```
#lang eopl

;;; An interpreter for the DYNAMIC-PROC language

;;; Daniel P. Friedman (dfried@cs.indiana.edu)
;;; Mitchell Wand (wand@ccs.neu.edu)
;;; From _Essentials of programming languages_, third edition (Cambridge,
;;;     Massachusetts: The MIT Press, 2008; ISBN 978-0-262-06279-4).

;;; John David Stone
;;; Department of Computer Science
;;; Grinnell College
;;; stone@cs.grinnell.edu

;;; Fiona Byrne & Tyler Dewey
;;; Certified killing it
;;; 14 September 2015

;;; Tyler Dewey
;;; Modified for multi-argument procs
;;; with default values
;;; October 3, 2015

;;; last revised October 5, 2015

(require "expvals-and-environments.scm")
(require "parser.scm")
(require "syntax-trees.scm")

;; run : String -> ExpVal
(define run
  (lambda (string)
    (value-of-program (scan&parse string))))

;; value-of-program : Program -> ExpVal
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))

;; value-of : Exp * Env -> ExpVal
(define value-of
  (lambda (exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (var-exp (var) (apply-env env var))
      (diff-exp (exp1 exp2)
        (let ((val1 (value-of exp1 env))
              (val2 (value-of exp2 env)))
          (let ((num1 (expval->num val1))
                (num2 (expval->num val2)))
            (num-val (- num1 num2)))))
      (zero?-exp (exp1)
        (let ((val1 (value-of exp1 env)))
          (let ((num1 (expval->num val1)))
            (if (zero? num1)
                (bool-val #t)
                (bool-val #f)))))
      (if-exp (exp1 exp2 exp3)
        (let ((val1 (value-of exp1 env)))
          (if (expval->bool val1)
              (value-of exp2 env)
              (value-of exp3 env))))
      (let-exp (var exp1 body)
        (let ((val1 (value-of exp1 env)))
          (value-of body (extend-env var val1 env))))
      (proc-exp (standard-vars optional-var-pairs body)
        (let* ((optional-vars (map car optional-var-pairs))
               (optional-var-values (map (lambda (p) (value-of (cdr p) env))
                                         optional-var-pairs))
               (vars (append standard-vars optional-vars)))
          (proc-val (a-proc vars
                            (length standard-vars)
                            body
                            (extend-env* optional-vars optional-var-values env)))))
      (call-exp (operator operands)
        (let ((proc (expval->proc (value-of operator env)))
              (args (map (lambda (operand) (value-of operand env))
                         operands)))
          (apply-procedure proc args env))))))

;;; The apply-procedure procedure
;;; applies the procedure represented by a given proc
;;; to a given value
;;; by evaluating the proc's body in an environment
;;; obtained by adding the binding for the proc's parameter
;;; to its stored environment.

;; apply-procedure : Proc * ExpVal -> ExpVal
(define apply-procedure
  (lambda (applicand arguments env)
    (cases proc applicand
      (a-proc (parameters min-arity body saved-env)
        (let ((num-args (length arguments)))
          (if (>= num-args min-arity)
              (value-of body (extend-env* parameters
                                          arguments
                                          saved-env))
              (report-arity-error min-arity num-args)))))))

;; report-apply-procedure-error : String -> ()
(define report-arity-error
  (lambda (expected found)
    (eopl:error 'apply-procedure
                "arity mismatch: expected at least ~a arguments, found ~a.~%"
                expected
                found)))

;;; Tests

(require "../test.scm")
(require "../character-sources.scm")
(require "scanner.scm")

;;; Constants

(test 0 (equal? (value-of (const-exp 0) (empty-env))
                (num-val 0)))
(test 1 (equal? (value-of-program (a-program (const-exp 0)))
                (num-val 0)))
(test 2 (equal? (run "0") (num-val 0)))

;;; Variables

(test 3 (equal? (value-of (var-exp 'alpha)
                          (extend-env 'alpha (num-val 1) (empty-env)))
                (num-val 1)))
(test 4 (equal? (value-of-program (a-program (var-exp 'i)))
                (num-val 1)))
(test 5 (equal? (run "v") (num-val 5)))

;;; Diff-expressions

(test 6 (equal? (value-of (diff-exp (const-exp 2) (const-exp 3))
                          (empty-env))
                (num-val -1)))
(test 7 (equal? (value-of-program
```

```
                (a-program (diff-exp (var-exp 'x) (const-exp 4))))
              (num-val 6)))
(test 8 (equal? (run "-(i, 5)") (num-val -4)))

;;; Zero?-expressions

(test 9 (equal? (value-of (zero?-exp (const-exp 0)) (empty-env))
              (bool-val #t)))
(test 10 (equal? (value-of (zero?-exp (const-exp 6)) (empty-env))
                 (bool-val #f)))
(test 11 (equal? (value-of-program
                   (a-program (zero?-exp (const-exp 0))))
                 (bool-val #t)))
(test 12 (equal? (value-of-program
                   (a-program (zero?-exp (const-exp 6))))
                 (bool-val #f)))
(test 13 (equal? (run "zero?(0)") (bool-val #t)))
(test 14 (equal? (run "zero?(6)") (bool-val #f)))

;;; If-expressions

(test 15 (equal? (value-of (if-exp (zero?-exp (const-exp 0))
                                   (const-exp 7)
                                   (const-exp 8))
                           (empty-env))
                 (num-val 7)))
(test 16 (equal? (value-of (if-exp (zero?-exp (const-exp 9))
                                   (const-exp 10)
                                   (const-exp 11))
                           (empty-env))
                 (num-val 11)))
(test 17 (equal? (value-of-program
                   (a-program (if-exp (zero?-exp (const-exp 0))
                                      (const-exp 7)
                                      (const-exp 8))))
