

```
; Syntax-rules functions
; Copyright 2005 Brian Alliet

; Oleg's syntax-rules version of symbol? from comp.lang.scheme
; Copyright 2003 Oleg Kiselyov
; Message ID: 7eb8ac3e.0309301103.5200afbf@posting.google.com
(define-syntax symbol??
  (syntax-rules ()
    ((symbol?? (x . y) kt kf) kf)           ; It's a pair, not a symbol
    ((symbol?? #(x ...) kt kf) kf)         ; It's a vector, not a symbol
    ((symbol?? maybe-symbol kt kf)
      (let-syntax
        ((test
          (syntax-rules ()
            ((test maybe-symbol t f) t)
            ((test x t f) f))))
        (test abracadabra kt kf))))))

; syntax-error is used to throw an error from a syntax-rule
; A call to syntax-error with args will not match any rules below and most Scheme implementations
; will display the call and the arguments (which should contain some helpful error message)
(define syntax-error
  (syntax-rules ()
    ((syntax-error) (syntax-error "call me with args!"))))
```

```

; A simple syntax-rules based pattern matching facility for Scheme based somewhat on Haskell
; Copyright 2005 Brian Alliet

; The expr argument to match-singe WILL be evaluated multiple times
; it shouldn't have side effects
(define-syntax match-single
  (syntax-rules (cons list quote @ _)
    ; Wildcard pattern (_ in haskell)
    ((match-single _ expr success fail) success)
    ; AS patterns (x@pat in haskell)
    ((match-single (var @ pat) expr success fail)
     (match-single pat expr (let ((var expr)) success) fail))
    ; Cons cell pattern (x:xs or (x,y) in haskell)
    ((match-single (cons p1 p2) expr success fail)
     (if (pair? expr)
         (let
            ((x (car expr)) (xs (cdr expr)))
            (match-single p1 x
                          (match-single p2 xs success fail)
                          fail))
         fail))
    ; Zero length list ([] in haskell)
    ((match-single (quote ()) expr success fail) (if (null? expr) success fail))
    ; Fixed length lists ([x1,x2,...xn] in haskell)
    ((match-single (list) expr success fail) (match-single '() expr success fail))
    ((match-single (list x y ...) expr success fail)
     (match-single (cons x (list y ...)) expr success fail))
    ; True and false constants, we know we can test these with eq? (True and False in haskell)
    ((match-single (quote #t) expr success fail) (if (eq? #t expr) success fail))
    ((match-single (quote #f) expr success fail) (if (eq? #f expr) success fail))
    ; Other quoted items are compared for equality with equal?
    ((match-single (quote pat) expr success fail) (if (equal? 'pat expr) success fail))
    ; Anything else MUST be a variable pattern
    ((match-single pat expr success fail)
     (symbol?? pat
      ; If it is a variable bind the value to it and succeed
      (let ((pat expr)) success)
      ; If not, it is something weird we can't handle, blow up
      (syntax-error "invalid pattern" 'pat)
     ))
  ))

; expr WILL be evaluated multiple times
(define-syntax match-helper
  (syntax-rules ()
    ((match-helper expr) (error "pattern match failure"))
    ((match-helper expr (pat body) r ...)
     (let
        ; Is there a better way to do this?
        ; If the match fails we need to get back up to this scope
        ((fail (lambda () (match-helper expr r ...))))
        (match-single pat expr body (fail))))
  ))

(define-syntax match
  (syntax-rules ()
    ((match expr clauses ...) (let ((e expr)) (match-helper e clauses ...)))
  ))

(define-syntax lambda-match
  (syntax-rules ()
    ((lambda-match clauses ...) (lambda (x) (match x clauses ...)))
  ))

(define-syntax define-match
  (syntax-rules ()
    ((define-match f clauses ...) (define f (lambda-match clauses ...)))
  ))

```

```

; Misc handy library functions (many based on Haskell and ML)
; Copyright 2005 Brian Alliet

; Undefined
(define (undefined) (error "undefined"))

; error
(define (error msg) ((string-append "Error: " msg)))

; Take an n-argument function and m initial arguments, returns a procedure that
; applies the m initial arguments followed by any remaining arguments
; This is a little different than Haskell's curry but Haskell's curry doesn't make much
; sense in Scheme
(define (curry f . initial)
  (λ rest
    (apply f (append initial rest))))

; Left and right section operators (as in SML or Haskell)
(define (left-section op arg) (λ (x) (op arg x)))
(define (right-section op arg) (λ (x) (op x arg)))

; fromTo generates a list containing each number in the range [beg,end]
; FEATURE: Use an accumulator
(define (from-to beg end)
  (cond
    ((> beg end) '())
    (else (cons beg (from-to (+ beg 1) end))))
  ))

; filter keeps only the elements of a list that match a predicate
; FEATURE: filter: optimize
(define (filter p xs)
  (match xs
    ('() '())
    ((cons x xs) (let ((rest (filter p xs))) (if (p x) (cons x rest) rest))))
  ))

; concat concatenates a list of lists
(define (concat xs) (apply append xs))

; haskell names for car and cdr
(define head car)
(define tail cdr)
(define fst car)
(define snd cdr)

; find the last element of a list
; FEATURE: last: optimize
(define-match last
  ('() (error "last: empty list"))
  ((list x) x)
  ((cons _ xs) (last xs)))

; FEATURE: init: optimize
(define-match init
  ('() (error "init: empty list"))
  ((list x) '())
  ((cons x xs) (cons x (init xs))))

; FEATURE: foldl: optimize
(define (foldl f z xs)
  (match xs
    ('() z)
    ((cons x xs) (foldl f (f z x) xs))))

; FEATURE: foldr: optimize
(define (foldr f z xs)
  (match xs
    ('() z)
    ((cons x xs) (f x (foldr f z xs)))))

; FEATURE: take: optimize
(define (take n xs)
  (if (eqv? n 0)
      '()

