL**": a simple language of "Register Operation Language". 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 = 2  { with { f {fun {x} {or {and {shl x} {1 0}} {1 0}}}}
                    { call f {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. 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. Later you will augment it with additional functions). 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:

```
;; Defining two new types
(define-type BIT = (U 0 1))
(define-type Bit-List = (Listof BIT))


;; The actual interpreter
#| BNF for the ROL language:
     <ROL>  ::=
```

```
      <RegE> ::=

      <Bits> ::=
  |#

(define-type RegE
    [Reg  < --fill in-- >]
    [And  < --fill in-- >]
    [Or   < --fill in-- >]
    [Shl  < --fill in-- >]
    [< --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. Parser for the Register Operation Language ROL

a. Introduce a new type definition called **RegE** (use the above code), which will play the role of the abstract syntax tree for your program. Note that the 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 as in the partial code provided above.

3. Substitutions

Using the following formal specifications (and the example of our FLANG 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>)

        BL[v/x]                 = BL

        {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}

  |#
```

4. Evaluation

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

(: eval : RegE -> RegE)
;; evaluates RegE expressions by reducing them to RegE that is
;; of the two variants (Reg..) or (Fun..) ….

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

  eval(bl)          = bl

  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(id)         = error!

 eval({with {x E1} E2}) = eval(E2[eval(E1)/x])

 eval(FUN) = FUN ;     assuming FUN is a function expression

 eval({call E1 E2}) = eval(Ef[eval(E2)/x])

                              if eval(E1)={fun {x} Ef}

                    = error!   otherwise

  |#
```

Note: The shl operation is a unary operation on bit-vectors that takes each of the bit one spot left to its current position, except for the currently left most one, which becomes – right most bit.

For this purpose, you need to complete the code for the following function:

```
(: reg-arith-op : (BIT BIT -> BIT) RegE RegE -> RegE)
  ;; 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)
....
```

You may want to allow this function to use an externally defined function with declaration:

```
(: bit-arith-op : (BIT BIT -> BIT) Bit-List Bit-List -> Bit-List)
```

5. Interface

Wrap it all up by writing the function

```
(: run : String -> Bit-List)
  ;; evaluate a ROL program contained in a string
....
```

; The FLANG Interpreter – substitution model. Provided as a point of reference.

```
; --<<<FLANG>>>-------------------------------------------------------


  ;; The Flang interpreter

  #lang pl

  #|
  The grammar:
    <FLANG> ::= <num>
              | { + <FLANG> <FLANG> }
              | { - <FLANG> <FLANG> }
              | { * <FLANG> <FLANG> }
              | { / <FLANG> <FLANG> }
              | { with { <id> <FLANG> } <FLANG> }
              | <id>
```

```
              | { fun { <id> } <FLANG> }
              | { call <FLANG> <FLANG> }

  Evaluation rules:

    subst:
      N[v/x]                 = N
      {+ E1 E2}[v/x]         = {+ E1[v/x] E2[v/x]}
      {- E1 E2}[v/x]         = {- E1[v/x] E2[v/x]}
      {* E1 E2}[v/x]         = {* E1[v/x] E2[v/x]}
      {/ E1 E2}[v/x]         = {/ E1[v/x] E2[v/x]}
      y[v/x]                 = y
      x[v/x]                 = v
      {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]} ; if y =/= x
      {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
      {call E1 E2}[v/x]     = {call E1[v/x] E2[v/x]}
      {fun {y} E}[v/x]      = {fun {y} E[v/x]}           ; if y =/= x
      {fun {x} E}[v/x]      = {fun {x} E}

    eval:
      eval(N)            = N
      eval({+ E1 E2})    = eval(E1) + eval(E2)  \ if both E1 and E2
      eval({- E1 E2})    = eval(E1) - eval(E2)   \ evaluate to numbers
      eval({* E1 E2})    = eval(E1) * eval(E2)   / otherwise error!
      eval({/ E1 E2})    = eval(E1) / eval(E2)  /
      eval(id)           = error!
      eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
      eval(FUN)          = FUN ; assuming FUN is a function expression
      eval({call E1 E2}) = eval(Ef[eval(E2)/x]) if eval(E1)={fun {x}
Ef}
                         = error!               otherwise
  |#

  (define-type FLANG
    [Num  Number]
    [Add  FLANG FLANG]
    [Sub  FLANG FLANG]
    [Mul  FLANG FLANG]
    [Div  FLANG FLANG]
    [Id   Symbol]
    [With Symbol FLANG FLANG]
    [Fun  Symbol FLANG]
    [Call FLANG FLANG])

  (: parse-sexpr : Sexpr -> FLANG)
  ;; to convert s-expressions into FLANGs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)     (Num n)]
      [(symbol: name)  (Id name)]
      [(cons 'with more)
       (match sexpr
         [(list 'with (list (symbol: name) named) body)
          (With name (parse-sexpr named) (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
      [(cons 'fun more)
       (match sexpr
```

```
      [(list 'fun (list (symbol: name)) body)
       (Fun name (parse-sexpr body))]
      [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(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))]
    [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

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

(: subst : FLANG Symbol FLANG -> FLANG)
;; 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
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]
    [(Call l r) (Call (subst l from to) (subst r from to))]
    [(Fun bound-id bound-body)
     (if (eq? bound-id from)
       expr
       (Fun bound-id (subst bound-body from to)))]))

(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper
(define (arith-op op expr1 expr2)
  (: Num->number : FLANG -> Number)
  (define (Num->number e)
    (cases e
      [(Num n) n]
      [else (error 'arith-op "expects a number, got: ~s" e)]))
  (Num (op (Num->number expr1) (Num->number expr2))))

(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions*
(define (eval expr)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l) (eval r))]
    [(Sub l r) (arith-op - (eval l) (eval r))]
```

```
    [(Mul l r) (arith-op * (eval l) (eval r))]
    [(Div l r) (arith-op / (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (eval named-expr)))]
    [(Id name) (error 'eval "free identifier: ~s" name)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr)])
       (cases fval
         [(Fun bound-id bound-body)
          (eval (subst bound-body
                       bound-id
                       (eval arg-expr)))]
         [else (error 'eval "`call' expects a function, got: ~s"
                      fval)]))]))

(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str))])
    (cases result
      [(Num n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))

;; tests
(test (run "{call {fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {call add3 1}}")
      => 4)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {with {add1 {fun {x} {+ x 1}}}
                {with {x 3}
                  {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x} x}}
              {with {foo {fun {x} {+ x 1}}}
                {call {call identity foo} 123}}}")
      => 124)
(test (run "{call {call {fun {x} {call x 1}}
                       {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
----------------------------------------------------------------------
```