                 (num-val 7)))
(test 18 (equal? (value-of-program
                   (a-program (if-exp (zero?-exp (const-exp 9))
                                      (const-exp 10)
                                      (const-exp 11))))
                 (num-val 11)))
(test 19 (equal? (run "if zero?(0) then 7 else 8") (num-val 7)))
(test 20 (equal? (run "if zero?(9) then 10 else 11") (num-val 11)))

;;; Let-expressions

(test 21 (equal? (value-of (let-exp 'alpha (const-exp 12) (var-exp 'alpha))
                           (empty-env))
                 (num-val 12)))
(test 22 (equal? (value-of-program
                   (a-program (let-exp 'alpha
                                       (const-exp 12)
                                       (var-exp 'alpha))))
                 (num-val 12)))
(test 23 (equal? (run "let alpha = 12 in alpha") (num-val 12)))

;;; Proc-expressions

(test 24 (equal? (value-of (proc-exp '(pi) '() (const-exp 14)) (empty-env))
                 (proc-val (a-proc '(pi) 1 (const-exp 14) (empty-env)))))
(test 25 (equal? (value-of-program
                   (a-program (proc-exp '(rho) '() (var-exp 'rho))))
                   (proc-val (a-proc '(rho) 1 (var-exp 'rho) (init-env)))))
(test 26 (equal? (run "proc (sigma) 15")
                 (proc-val (a-proc '(sigma) 1 (const-exp 15) (init-env)))))

;;; Call-expressions

(test 27 (equal? (value-of
                   (call-exp (proc-exp '(tau) '() (diff-exp (var-exp 'tau)
                                                            (const-exp 16)))
                             (list (const-exp 17)))
                   (empty-env))
                 (num-val 1)))
(test 28 (equal? (value-of-program
                   (a-program
                     (call-exp (proc-exp '(upsilon)
                                         '()
                                         (zero?-exp (var-exp 'upsilon)))
                               (list (var-exp 'x)))))
                 (bool-val #f)))
(test 29 (equal? (run "(proc (phi) -(0, phi) 18)")
                 (num-val -18)))

;;; Programs from the textbook

(test 30 (equal? (run "-(55, -(x, 11))") (num-val 56)))
(test 31 (equal? (run "-(-(x, 3), -(v, i))") (num-val 3)))
(test 32 (equal? (value-of
                   (parse-expression
                     (scanner
                      (make-character-source
                        "if zero?(-(x, 11)) then -(y, 2) else -(y, 4)")))
                   (extend-env 'x (num-val 33)
                               (extend-env 'y (num-val 22) (empty-env))))
                 (num-val 18)))
(test 33 (equal? (run "let x = 5 in -(x, 3)") (num-val 2)))
(test 34 (equal? (run "let z = 5
                       in let x = 3
                          in let y = -(x, 1)      % here x = 3
                             in let x = 4
                                in -(z, -(x, y))  % here x = 4")
                 (num-val 3)))
(test 35 (equal? (run "let x = 7
                       in let y = 2
                          in let y = let x = -(x, 1)
                                     in -(x, y)
                             in -(-(x, 8), y)")
                 (num-val -5)))
(test 36 (equal? (run "let f = proc (x) -(x, 11)
                       in (f (f 77))")
                 (num-val 55)))
(test 37 (equal? (run "(proc (f) (f (f 77))
                        proc (x) -(x, 11))")
                 (num-val 55)))
(test 38 (equal? (run "let free = 31
                       in let addfree = proc (augend) -(augend, -(0, free))
                          in let free = 53
                             in (addfree free)")
                 (num-val 84)))

;; The following expression should signal errors:

;; (run "-(zero?(0), 5)")
;; (run "-(5, zero?(0))")
;; (run "zero?(zero?(0))")
;; (run "if 0 then 1 else 2")
;; (run "(35 42)")
;; (run "(zero?(0) 42)")
;; (run "zero?(proc (x) x)")

;;; The procedure definitions are the work of Friedman and Wand,
;;; who published them on Mitchell Wand's Github site,
;;; as part of the repository https://github.com/mwand/eopl3,
;;; under the Creative Commons Attribution-Noncommercial 3.0 Unported License.

;;; The test cases are
;;; copyright (C) 2009, 2015 by John David Stone
```

```
;;; and are similarly released
;;; under the Creative Commons Attribution-Noncommercial 3.0 Unported license.
```

```
#lang eopl

;;; Expressed values and environments for DYNAMIC-PROC

;;; John David Stone
;;; Department of Computer Science
;;; Grinnell College
;;; stone@cs.grinnell.edu

;;; Fiona Byrne & Tyler Dewey
;;; Certified killing it
;;; 14 September 2015

;;; Tyler Dewey
;;; Modified for multi-argument procs
;;; with default values
;;; October 3, 2015

;;; created February 5, 2009
;;; last revised October 5, 2015

;;; This module defines a data type
;;; for expressed values of the PROC programming language,
;;; as described in section 3.2 of
;;; _Essentials of programming languages_, third edition
;;; (Cambridge, Massachusetts: The MIT Press, 2008; ISBN 978-0-262-06279-4),
;;; by Daniel P. Friedman and Mitchell Wand.
;;; It also defines simple environments,
;;; as described in section 2.2 of that book.
;;; The two datatypes are presented together
;;; because they are mutually recursive.

(require "../natural-numbers.scm")
(require "../list-of.scm")
(require "syntax-trees.scm")

; ======= Expressed values ==========================================

;;; An expressed value in PROC
;;; is either an exact integer, a Boolean,
;;; or a value of the proc (i.e., closure)
;;; data type defined below.

(define-datatype expval expval?
  (num-val (num exact-integer?))
  (bool-val (bool boolean?))
  (proc-val (proc proc?)))

;;; We supplement the data type interface
;;; with projection functions that recover the values
;;; stored in the respective fields of the variants.