```

```

      (match xs
        ('() '())
        ((cons x xs) (cons x (take (- n 1) xs)))))

(define (drop n xs) (list-tail xs n))

(define (intersperse sep xs)
  (match xs
    ('() '())
    ((list _) '())
    ((cons x xs) (cons x (cons sep (intersperse sep xs)))))

; concatMap concatenates the results of applying map to a list
(define (concat-map f xs)
  (concat (map f xs)))

(define (show x) (apply string-append (show_ x '())))

(define (show_ x rest)
  (cond
    ((null? x) (cons "" rest))
    ((pair? x) (cons "(" (tail (foldr (λ (x r) (cons " " (show_ x r))) (cons ")" rest) x))))
    ((string? x) (cons "\"" (cons x (cons "\"" rest))))
    ((number? x) (cons (number->string x) rest))
    ((symbol? x) (cons (symbol->string x) rest))
    ((char? x) (cons (list->string (list x)) rest))
    ('#t (cons "#t" rest))
    ('#f (cons "#f" rest))
    (else (error "can't show that"))
  ))

(define (lookup a xs)
  (match xs
    ('() #f)
    ((cons (cons k v) xs) (if (equal? k a) v (lookup a xs)))))

(define (zipWith f xs ys) (map f xs ys))
(define zip (curry zipWith cons))

(define (max x y) (if (> x y) x y))
(define (min x y) (if (< x y) x y))

(define (all f xs)
  (match xs
    ('() #t)
    ((cons x xs) (and (f x) (all f xs)))))

(define (compose f g) (λ x (f (apply g x))))

```

```
; Misc handy library functions (many based on Haskell and ML)
; Copyright 2005 Brian Alliet

; delete the first element from a list which is equal? to x
; FEATURE: Use an accumulator
(define (delete y xs)
  (match xs
    ('() '())
    ((cons x xs) (if (equal? x y) xs (cons x (delete y xs))))
  ))

; replace replaces the first element of a list equal? to old with new
; FEATURE: Use an accumulator
(define (replace old new xs)
  (match xs
    ('() '())
    ((cons x xs) (if (equal? old x) (cons new xs) (cons x (replace old new xs))))
  ))
```

```
; Haskell style list comprehension
; Copyright 2005 Brian Alliet

; [expr/guard, pat <- gen] => (list-comp expr guard (pat <- gen))
; FEATURE: Optimize this, right now it generates too many 1 element lists
(define-syntax list-comp
  (syntax-rules (<-)
    ((list-comp e) (cons e '()))
    ((list-comp e (p <- l) q ...)
      (concat-map
        (lambda-match
          (p (list-comp e q ...))
          (_ '())
          ) l))
    ((list-comp e b q ...) (if b (list-comp e q ...) '()))
  ))
```

```

; Haskell style monads for scheme
; Copyright 2005 Brian Alliet

(define create-monad-dict vector)

; Primitive functions to deal with monad dictionaries
; These functions extract functions from the monad dictionary
(define (return-value dict) (vector-ref dict 0))
(define (bind-value dict) (vector-ref dict 1))
(define (fail-value dict) (vector-ref dict 2))
(define (mzero-value dict) (vector-ref dict 3))
(define (mplus-value dict) (vector-ref dict 4))

; And these call the above functions with arguments
(define (return dict x) ((return-value dict) x))
(define (bind dict m f) ((bind-value dict) m f))
(define (fail dict s) ((fail-value dict) s))
(define (mzero mzero-value))
(define (mplus dict a b) ((mplus-value dict) a b))

; Generic monad function defined in terms of the above interface
(define (bind_ d p q) (bind d p (lambda (x) q)))
(define (msum d: xs) (foldr (mplus-value d:) (mzero-value d:) xs))
(define (lift-m d: f) (lambda (x) (bind d: x (lambda (x2) (return d: (f x2))))))
(define (sequence d: xs)
  (let*
    ((b (bind-value d:))
     (r (return-value d:))
     (mcons (lambda (p q) (b p (lambda (y) (r (cons x y)))))))
    (foldr mcons (return d: '()) xs)))
(define (sequence_ d: xs) (foldr (curry bind_ d:) (return d: '()) xs))
(define (map-m d f xs) (sequence d (map f xs)))
(define (map-m_ d f xs) (sequence_ d (map f xs)))

(define-syntax do
  (syntax-rules (<- let)
    ((do d: (p <- e) rest ...) (bind d: e
      (lambda-match
        (p (do d: rest ...))
        (_ (fail d: "pattern match failure in do expression")))))
    ((do d: (let (var expr) ...) rest ...) (let ((var expr) ...) (do d: rest ...)))
    ((do d: e) e)
    ((do d: e rest ...) (bind d: e (lambda (ignore) (do d: rest ...))))
  ))

; Monad instances

; List Monad
(define (list:return x) (list x))
(define (list:bind x f) (concat-map f x))
(define (list:fail s) '())
(define (list:mzero '()))
(define (list:mplus append))
(define list: (create-monad-dict list:return list:bind list:fail list:mzero list:mplus))

; Maybe Monad (which isn't lazy, so doesn't work as well as in Haskell)
(define (maybe:return x) (cons 'just x))
(define (maybe:bind mx f)
  (match mx
    ((cons 'just x) (f x))
    ('nothing 'nothing)
    (_ (error "maybe:bind thats not a maybe"))))
(define (maybe:fail s) 'nothing)
(define (maybe:mzero 'nothing))
(define (maybe:mplus l r)
  (match l
    ((cons 'just _) l)
    ('nothing r)
    (_ (error "mplus: thats not a maybe"))))
(define maybe: (create-monad-dict maybe:return maybe:bind maybe:fail maybe:mzero maybe:mplus))

; Identity Monad
(define (id:return x) x)
(define (id:bind x f) (f x))
(define (id:fail s) (error s))
(define id: (create-monad-dict id:return id:bind id:fail))