;; expval->num : ExpVal -> Int
(define expval->num
  (lambda (ev)
    (cases expval ev
      (num-val (num) num)
      (bool-val (bool)
        (report-domain-error 'expval->num "boolean" ev))
      (proc-val (proc)
        (report-domain-error 'expval->num "procedure" ev)))))

;; report-domain-error : Symbol * String * ExpVal -> (aborts the computation)
(define report-domain-error
  (lambda (location bad-type bad-ev)
    (eopl:error location
                "undefined for expressed ~a value ~s~%"
                bad-type
                bad-ev)))

;; expval->bool : ExpVal -> Bool
(define expval->bool
  (lambda (ev)
    (cases expval ev
      (num-val (num)
        (report-domain-error 'expval->bool "numeric" ev))
      (bool-val (bool) bool)
      (proc-val (proc)
        (report-domain-error 'expval->bool "procedure" ev)))))

;; expval->bool : Expval -> Proc
(define expval->proc
  (lambda (ev)
    (cases expval ev
      (num-val (num)
        (report-domain-error 'expval->proc "numeric" ev))
      (bool-val (bool)
        (report-domain-error 'expval->proc "boolean" ev))
      (proc-val (proc) proc))))

; ======= Closures ==========================================================

;; The identifiers used in this data type definition
;; differ slightly from those used in Friedman and Wand's book,
;; to avoid conflicts with standard Scheme's built-in procedure? procedure
;; and the identifier? procedure built into some languages under Racket.

(define-datatype proc proc?
  (a-proc (parameters (list-of symbol?))
          (min-arity natural-number?)
          (body expression?)
          (saved-env environment?)))

; ======= Environments ======================================================

;;; An environment is either empty or extends another environment
;;; by adding one new variable,
;;; to which some denoted value is bound.
;;; In the PROC language
;;; that Friedman and Wand introduce
;;; in section 3.3 of _Essentials of programming languages_,
;;; denoted values and expressed values are the same,
;;; so we'll use values of the expval data type in this role.

(define-datatype environment environment?
  (empty-env)
  (extend-env
    (var symbol?)
    (val expval?)
    (saved environment?)))

(define extend-env*
  (lambda (vars vals saved)
    (if (>= (length vars) (length vals))
        (if (null? vals)
            saved
            (extend-env (car vars) (car vals)
                        (extend-env* (cdr vars) (cdr vals) saved)))
        (eopl:error 'extend-env*
                    "must have at least as many vars as vals~%"))))

;;; The apply-env procedure looks up a given variable
;;; in a given environment
;;; and returns the denoted value bound to it.
;;; It is an error to apply apply-env
;;; to a variable that is not bound in the given environment.

;; apply-env : Env * Sym -> ExpVal
```

```
(define apply-env
  (lambda (env sought)
    (let kernel ((remaining env))
      (cases environment remaining
        (empty-env ()
          (report-no-binding-found sought env))
        (extend-env (var val saved)
          (if (eqv? var sought)
              val
              (kernel saved)))))))

(define report-no-binding-found
  (lambda (sought env)
    (eopl:error 'apply-env
                "No binding for ~s was found in environment ~s.~%"
                sought
                env)))

;;; PROC programs are evaluated
;;; in an initial environment
;;; containing bindings for a few Roman numerals.
;;; The init-env procedure constructs and returns this environment.

;;; This code is taken
;;; from section 3.2 of _Essentials of programming languages_.

;; init-env : () -> Env
(define init-env
  (lambda ()
    (extend-env 'i (num-val 1)
      (extend-env 'v (num-val 5)
        (extend-env 'x (num-val 10) (empty-env))))))

(provide expval expval? num-val bool-val proc-val expval->num expval->bool
    expval->proc proc proc? a-proc environment environment? empty-env
    extend-env extend-env* apply-env init-env)

;;; The definition of the init-env procedure
;;; is due to Daniel P. Friedman (dfried@cs.indiana.edu)
;;; and Mitchell Wand (wand@ccs.neu.edu),
;;; who made it available as part of the Git repository
;;; at https://github.com/mwand/eopl3,
;;; under the Creative Commons Attribution-Noncommercial 3.0 Unported license
;;; (http://creativecommons.org/licenses/by-nc/3.0/).

;;; The remaining definitions are
;;; copyright (C) 2009, 2015 by John David Stone
;;; and are similarly released
;;; under the Creative Commons Attribution-Noncommercial 3.0 Unported license.
```

```
#lang eopl

;;; Syntax trees for DYNAMIC-PROC

;;; John David Stone
;;; Department of Computer Science
;;; Grinnell College
;;; stone@cs.grinnell.edu

;;; Fiona Byrne & Tyler Dewey
;;; Certified killing it
;;; 14 September 2015

;;; Tyler Dewey
;;; Modified for multi-argument procs
;;; with default values
;;; October 3, 2015

;;; created February 3, 2009
;;; last revised October 5, 2015

;;; This module defines a data type
;;; for abstract syntax trees of the PROC programming language,
;;; as described in section 3.2 of
;;; _Essentials of programming languages_, third edition
;;; (Cambridge, Massachusetts: The MIT Press, 2008; ISBN 978-0-262-06279-4),
;;; by Daniel P. Friedman and Mitchell Wand.

(require "../natural-numbers.scm")
(require "../list-of.scm")

;;; The grammar for PROC is as follows:

;;;       <program> ::= <expression>
;;;    <expression> ::= <numeral>
;;;                   | - ( <expression> , <expression> )
;;;                   | zero? ( <expression> )
;;;                   | if <expression> then <expression> else <expression>
;;;                   | <identifier>
;;;                   | let <identifier> = <expression> in <expression>
;;;                   | proc ([<identifier>{, <identifiers>}* {, (<identifier> <exp
ression>)}*]) <expression>
;;;                   | ( <expression> [<expression> {, <expression>}*])

;;; The data type definitions exactly reflect this grammar.

(define-datatype program program?
  (a-program (exp expression?)))

(define-datatype expression expression?
  (const-exp (datum exact-integer?))
  (diff-exp (minuend expression?)
            (subtrahend expression?))
  (zero?-exp (testee expression?))
  (if-exp (condition expression?)
          (consequent expression?)
          (alternative expression?))
  (var-exp (id symbol?))
  (let-exp (bound-var symbol?)
           (bound-value expression?)
           (body expression?))
  (proc-exp (parameters (list-of symbol?))
            (optional-parameters (list-of symbol-expression-pair?))
            (body expression?))
  (call-exp (operator expression?)
            (operands (list-of expression?))))

(define symbol-expression-pair?
 (lambda (something)
   (and (pair? something)
        (symbol? (car something))
        (expression? (cdr something)))))

(provide program program? a-program expression expression? const-exp diff-exp
         zero?-exp if-exp var-exp let-exp proc-exp call-exp symbol-expression-pair?
)

;;; copyright (C) 2009, 2015 John David Stone

;;; This program is free software.
;;; You may redistribute it and/or modify it
;;; under the terms of the GNU General Public License
;;; as published by the Free Software Foundation --
;;; either version 3 of the License,
;;; or (at your option) any later version.
;;; A copy of the GNU General Public License
;;; is available on the World Wide Web at
;;;
;;;                http://www.gnu.org/licenses/gpl.html

;;; This program is distributed
;;; in the hope that it will be useful,
;;; but WITHOUT ANY WARRANTY --
;;; without even the implied warranty
;;; of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
;;; See the GNU General Public License for more details.
```

```
#lang eopl

;;; A parser for the PROC language

;;; John David Stone
;;; Department of Computer Science
;;; Grinnell College
;;; stone@cs.grinnell.edu

;;; Tyler Dewey
;;; Modified for multi-argument procs
;;; with default values
;;; October 3, 2015

;;; created February 5, 2009
;;; last revised October 5, 2015

;;; This file provides a parser for the PROC language
;;; developed by Daniel P. Friedman and Mitchell Wand
;;; in section 3.2 of their book
;;; _Essentials of programming languages_ (third edition).

(require "tokens.scm")
(require "scanner.scm")
(require "syntax-trees.scm")
(require "../character-sources.scm")

;;; The acquire procedure recovers a token
;;; from a given source,
;;; signalling an error if none is available.

;; acquire-token : Token-source -> Token
(define acquire-token
  (lambda (token-source)
    (when (token-source 'at-end?)
      (report-unexpected-end-of-source-error))
    (token-source 'get)))

;;; Alias for acquire token to convey intent

;; acquire-token : Token-source -> Token
(define discard-token acquire-token)

;;; The peek procedure retrieves a token
;;; from a given source, signalling if none
;;; is found, but not removing the token from
;;; the token source

;; peek-token : Token-source -> Token
(define peek-token
  (lambda (token-source)
    (when (token-source 'at-end?)
      (report-unexpected-end-of-source-error))
    (token-source 'peek)))

(define report-unexpected-end-of-source-error
  (lambda ()
    (eopl:error 'acquire-token
                "The end of the input was encountered unexpectedly.")))

;;; The match-and-discard procedure
;;; gets a token from a given source
;;; and compares it with the token
;;; that the parser expects to find.
;;; If they don't match, an error is reported.

;; match-and-discard : Token-source * Token -> ()
(define match-and-discard
  (lambda (token-source expected)
    (let ((discard (acquire-token token-source)))
      (unless (equal? expected discard)
        (report-unexpected-token-error discard expected)))))

(define report-unexpected-token-error
  (lambda (found expected)
    (eopl:error 'match-and-discard
                "The token ~a does not match the expected token ~a.~%"
                found
                expected)))

;;; There is a separate parsing procedure
;;; for each kind of internal node.

;;; parse-program : Token-source -> Program
(define parse-program
  (lambda (token-source)
    (a-program (parse-expression token-source))))

;;; parse-expression : Token-source -> Expression
(define parse-expression
  (lambda (token-source)
    (parse-expression-core
     token-source
     (lambda ()
       (report-bad-initial-token-error "A close parenthesis")))))

;;; parse-expression-list : Token-source -> Expression
(define parse-expression-list
  (lambda (token-source)
    (let ((expression (parse-expression-core token-source
                                             (lambda () '()))))
      (if (null? expression)
          '()
          (cons expression (parse-expression-list token-source))))))

;;; parse-expression-core : Token-source * Function -> Expression
(define parse-expression-core
  (lambda (token-source close-parenthesis-behavior)

    ;; Get a token and determine which of the analyses of expressions
    ;; should be used.

    (let ((current (acquire-token token-source)))
      (cases token current
        (numeral-token (value)
          (const-exp value))
        (minus-sign ()
          (parse-diff-exp token-source))
        (open-parenthesis ()
          (parse-call-exp token-source))
        (comma ()
          (report-bad-initial-token-error "A comma"))
        (close-parenthesis ()
          (close-parenthesis-behavior))
        (zero?-token ()
          (parse-zero?-exp token-source))
        (if-token ()
          (parse-if-exp token-source))
        (then-token ()
          (report-bad-initial-token-error "The keyword then"))
        (else-token ()
          (report-bad-initial-token-error "The keyword else"))
        (identifier-token (id)
          (var-exp id))
        (let-token ()
          (parse-let-exp token-source))
        (equals-sign ()
          (report-bad-initial-token-error "An equals sign"))
```

```
        (in-token ()
          (report-bad-initial-token-error "The keyword in"))
        (proc-token ()
          (parse-proc-exp token-source))))))

;; report-bad-initial-token-error : String -> ()
(define report-bad-initial-token-error
  (lambda (bad-token-string)
    (eopl:error 'parse-expression
                "~a may not occur at the beginning of an expression.~%"
                bad-token-string)))

;; parse-diff-exp : Token-source -> DiffExp
(define parse-diff-exp
  (lambda (token-source)
    (match-and-discard token-source (open-parenthesis))
    (let ((minuend (parse-expression token-source)))
      (match-and-discard token-source (comma))
      (let ((subtrahend (parse-expression token-source)))
        (match-and-discard token-source (close-parenthesis))
        (diff-exp minuend subtrahend)))))

;; parse-call-exp : Token-source -> CallExp
(define parse-call-exp
  (lambda (token-source)
    (let* ((operator (parse-expression token-source))
           (operands (parse-expression-list token-source)))
      (call-exp operator operands))))

;; parse-zero?-exp : Token-source -> Zero?Exp
(define parse-zero?-exp
  (lambda (token-source)
    (match-and-discard token-source (open-parenthesis))
    (let ((testee (parse-expression token-source)))
      (match-and-discard token-source (close-parenthesis))
      (zero?-exp testee))))

;; parse-if-exp : Token-source -> IfExp
(define parse-if-exp
  (lambda (token-source)
    (let ((condition (parse-expression token-source)))
      (match-and-discard token-source (then-token))
      (let ((consequent (parse-expression token-source)))
        (match-and-discard token-source (else-token))
        (let ((alternative (parse-expression token-source)))
          (if-exp condition consequent alternative))))))

;; parse-let-exp : Token-source -> LetExp
(define parse-let-exp
  (lambda (token-source)
    (let ((bound-var (acquire-identifier token-source)))
      (match-and-discard token-source (equals-sign))
      (let ((bound-value (parse-expression token-source)))
        (match-and-discard token-source (in-token))
        (let ((body (parse-expression token-source)))
          (let-exp bound-var bound-value body))))))

;; acquire-identifier : Token-source -> Sym
(define acquire-identifier
  (lambda (token-source)
    (let ((candidate (acquire-token token-source)))
      (cases token candidate
        (numeral-token (num)
          (report-acquire-identifier-error "A numeral"))
        (minus-sign ()
          (report-acquire-identifier-error "A minus sign"))
        (open-parenthesis ()
          (report-acquire-identifier-error "An open parenthesis"))
        (comma ()
          (report-acquire-identifier-error "A comma"))
        (close-parenthesis ()
          (report-acquire-identifier-error "A close parenthesis"))
        (zero?-token ()
          (report-acquire-identifier-error "The keyword zero?"))