```

```
; State Monad
(define (state:return x) (λ (state) (cons state x)))
(define (state:bind r f) (λ (state) (match (r state) ((cons state2 x) ((f x) state2)))))
(define (state:fail s) (error s))
(define state: (create-monad-dict state:return state:bind state:fail))

(define get-state (λ (state) (cons state state)))
(define (put-state x) (λ (state) (cons x '())))
(define (modify-state f) (state:bind get-state (λ (s) (put-state (f s)))))
```

```

; Haskell-like IO Monad
; Copyright 2005 Brian Alliet

(define (io:return x) (λ () x))
(define (io:bind io f) (λ () ((f (io))))))
(define (io:fail s) (λ () (error s)))
(define io: (create-monad-dict io:return io:bind io:fail))
(define (unsafe-perform-io io) (io))

(define (h-get-char handle) (λ() (read-char handle)))
(define (h-put-char handle c) (λ () (write-char c handle) '()))
(define (h-is-eof? handle) (λ () (eof-object? (peek-char handle))))
(define (h-put-str-list handle cs) (map-m_ io: (curry h-put-char handle) cs))
(define (h-get-contents-list handle) (do io:
  (b <- (h-is-eof? handle))
  (if b
    (io:return '())
    (do io:
      (c <- (h-get-char handle))
      (cs <- (h-get-contents-list handle))
      (io:return (cons c cs))))))

(define (h-put-str handle s) (h-put-str-list handle (string->list s)))
(define (h-put-str-ln handle s) (bind_ io: (h-put-str handle s) (h-put-char handle #\newline)))
(define (h-get-contents handle) ((lift-m io: list->string) (h-get-contents-list handle)))

(define (open-file name mode) (λ ()
  (match mode
    ('read-mode (open-input-file name))
    ('write-mode (open-output-file name))))))

(define (h-close handle) (λ ()
  (cond
    ((input-port? handle) (close-input-port handle))
    ((output-port? handle) (close-output-port handle))))))

; Unlike haskell these have to be in the IO monad because they can be changed
(define stdin (λ () (current-input-port)))
(define stdout (λ () (current-output-port)))

(define (read-file name) (do io:
  (h <- (open-file name 'read-mode))
  (cs <- (h-get-contents h))
  (h-close h)
  (io:return cs)))

(define (write-file name s) (do io:
  (h <- (open-file name 'write-mode))
  (h-put-str h s)
  (h-close h)))

(define get-char (io:bind stdin (λ (h) (h-get-char h))))
(define put-char c (io:bind stdout (λ (h) (h-put-char c))))
(define is-eof? (io:bind stdin (λ (h) (h-is-eof? h))))
(define put-str s (io:bind stdout (λ (h) (h-put-str h s))))
(define put-str-ln s (io:bind stdout (λ (h) (h-put-str-ln h s))))

```

```

; Brian Alliet
; Programming Language Theory
; 4005-710-01
; Copyright 2005 Brian Alliet

; list-perms generates a list of all the permutations of a given list
(define-match list-perms
  ('() '()) ; there is only one permutation of the empty list
  ; the permutations of a non empty list is every element of the list ;
  ; follow by each permutation of the remaining elements
  (xs (list-comp (cons x ps) (x <- xs) (ps <- (list-perms (delete x xs))))))

; perms finds all the permutations of the list of numbers in the range [1,n]
(define (perms n) (list-perms (from-to 1 n)))

; next-perm finds the next permutation in a sequence of #f if there are none left
(define-match next-perm
  ; the empty list and lists with a single element have no other permutations
  ('() #f)
  ((list _) #f)
  ((cons x xs)
   ; first just try to permute the tail
   (match (next-perm xs)
    ; couldn't permute the tail anymore, reverse it to get it back in ascending order
    ('#f (let ((xs2 (reverse xs)))
            ; find an element bigger than the current head in xs2
            (match (filter (right-section > x) xs2)
             ; found one, put that back in front and put the current head in its place
             ((cons x2 _) (cons x2 (replace x2 x xs2)))
             ; didn't find one, no more permutations are possible
             ('() #f))))
    ; we could permute the tail, stick the car back on and return
    (xs2 (cons x xs2))))))

; permute finds all the permutations of a given list using next-perm
(define (permute xs)
  (cons xs ; the given list is always a valid permutation, so cons that on to the results
        (match (next-perm xs) ; find the next one
          ('#f '()) ; no next permutation, terminate the list
          (xs2 (permute xs2))))))

(define (main args) (unsafe-perform-io (do io:
  (put-str-ln (show (perms 3)))
  (put-str-ln (show (permute (from-to 1 3))))
  (io:return (if
    (and
      (equal? (permute (from-to 1 6)) (perms 6))
      (= (length (perms 7)) 5040))
    0 1))
  )))

```

```

; Brian Alliet
; Programming Language Theory
; 4005-710-01
; Copyright 2005 Brian Alliet

; Sudoku stuff for Homework 2

(define (backtrack grid)
  (let
    ((p (undecided grid))
     (if (not p)
         (list grid) ; it is solved, return this as a solution
         (apply append (map (λ (x) (solve (substitute x p grid))) (list-ref grid p))))))

(define (solve grid)
  (let
    ((grid2 (simplify grid))
     (if (memq #f grid2)
         '() ; if there are "impossible" cells there is no solution
         (backtrack grid2)))) ; if not, backtrack (which won't do anything for simple puzzles)

; split-at splits a list in two at position n. The result is a
; pair containing the two lists
; (split-at 3 '(1 2 3 4 5)) => '((1 2 3) . (4 5))
(define (split-at n xs)
  (if (null? xs)
      (cons '() '())
      (if (= 0 n)
          (cons '() xs)
          (let
            ((rest (split-at (- n 1) (cdr xs))))
              (cons (cons (car xs) (car rest)) (cdr rest))))))

; Reduce uses prune to eliminate impossible candidates from the grid
(define (reduce grid)
  (letrec
    ; Reduce-one reduces a single element then recurses on the rest
    ; of the grid
    ((reduce-one (λ (n xs)
                  (if (null? xs) ; if we hit the end of the grid we're done
                      '()
                      (let*
                        ((x (car xs)) ; save the current cell
                         ; if this cell is 0 then replace it with all possible
                         ; candidates
                         (x (if (eqv? 0 x) '(1 2 3 4 5 6 7 8 9) x)))
                          ; cons the result on to the results of the rest of
                          ; the grid
                          (cons
                           ; if it is a pair (list) we have to prune it
                           ; if not, return it unmodified
                           (if (pair? x)
                               (let*
                                 ; Split the grid at the current element
                                 ((split (split-at n grid))
                                  ; prune the current element
                                  (pruned (prune (car split) x (cddr split))))
                                 (cond
                                  ; if we eliminated everything we're screwed
                                  ; we'll use #f to represent "Impossible"
                                  ((null? pruned) #f)
                                  ; If the pruned list has a single element
                                  ; then it is a known correct value
                                  ((null? (cdr pruned)) (car pruned))
                                  ; If not, just return the new pruned list
                                  (else pruned)))
                               x)
                           ; call ourselves on the rest of the list
                           (reduce-one (+ n 1) (cdr xs))))))))))

  (let
    ; start off reducing the whole grid (which starts with element 0)
    ((new (reduce-one 0 grid))
     (cell-length (λ (cell) (if cell (if (pair? cell) (length cell) 1) 0))))
    ; if we make some changes make a second pass
    (if (equal? (map cell-length grid) (map cell-length new))
        ))
  )

```

```

        grid
        (reduce new))))))
; END PUBLIC

;checks to see if item is at position in vector - returns true or false...
(define in-vector-at-position?
  (λ (vector item position)
    (cond ( (equal? (vector? vector) #f) #f)
          (else
           (if (equal? (vector-ref vector position) item) #t #f))))))

(define ok??-helper
  (λ (digit position cells elements)
    (cond ((null? cells) #t)
          ((equal? position (car cells)) (ok??-helper digit position (cdr cells) elements)) ;skips over position if it is the current "car"
          (else
           (if (equal? (in-vector-at-position? elements digit (car cells)) #t ) ;if index at car of cells is in the vector at that index, stop and return false since it was found
               #f
               (ok??-helper digit position (cdr cells) elements)))))) ; if not found at current index, continue to loop through all indexes

;is true exactly if the (non-zero) digit has not been entered into certain cells of the vector elements.
;false if it does exist, although position will be skipped if it is given within vector
;The list cells contains the indices to check;
;however, position is the index of digit in elements and this value must be skipped when checking out cells.
(define ok??
  (λ (digit position cells elements)
    ;check to make sure values are appropriate
    (cond ( (equal? (or (equal? (null? cells) #t) (equal? (vector? elements) #f)) #t) #f) ;not indicates to check so can't exist or not vector as incoming
          (else
           (ok??-helper digit position cells elements)))) ;;activates checking helper function

;ok-question.scm
;Mike Mertsock, PLT 20051
;Provides the ok? function for the Sudoku group project
;usage: see documentation for ok? -- at bottom of file
;test harness:
; ok-question-test.scm provides a test harness
; for working within the scope of the finished ok? and ok??
; functions.

;Helper for ok?-checkgroup.
;Calls ok??
;Checks if a particular index within the Sudoku solution
;is legal within the given group of elements.
;param velements: a VECTOR of the Sudoku solution
;param group: an item from rows/columns/boxes: a list
; indexes into the velements vector.
(define ok?-checki (λ (index group velements)
  (if (number? (vector-ref velements index))
      (if (> (vector-ref velements index) 0)
          (ok?? (vector-ref velements index) index group velements)
          #t ;ignore zero entries
      )
      #t ;ignore non-numeric (list) entries
  )
))

;Helper for ok?-checkgroups.
;Checks if all of finalized solution elements are legal
;within a single specified group (row/column/box).
;param velements: a VECTOR of the Sudoku solution
;param group: a list representing one row/col/box of indexes
(define ok?-checkgroup (λ (velements group)
  ;for each index in group
  (letrec
    ((check-partial (λ (grouppart)
      (if (null? grouppart)