        (if-token ()
          (report-acquire-identifier-error "The keyword if"))
        (then-token ()
          (report-acquire-identifier-error "The keyword then"))
        (else-token ()
          (report-acquire-identifier-error "The keyword else"))
        (identifier-token (id) id)
        (let-token ()
          (report-acquire-identifier-error "The keyword let"))
        (equals-sign ()
          (report-acquire-identifier-error "An equals sign"))
        (in-token ()
          (report-acquire-identifier-error "The keyword in"))
        (proc-token ()
          (report-acquire-identifier-error "The keyword proc"))))))

;; report-acquire-identifier-error : String -> ()
(define report-acquire-identifier-error
  (lambda (bad-token-string)
    (eopl:error 'acquire-identifier
                "~a was found in place of an identifier.~%"
                bad-token-string)))

;; parse-proc-exp : Token-source -> ProcExp
(define parse-proc-exp
  (lambda (token-source)
    (let* ((parameters (acquire-parameters token-source))
           (optional-parameters (acquire-optional-parameters token-source))
           (body (parse-expression token-source)))
      (proc-exp parameters optional-parameters body))))

;; acquire-parameters : Token-source -> Scheme-list of identifiers
(define acquire-parameters
 (lambda (token-source)
   (match-and-discard token-source (open-parenthesis))
   ;; peek at next token, because acquire-optional-parameters will remove close par
en
   (let ((param-candidate (peek-token token-source)))
     (cases token param-candidate

       ;; found empty parameter list
       (close-parenthesis () '()) ;; end of params
       (open-parenthesis () '())  ;; beginning of optional params

       (identifier-token (id)

         ;; remove "peeked" identifier
         (match-and-discard token-source (identifier-token id))

         ;; get remaining standard params
         (let acquire-remaining-parameters ((so-far (list id)))
           (let ((separator-candidate (peek-token token-source)))
             (cases token separator-candidate
               (close-parenthesis () (reverse so-far)) ;; parameter list is constru
cted in reverse &
                                                       ;; final order is important
for evaluation
               (comma ()
                 (discard-token token-source)
                 (cases token (peek-token token-source)
                   (open-parenthesis () (reverse so-far)) ;; open-parenthesis indic
ates start of
                                                          ;; optional param list
                   (else
```

```
                    (acquire-remaining-parameters (cons (acquire-identifier token-s
ource) so-far)))))
                (else
                 (report-acquire-parameter-error separator-candidate))))))
          (else
            (report-acquire-parameter-error param-candidate))))))

;; acquire-optional-parameters : Token-source -> Scheme-list of identifier & expres
sion pairs
(define acquire-optional-parameters
  (lambda (token-source)
    (let ((candidate (peek-token token-source)))
      (cases token candidate
        (close-parenthesis ()
          (discard-token token-source) ;; remove close-paren on top
          '())
        (open-parenthesis ()
          (let ((param-pair (acquire-optional-param-pair token-source)))

            ;; get remaining optional params
            (let acquire-remaining-optional-params ((so-far (list param-pair)))

              (let ((separator-candidate (peek-token token-source)))
                (cases token separator-candidate
                  (close-parenthesis ()
                     (discard-token token-source) ;; remove close-paren on top
                     (reverse so-far)) ;; list is constructed in reverse & final
                                       ;; order is important for evaluation
                  (comma ()
                    (discard-token token-source) ;; remove comma on top
                    (acquire-remaining-optional-params
                     (cons (acquire-optional-param-pair token-source) so-far)))
                  (else
                   (report-acquire-parameter-error separator-candidate)))))))
        (else
         (report-acquire-parameter-error candidate))))))

(define acquire-optional-param-pair
  (lambda (token-source)
    (match-and-discard token-source (open-parenthesis))
    (let ((param-pair (cons (acquire-identifier token-source)
                            (parse-expression token-source))))
      (match-and-discard token-source (close-parenthesis))
      param-pair)))

(define report-acquire-parameter-error
  (lambda (found-token)
    (eopl:error 'acquire-parameters
                "~a was found in malformed a parameter list.~%"
                found-token)))

;;; The scan&parse procedure takes a string or an input port as its
;;; argument and returns a syntax tree for the program that the source
;;; provides.