```



```

( 18 19 20 21 22 23 24 25 26)
( 27 28 29 30 31 32 33 34 35)
( 27 28 29 30 31 32 33 34 35)
( 27 28 29 30 31 32 33 34 35)
( 27 28 29 30 31 32 33 34 35)
( 27 28 29 30 31 32 33 34 35)
( 27 28 29 30 31 32 33 34 35)
( 27 28 29 30 31 32 33 34 35)
( 27 28 29 30 31 32 33 34 35)
( 27 28 29 30 31 32 33 34 35)
( 36 37 38 39 40 41 42 43 44)
( 36 37 38 39 40 41 42 43 44)
( 36 37 38 39 40 41 42 43 44)
( 36 37 38 39 40 41 42 43 44)
( 36 37 38 39 40 41 42 43 44)
( 36 37 38 39 40 41 42 43 44)
( 36 37 38 39 40 41 42 43 44)
( 36 37 38 39 40 41 42 43 44)
( 36 37 38 39 40 41 42 43 44)
( 45 46 47 48 49 50 51 52 53)
( 45 46 47 48 49 50 51 52 53)
( 45 46 47 48 49 50 51 52 53)
( 45 46 47 48 49 50 51 52 53)
( 45 46 47 48 49 50 51 52 53)
( 45 46 47 48 49 50 51 52 53)
( 45 46 47 48 49 50 51 52 53)
( 45 46 47 48 49 50 51 52 53)
( 45 46 47 48 49 50 51 52 53)
( 45 46 47 48 49 50 51 52 53)
( 54 55 56 57 58 59 60 61 62)
( 54 55 56 57 58 59 60 61 62)
( 54 55 56 57 58 59 60 61 62)
( 54 55 56 57 58 59 60 61 62)
( 54 55 56 57 58 59 60 61 62)
( 54 55 56 57 58 59 60 61 62)
( 54 55 56 57 58 59 60 61 62)
( 54 55 56 57 58 59 60 61 62)
( 54 55 56 57 58 59 60 61 62)
( 54 55 56 57 58 59 60 61 62)
( 63 64 65 66 67 68 69 70 71)
( 63 64 65 66 67 68 69 70 71)
( 63 64 65 66 67 68 69 70 71)
( 63 64 65 66 67 68 69 70 71)
( 63 64 65 66 67 68 69 70 71)
( 63 64 65 66 67 68 69 70 71)
( 63 64 65 66 67 68 69 70 71)
( 63 64 65 66 67 68 69 70 71)
( 63 64 65 66 67 68 69 70 71)
( 63 64 65 66 67 68 69 70 71)
( 72 73 74 75 76 77 78 79 80)
( 72 73 74 75 76 77 78 79 80)
( 72 73 74 75 76 77 78 79 80)
( 72 73 74 75 76 77 78 79 80)
( 72 73 74 75 76 77 78 79 80)
( 72 73 74 75 76 77 78 79 80)
( 72 73 74 75 76 77 78 79 80)
( 72 73 74 75 76 77 78 79 80)
( 72 73 74 75 76 77 78 79 80)
))

(define columns '#( ; maps position to all positions in it's column
( 0 9 18 27 36 45 54 63 72)
( 1 10 19 28 37 46 55 64 73)
( 2 11 20 29 38 47 56 65 74)
( 3 12 21 30 39 48 57 66 75)
( 4 13 22 31 40 49 58 67 76)
( 5 14 23 32 41 50 59 68 77)
( 6 15 24 33 42 51 60 69 78)
( 7 16 25 34 43 52 61 70 79)
( 8 17 26 35 44 53 62 71 80)
( 0 9 18 27 36 45 54 63 72)
( 1 10 19 28 37 46 55 64 73)
( 2 11 20 29 38 47 56 65 74)
( 3 12 21 30 39 48 57 66 75)
( 4 13 22 31 40 49 58 67 76)
( 5 14 23 32 41 50 59 68 77)
( 6 15 24 33 42 51 60 69 78)
( 7 16 25 34 43 52 61 70 79)
( 8 17 26 35 44 53 62 71 80)

```

```

( 0 9 18 27 36 45 54 63 72)
( 1 10 19 28 37 46 55 64 73)
( 2 11 20 29 38 47 56 65 74)
( 3 12 21 30 39 48 57 66 75)
( 4 13 22 31 40 49 58 67 76)
( 5 14 23 32 41 50 59 68 77)
( 6 15 24 33 42 51 60 69 78)
( 7 16 25 34 43 52 61 70 79)
( 8 17 26 35 44 53 62 71 80)
( 0 9 18 27 36 45 54 63 72)
( 1 10 19 28 37 46 55 64 73)
( 2 11 20 29 38 47 56 65 74)
( 3 12 21 30 39 48 57 66 75)
( 4 13 22 31 40 49 58 67 76)
( 5 14 23 32 41 50 59 68 77)
( 6 15 24 33 42 51 60 69 78)
( 7 16 25 34 43 52 61 70 79)
( 8 17 26 35 44 53 62 71 80)
( 0 9 18 27 36 45 54 63 72)
( 1 10 19 28 37 46 55 64 73)
( 2 11 20 29 38 47 56 65 74)
( 3 12 21 30 39 48 57 66 75)
( 4 13 22 31 40 49 58 67 76)
( 5 14 23 32 41 50 59 68 77)
( 6 15 24 33 42 51 60 69 78)
( 7 16 25 34 43 52 61 70 79)
( 8 17 26 35 44 53 62 71 80)
( 0 9 18 27 36 45 54 63 72)
( 1 10 19 28 37 46 55 64 73)
( 2 11 20 29 38 47 56 65 74)
( 3 12 21 30 39 48 57 66 75)
( 4 13 22 31 40 49 58 67 76)
( 5 14 23 32 41 50 59 68 77)
( 6 15 24 33 42 51 60 69 78)
( 7 16 25 34 43 52 61 70 79)
( 8 17 26 35 44 53 62 71 80)
( 0 9 18 27 36 45 54 63 72)
( 1 10 19 28 37 46 55 64 73)
( 2 11 20 29 38 47 56 65 74)
( 3 12 21 30 39 48 57 66 75)
( 4 13 22 31 40 49 58 67 76)
( 5 14 23 32 41 50 59 68 77)
( 6 15 24 33 42 51 60 69 78)
( 7 16 25 34 43 52 61 70 79)
( 8 17 26 35 44 53 62 71 80)
( 0 9 18 27 36 45 54 63 72)
( 1 10 19 28 37 46 55 64 73)
( 2 11 20 29 38 47 56 65 74)
( 3 12 21 30 39 48 57 66 75)
( 4 13 22 31 40 49 58 67 76)
( 5 14 23 32 41 50 59 68 77)
( 6 15 24 33 42 51 60 69 78)
( 7 16 25 34 43 52 61 70 79)
( 8 17 26 35 44 53 62 71 80)
))

(define boxes '#( ; maps position to all positions in it's box
( 0 1 2 9 10 11 18 19 20)
( 0 1 2 9 10 11 18 19 20)
( 0 1 2 9 10 11 18 19 20)
( 3 4 5 12 13 14 21 22 23)
( 3 4 5 12 13 14 21 22 23)
( 3 4 5 12 13 14 21 22 23)
( 6 7 8 15 16 17 24 25 26)
( 6 7 8 15 16 17 24 25 26)
( 6 7 8 15 16 17 24 25 26)
( 0 1 2 9 10 11 18 19 20)