;; scan&parse : SchemeVal -> Program
(define scan&parse
  (lambda (given)
    (let* ((token-source (scanner (make-character-source given)))
           (syntax-tree (parse-program token-source)))
      (if (token-source 'at-end?)
          syntax-tree
          (report-leftover-tokens-error)))))

(define report-leftover-tokens-error
  (lambda ()
    (eopl:error 'scan&parse
                "There were extra, unusable tokens at the end of the program.")))
```

```
(provide scan&parse parse-program parse-expression)

;; ;;; Tests

 (require "../test.scm")

;; ;; Parsing numerals.

;; (test 0 (equal? (parse-expression (scanner (make-character-source "0")))
;;                 (const-exp 0)))
;; (test 1 (equal? (parse-expression
;;                   (scanner (make-character-source "8128")))
;;                 (const-exp 8128)))
;; (test 2 (equal? (parse-expression
;;                   (scanner
;;                     (make-character-source
;;                       "99999999939999999999999999999999999999999999999999")))
;;                 (const-exp 99999999939999999999999999999999999999999999999999)))
;; (test 3 (equal? (parse-program (scanner (make-character-source "0")))
;;                 (a-program (const-exp 0))))
;; (test 4 (equal? (parse-program
;;                   (scanner (make-character-source "8128")))
;;                 (a-program (const-exp 8128))))
;; (test 5 (equal? (parse-program
;;                   (scanner
;;                     (make-character-source
;;                       "99999999939999999999999999999999999999999999999999")))
;;                 (a-program
;;                   (const-exp
;;                     99999999939999999999999999999999999999999999999999))))

;; ;; Parsing simple identifiers.

;; (test 6 (equal? (parse-expression (scanner (make-character-source "x")))
;;                 (var-exp 'x)))
;; (test 7 (equal? (parse-expression
;;                   (scanner (make-character-source "foo?2")))
;;                 (var-exp 'foo?2)))
;; (test 8 (equal? (parse-program (scanner (make-character-source "x")))
;;                 (a-program (var-exp 'x))))
;; (test 9 (equal? (parse-program (scanner (make-character-source "foo?2")))
;;                 (a-program (var-exp 'foo?2))))

;; ;; Parsing simple diff-expressions.

;; (test 10 (equal? (parse-diff-exp
;;                    (scanner (make-character-source "(1, 2)")))
;;                  (diff-exp (const-exp 1) (const-exp 2))))
;; (test 11 (equal? (parse-expression
;;                    (scanner (make-character-source "-(alpha, beta)")))
;;                  (diff-exp (var-exp 'alpha) (var-exp 'beta))))
;; (test 12 (equal? (parse-program
;;                    (scanner (make-character-source "-(gamma, 3)")))
;;                  (a-program (diff-exp (var-exp 'gamma) (const-exp 3)))))

;; ;; Parsing simple zero?-expressions.

;; (test 13 (equal? (parse-zero?-exp
;;                    (scanner (make-character-source "(4)")))
;;                  (zero?-exp (const-exp 4))))
;; (test 14 (equal? (parse-expression
;;                    (scanner (make-character-source "zero?(delta)")))
;;                  (zero?-exp (var-exp 'delta))))
;; (test 15 (equal? (parse-program
;;                    (scanner (make-character-source "zero?(5)")))
;;                  (a-program (zero?-exp (const-exp 5)))))

;; ;; Parsing simple if-expressions.
```

```
;; (test 16 (equal? (parse-if-exp
;;                    (scanner (make-character-source
;;                              "zero?(epsilon) then 6 else zeta")))
;;                  (if-exp (zero?-exp (var-exp 'epsilon))
;;                          (const-exp 6)
;;                          (var-exp 'zeta))))
;; (test 17 (equal? (parse-expression
;;                    (scanner (make-character-source
;;                              "if zero?(7) then eta else 8")))
;;                  (if-exp (zero?-exp (const-exp 7))
;;                          (var-exp 'eta)
;;                          (const-exp 8))))
;; (test 18 (equal? (parse-program
;;                    (scanner (make-character-source
;;                              "if zero?(theta) then 9 else iota")))
;;                  (a-program (if-exp (zero?-exp (var-exp 'theta))
;;                                     (const-exp 9)
;;                                     (var-exp 'iota)))))

;; ;; Parsing simple let-expressions.

;; (test 19 (equal? (parse-let-exp
;;                    (scanner (make-character-source "kappa = 10 in kappa")))
;;                  (let-exp 'kappa (const-exp 10) (var-exp 'kappa))))
;; (test 20 (equal? (parse-expression
;;                    (scanner (make-character-source "let mu = nu in 11")))
;;                  (let-exp 'mu (var-exp 'nu) (const-exp 11))))
;; (test 21 (equal? (parse-program
;;                    (scanner (make-character-source
;;                              "let omicron = -(pi, 12) in -(omicron, 13)")))
;;                  (a-program (let-exp 'omicron
;;                                      (diff-exp (var-exp 'pi)
;;                                                (const-exp 12))
;;                                      (diff-exp (var-exp 'omicron)
;;                                                (const-exp 13))))))

;; ;; Parsing simple proc-expressions.

(test 22 (equal? (parse-proc-exp
                  (scanner (make-character-source "(pi) 14")))
                 (proc-exp '(pi) '() (const-exp 14))))
(test 23 (equal? (parse-expression
                  (scanner (make-character-source "proc (rho) 15")))
                 (proc-exp '(rho) '() (const-exp 15))))
(test 24 (equal? (parse-program
                  (scanner (make-character-source "proc (sigma) 16")))
                 (a-program (proc-exp '(sigma) '() (const-exp 16)))))

;; ;; Parsing simple call-expressions.

(test 25 (equal? (parse-call-exp
                  (scanner (make-character-source "tau 17")))
                 (call-exp (var-exp 'tau) (list (const-exp 17)))))
(test 26 (equal? (parse-expression
                  (scanner (make-character-source "(upsilon 18)")))
                 (call-exp (var-exp 'upsilon)
                           (list (const-exp 18)))))
(test 27 (equal? (parse-program
                  (scanner (make-character-source "(phi 19)")))
                 (a-program (call-exp (var-exp 'phi)
                                      (list (const-exp 19))))))

;; ;; A more complex example.