```

```

( 0 1 2 9 10 11 18 19 20)
( 0 1 2 9 10 11 18 19 20)
( 3 4 5 12 13 14 21 22 23)
( 3 4 5 12 13 14 21 22 23)
( 3 4 5 12 13 14 21 22 23)
( 6 7 8 15 16 17 24 25 26)
( 6 7 8 15 16 17 24 25 26)
( 6 7 8 15 16 17 24 25 26)
( 0 1 2 9 10 11 18 19 20)
( 0 1 2 9 10 11 18 19 20)
( 0 1 2 9 10 11 18 19 20)
( 3 4 5 12 13 14 21 22 23)
( 3 4 5 12 13 14 21 22 23)
( 3 4 5 12 13 14 21 22 23)
( 6 7 8 15 16 17 24 25 26)
( 6 7 8 15 16 17 24 25 26)
( 6 7 8 15 16 17 24 25 26)
( 27 28 29 36 37 38 45 46 47)
( 27 28 29 36 37 38 45 46 47)
( 27 28 29 36 37 38 45 46 47)
( 30 31 32 39 40 41 48 49 50)
( 30 31 32 39 40 41 48 49 50)
( 30 31 32 39 40 41 48 49 50)
( 33 34 35 42 43 44 51 52 53)
( 33 34 35 42 43 44 51 52 53)
( 33 34 35 42 43 44 51 52 53)
( 27 28 29 36 37 38 45 46 47)
( 27 28 29 36 37 38 45 46 47)
( 27 28 29 36 37 38 45 46 47)
( 30 31 32 39 40 41 48 49 50)
( 30 31 32 39 40 41 48 49 50)
( 30 31 32 39 40 41 48 49 50)
( 33 34 35 42 43 44 51 52 53)
( 33 34 35 42 43 44 51 52 53)
( 33 34 35 42 43 44 51 52 53)
( 27 28 29 36 37 38 45 46 47)
( 27 28 29 36 37 38 45 46 47)
( 27 28 29 36 37 38 45 46 47)
( 30 31 32 39 40 41 48 49 50)
( 30 31 32 39 40 41 48 49 50)
( 30 31 32 39 40 41 48 49 50)
( 33 34 35 42 43 44 51 52 53)
( 33 34 35 42 43 44 51 52 53)
( 33 34 35 42 43 44 51 52 53)
( 54 55 56 63 64 65 72 73 74)
( 54 55 56 63 64 65 72 73 74)
( 54 55 56 63 64 65 72 73 74)
( 57 58 59 66 67 68 75 76 77)
( 57 58 59 66 67 68 75 76 77)
( 57 58 59 66 67 68 75 76 77)
( 57 58 59 66 67 68 75 76 77)
( 60 61 62 69 70 71 78 79 80)
( 60 61 62 69 70 71 78 79 80)
( 60 61 62 69 70 71 78 79 80)
( 60 61 62 69 70 71 78 79 80)
( 54 55 56 63 64 65 72 73 74)
( 54 55 56 63 64 65 72 73 74)
( 54 55 56 63 64 65 72 73 74)
( 57 58 59 66 67 68 75 76 77)
( 57 58 59 66 67 68 75 76 77)
( 57 58 59 66 67 68 75 76 77)
( 60 61 62 69 70 71 78 79 80)
( 60 61 62 69 70 71 78 79 80)
( 60 61 62 69 70 71 78 79 80)
( 54 55 56 63 64 65 72 73 74)
( 54 55 56 63 64 65 72 73 74)
( 54 55 56 63 64 65 72 73 74)
( 57 58 59 66 67 68 75 76 77)
( 57 58 59 66 67 68 75 76 77)
( 57 58 59 66 67 68 75 76 77)
( 60 61 62 69 70 71 78 79 80)
( 60 61 62 69 70 71 78 79 80)
( 60 61 62 69 70 71 78 79 80)
))
(define groups '( ; list of lists of positions in each row/column/box
; rows
( 0 1 2 3 4 5 6 7 8)

```

```

( 9 10 11 12 13 14 15 16 17)
( 18 19 20 21 22 23 24 25 26)
( 27 28 29 30 31 32 33 34 35)
( 36 37 38 39 40 41 42 43 44)
( 45 46 47 48 49 50 51 52 53)
( 54 55 56 57 58 59 60 61 62)
( 63 64 65 66 67 68 69 70 71)
( 72 73 74 75 76 77 78 79 80)
; columns
( 0 9 18 27 36 45 54 63 72)
( 1 10 19 28 37 46 55 64 73)
( 2 11 20 29 38 47 56 65 74)
( 3 12 21 30 39 48 57 66 75)
( 4 13 22 31 40 49 58 67 76)
( 5 14 23 32 41 50 59 68 77)
( 6 15 24 33 42 51 60 69 78)
( 7 16 25 34 43 52 61 70 79)
( 8 17 26 35 44 53 62 71 80)
; boxes
( 0 1 2 9 10 11 18 19 20)
( 3 4 5 12 13 14 21 22 23)
( 6 7 8 15 16 17 24 25 26)
( 27 28 29 36 37 38 45 46 47)
( 30 31 32 39 40 41 48 49 50)
( 33 34 35 42 43 44 51 52 53)
( 54 55 56 63 64 65 72 73 74)
( 57 58 59 66 67 68 75 76 77)
( 60 61 62 69 70 71 78 79 80)
))

(define prune
  (λ (head candidates tail)
    (let each ((c candidates) (fit '()))
      (if (null? c)
          fit
          (if (ok? (append head (cons (car c) tail)))
              (each (cdr c) (cons (car c) fit))
              (each (cdr c) fit))))))

;checks a list for a digit
;done by josh harlow
;for suduko probelm
(define check-list
  (λ (list-in digit)
    (cond ((null? list-in) #f)
          (else
           (if (equal? #f (list? list-in)) ; incoming list is only one digit
               #f
               (if (equal? (car list-in) digit)
                   #t
                   (check-list (cdr list-in) digit)))))))

;see's how many times the number is repeating
(define num-times-repeating
  (λ (digit group start-num)
    (if (null? group)
        start-num
        (let ((found-in-own-list (check-list (car group) digit)))
          (if (equal? #t found-in-own-list)
              (num-times-repeating digit (cdr group) (+ 1 start-num))
              (num-times-repeating digit (cdr group) start-num))))))

;locates the first position where it occurs
;used only when num-times-repeating returns 1
(define find-group-num
  (λ (digit group start-num)
    (cond ( (null? group) #f)
          ( else
           ( if (equal? #t (check-list (car group) digit))
               start-num
               (find-group-num digit (cdr group) (+ start-num 1))))))

;activator