 (test 28 (equal? (parse-expression
                   (scanner (make-character-source
                             "if zero?(14)
                              then let sigma = tau
                                   in -(sigma, 15)
                              else upsilon")))
                  (if-exp (zero?-exp (const-exp 14))
                          (let-exp 'sigma
                                   (var-exp 'tau)
                                   (diff-exp (var-exp 'sigma)
                                             (const-exp 15)))
                          (var-exp 'upsilon))))
 (test 29 (equal? (parse-program
                   (scanner (make-character-source
                             "if zero?(phi)
                              then let chi = 16
                                   in -(phi, chi)
                              else 17")))
                  (a-program (if-exp (zero?-exp (var-exp 'phi))
                                     (let-exp 'chi
                                              (const-exp 16)
                                              (diff-exp (var-exp 'phi)
                                                        (var-exp 'chi)))
                                     (const-exp 17)))))

;; ;; Testing scan&parse.

;; (test 30 (equal? (scan&parse "18")
;;                  (a-program (const-exp 18))))
;; (test 31 (equal? (scan&parse "omega")
;;                  (a-program (var-exp 'omega))))
;; (test 32 (equal? (scan&parse "-(19, aleph)")
;;                  (a-program (diff-exp (const-exp 19) (var-exp 'aleph)))))
;; (test 33 (equal? (scan&parse "zero?(20)")
;;                  (a-program (zero?-exp (const-exp 20)))))
;; (test 34 (equal? (scan&parse "if zero?(bet) then 21 else gimel")
;;                  (a-program (if-exp (zero?-exp (var-exp 'bet))
;;                                     (const-exp 21)
;;                                     (var-exp 'gimel)))))
;; (test 35 (equal? (scan&parse "let dalet = 22 in he")
;;                  (a-program (let-exp 'dalet
;;                                      (const-exp 22)
;;                                      (var-exp 'he)))))
;; (test 36 (equal? (scan&parse "proc (vav) 23")
;;                  (a-program (proc-exp 'vav (const-exp 23)))))
;; (test 37 (equal? (scan&parse "(zayin 24)")
;;                  (a-program (call-exp (var-exp 'zayin) (const-exp 24)))))


;; ;; Some sample programs from the textbook.

;; (test 38 (equal? (scan&parse "-(55, -(x, 11))")
;;                  (a-program (diff-exp (const-exp 55)
;;                                       (diff-exp (var-exp 'x)
;;                                                 (const-exp 11))))))
;; (test 39 (equal? (scan&parse "let x = 5 in -(x, 3)")
;;                  (a-program (let-exp 'x
;;                                      (const-exp 5)
;;                                      (diff-exp (var-exp 'x)
;;                                                (const-exp 3))))))
;; (test 40 (equal? (scan&parse "let z = 5
;;                               in let x = 3
;;                                  in let y = -(x, 1)      % here x = 3
;;                                     in let x = 4
;;                                        in -(z, -(x, y))  % here x = 4")
;;                  (a-program
;;                    (let-exp
;;                      'z
;;                      (const-exp 5)
;;                      (let-exp 'x
;;                               (const-exp 3)
;;                               (let-exp 'y
;;                                        (diff-exp
;;                                          (var-exp 'x)
```

```
;;                                      (const-exp 1))
;;                                    (let-exp 'x
;;                                            (const-exp 4)
;;                                            (diff-exp
;;                                              (var-exp 'z)
;;                                              (diff-exp
;;                                                (var-exp 'x)
;;                                                (var-exp 'y)))))))))))
;; (test 41 (equal? (scan&parse "let z = 7
;;                               in let y = 2
;;                                   in let y = let x = -(x, 1)
;;                                              in -(x, y)
;;                                      in -(-(x, 8), y)")
;;                  (a-program
;;                    (let-exp 'z
;;                             (const-exp 7)
;;                             (let-exp 'y
;;                                      (const-exp 2)
;;                                      (let-exp 'y
;;                                               (let-exp 'x
;;                                                        (diff-exp
;;                                                          (var-exp 'x)
;;                                                          (const-exp 1))
;;                                                        (diff-exp
;;                                                          (var-exp 'x)
;;                                                          (var-exp 'y)))
;;                                               (diff-exp
;;                                                 (diff-exp
;;                                                   (var-exp 'x)
;;                                                   (const-exp 8))
;;                                                 (var-exp 'y))))))))
;;
;; (test 42 (equal? (scan&parse "let f = proc (x) -(x, 11)
;;                               in (f (f 77))")
;;                  (a-program
;;                    (let-exp 'f
;;                             (proc-exp 'x (diff-exp (var-exp 'x)
;;                                                    (const-exp 11)))
;;                             (call-exp (var-exp 'f)
;;                                       (call-exp (var-exp 'f)
;;                                                 (const-exp 77)))))))
;;
;; (test 43 (equal? (scan&parse "(proc (f) (f (f 77))
;;                                proc (x) -(x, 11))")
;;                  (a-program
;;                    (call-exp
;;                      (proc-exp 'f (call-exp (var-exp 'f)
;;                                             (call-exp (var-exp 'f)
;;                                                       (const-exp 77))))
;;                      (proc-exp 'x (diff-exp (var-exp 'x)
;;                                             (const-exp 11)))))))
;;
;; (test 44 (equal? (scan&parse "let x = 200
;;                               in let f = proc (z) -(z, x)
;;                                  in let x = 100
;;                                     in let g = proc (z) -(z, x)
;;                                        in -((f 1), (g 1))")
;;                  (a-program
;;                    (let-exp
;;                      'x
;;                      (const-exp 200)
;;                      (let-exp
;;                        'f
;;                        (proc-exp 'z (diff-exp (var-exp 'z)
;;                                               (var-exp 'x)))
;;                        (let-exp 'x
;;                                 (const-exp 100)
;;                                 (let-exp
;;                                   'g
;;                                   (proc-exp 'z (diff-exp (var-exp 'z)
;;                                                          (var-exp 'x)))
;;                                   (diff-exp (call-exp (var-exp 'f)
;;                                                       (const-exp 1))
;;                                             (call-exp (var-exp 'g)
;;                                                       (const-exp
;;                                                        1)))))))))))
;;
;; (test 45 (equal? (scan&parse "let makemult = proc (maker)
;;                                              proc (x)
;;                                               if zero?(x)
;;                                               then 0
;;                                               else -(((maker maker) -(x, 1)),
;;                                                       4)
;;                               in let times4 = proc (x)
;;                                               ((makemult makemult) x)
;;                                  in (times4 3)")
;;                  (a-program
;;                    (let-exp
;;                      'makemult
;;                      (proc-exp
;;                        'maker
;;                        (proc-exp
;;                          'x
;;                          (if-exp
;;                            (zero?-exp (var-exp 'x))
;;                            (const-exp 0)
;;                            (diff-exp
;;                              (call-exp
;;                                (call-exp (var-exp 'maker) (var-exp 'maker))
;;                                (diff-exp (var-exp 'x) (const-exp 1)))
;;                              (const-exp 4)))))
;;                      (let-exp 'times4
;;                               (proc-exp 'x
;;                                         (call-exp
;;                                           (call-exp (var-exp 'makemult)
;;                                                     (var-exp 'makemult))
;;                                           (var-exp 'x)))
;;                               (call-exp (var-exp 'times4)
;;                                         (const-exp 3)))))))

;; ;; Parsing multi-argument proc-expressions.

 (test 46 (equal? (parse-proc-exp
                    (scanner (make-character-source "() 14")))
                  (proc-exp '() '() (const-exp 14))))

 (test 47 (equal? (parse-proc-exp
                    (scanner (make-character-source "(pi, rho) 14")))
                  (proc-exp '(pi rho) '() (const-exp 14))))

;; ;; Parsing multi-argument call-expressions.

 (test 48 (equal? (parse-call-exp
                    (scanner (make-character-source "tau)")))
                  (call-exp (var-exp 'tau) (list))))

 (test 49 (equal? (parse-call-exp
                    (scanner (make-character-source "tau 17 18)")))
                  (call-exp (var-exp 'tau) (list (const-exp 17)
                                                 (const-exp 18)))))

;; ;; Parsing optional-arg proc-expressions.

 (test 50 (equal? (parse-proc-exp
                    (scanner (make-character-source "(pi, (rho 1)) 14")))
                  (proc-exp '(pi) (list (cons 'rho (const-exp 1))) (const-exp 14)))
)
```

```
 (test 51 (equal? (parse-proc-exp
                   (scanner (make-character-source "((pi 1), (rho x)) 14")))
                  (proc-exp '()
                            (list (cons 'pi (const-exp 1))
                                  (cons 'rho (var-exp 'x)))
                            (const-exp 14))))

;; ;; The following expressions should raise errors when evaluated.

;; ;; (parse-diff-exp (scanner (make-character-source "")))
;; ;; (parse-diff-exp (scanner (make-character-source "vav")))
;; ;; (parse-diff-exp (scanner (make-character-source "(23")))
;; ;; (parse-diff-exp (scanner (make-character-source "(zayin)")))
;; ;; (parse-diff-exp (scanner (make-character-source "(24, ")))
;; ;; (parse-diff-exp (scanner (make-character-source "(chet, 25")))
;; ;; (parse-diff-exp (scanner (make-character-source "(tet, 26, yodh)")))
;; ;; (parse-zero?-exp (scanner (make-character-source "")))
;; ;; (parse-zero?-exp (scanner (make-character-source "27")))
;; ;; (parse-zero?-exp (scanner (make-character-source "(khaph")))
;; ;; (parse-zero?-exp (scanner (make-character-source "(29, lamed)")))
;; ;; (parse-if-exp (scanner (make-character-source "30")))
;; ;; (parse-if-exp (scanner (make-character-source "mem else 31")))
;; ;; (parse-if-exp (scanner (make-character-source "nun 32")))
;; ;; (parse-if-exp (scanner (make-character-source "samed then")))
;; ;; (parse-if-exp (scanner (make-character-source "33 then ayin")))
;; ;; (parse-if-exp (scanner (make-character-source "34 then pei 35")))
;; ;; (parse-if-exp (scanner (make-character-source "tsadi then 36 else")))
;; ;; (parse-let-exp (scanner (make-character-source "")))
;; ;; (parse-let-exp (scanner (make-character-source "39 = quph in 40")))
;; ;; (parse-let-exp (scanner (make-character-source "resh")))
;; ;; (parse-let-exp (scanner (make-character-source "shin 41")))
;; ;; (parse-let-exp (scanner (make-character-source "tav =")))
;; ;; (parse-let-exp (scanner (make-character-source "alif = 42")))
;; ;; (parse-let-exp (scanner (make-character-source "ba = 43 (")))
;; ;; (parse-let-exp (scanner (make-character-source "ta = 44 in")))
;; ;; (parse-let-exp (scanner (make-character-source "gim = 45 in let")))
;; ;; (parse-proc-exp (scanner (make-character-source "proc")))
;; ;; (parse-proc-exp (scanner (make-character-source "(46")))
;; ;; (parse-proc-exp (scanner (make-character-source "(ha,")))
;; ;; (parse-call-exp (scanner (make-character-source "dal)")))
;; ;; (parse-call-exp (scanner (make-character-source "ra 48,")))

;;; copyright (C) 2009, 2015 John David Stone

;;; This program is free software.
;;; You may redistribute it and/or modify it
;;; under the terms of the GNU General Public License
;;; as published by the Free Software Foundation --
;;; either version 3 of the License,
;;; or (at your option) any later version.
;;; A copy of the GNU General Public License
;;; is available on the World Wide Web at
;;;
;;;                http://www.gnu.org/licenses/gpl.html

;;; This program is distributed
;;; in the hope that it will be useful,
;;; but WITHOUT ANY WARRANTY --
;;; without even the implied warranty
;;; of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
;;; See the GNU General Public License for more details.
```