```

```

(define unique-slot
  (λ (digit group)
    (let (( num-times (num-times-repeating digit group 0)))
      (cond ( (equal? 0 num-times) #f)
            ( (equal? 1 num-times) (find-group-num digit group 0))
            (else
             #f))))))

;Mike Mertsock, PLT 20051

;Attempts to find a unique final location to place one
;digit within the group of grid locations.
;param group: A list Suduko grid entries.
; example: '(1 (2 3) 5 0 (4 2 6) 7 0 0 8)
;returns: if successful, returns a list with two elements.
;First element is the digit for which we found a location, and
;the second element is the index into the specified group at
;which to place the digit. Returns #f if this attempt fails.
; example: returns '(3 1) for the group above (place 3 at index 1).

(define unique-digit-slot (λ (group)
  ;for each digit 1 thru 9, try to find unique-slot.
  ;if found, return immediately, else try n+1
  (let loop ((n 1))
    (if (> n 9)
        #f
        (let ((result (unique-slot n group)))
          (if (number? result)
              (list n result)
              (loop (+ n 1)))
          )
        )
    ))
))

; Brian Alliet
; Programming Language Theory
; 4005-710-01
; Copyright 2005 Brian Alliet

; Sudoku stuff for Homework 2

(define (substitute r n xs)
  (if (null? xs)
      '()
      (if (= 0 n)
          (cons r (cdr xs))
          (cons (car xs) (substitute r (- n 1) (cdr xs))))))

(define undecided
  (λ (elements)
    (λ (elements) (idx 0))
    (let index ((e elements) (idx 0))
      (if (null? e)
          #f
          (if (list? (car e))
              idx
              (index (cdr e) (+ 1 idx)))
          )
      )
    ))))

; Lomax Escarmant
; Programming Language Theory.

; First applies reduce and then tries to substitute for
; unique-digit-slot in each row, coloumn, and box. If the puzzle cannot
; be solved this will return false.

(define simplify
  (λ (elements)

```

```

(simplify-rec groups (reduce elements))))
; Recurses until all rows, columns and boxes are checked
(define simplify-rec
  (λ (grps elements)
    (if (null? grps)
        ; All rows, columns and boxes traversed.
        elements
        ;(if (undecided elements)
            ; Still undecided terms, puzzle not solved.
            ;#f
            ;elements ; <-- should be false, view for testing purposes
            ; Puzzle solved.
            ;elements)
        ; There are more rows, columns and boxes to traverse
        ; Recurse on rows, columns and boxes
        ((λ (sub)
          (if sub
              ; Found unique digit slot so substitute and start from
              ; beginning.
              (simplify (apply substitute
                        (list (car sub)
                             (list-ref (car grps) (cadr sub))
                             elements)))
              ; Did not find unique digit slot
              ; continue in hopes next row, col or box
              ; will work.
              (simplify-rec (cdr grps) elements))))
         ; Attempt to find unique digit slot and pass
         ; results to preceding lambda
         (unique-digit-slot (select (car grps)
                                   elements))))))

(define f '(
  0 0 0 1 0 0 7 0 2
  0 3 0 9 5 0 0 0 0
  0 0 1 0 0 2 0 0 3

  5 9 0 0 0 0 3 0 1
  0 2 0 0 0 0 0 7 0
  7 0 3 0 0 0 0 9 8

  8 0 0 2 0 0 1 0 0
  0 0 0 0 8 5 0 6 0
  6 0 5 0 0 9 0 0 0 ))

(define t '( ; tough
  8 3 0 0 0 0 0 4 6
  0 2 0 1 0 4 0 3 0
  0 0 0 0 0 0 0 0 0

  0 0 2 9 0 6 5 0 0
  1 4 0 0 0 0 0 2 3
  0 0 5 4 0 3 1 0 0

  0 0 0 0 0 0 0 0 0
  0 6 0 3 0 8 0 7 0
  9 5 0 0 0 0 0 6 2 ))

(define d '( ; diabolical
  8 0 0 7 0 1 0 0 2
  0 0 6 0 0 0 7 0 0
  0 1 7 0 0 0 8 9 0

  0 0 0 1 7 3 0 0 0
  7 0 0 0 0 0 0 0 6
  0 0 0 9 5 6 0 0 0

  0 9 5 0 0 0 4 1 0
  0 0 8 0 0 0 5 0 0
  3 0 0 6 0 5 0 0 7 ))

```

```
(define hard '(
```

```

0 0 0 8          0 2          0 0 0
5              0 0 0 0 0 0 0 1    0 3          0 0
0 0 6          0 5          0 8          0 0
0 0 9          0 1          0 0 0          0 0
1              0 0 0 0 0 0 0 2    0 0 0
0 0 0 9        0 7          0 0 0
0 6           1          0 3          0 7          8
0           0
0 5           0 0 0 0 0 4    0
0 7           2          0 4          0 1          5
0
))

(define hard2 '(
3          0 0 2          0 0 9          0 0
0 0 0 0 0 0 0 0 5    0 1          0 4          0 0 0
0 7           0 0 0 8          0 0
0 0 9        0 0 0 7          0 0 0 6
5           0 0 0 2          0 0
0 0 1        0 9          0 4          0
0 0 0 3      0 0 0 0 0 0 0 0    0 0 7
8           0 0 5
0 0 6
))

```

```

; Brian Alliet
; Programming Language Theory
; 4005-710-01
; Copyright 2005 Brian Alliet

; Part A)

; The following expression evaluates to itself.
; ((lambda (x) (cons x (cons (cons 'quote (cons x '())) '())) ' (lambda (x) (cons x (cons (cons 'quote (cons x '())) '()))))
(define quine
  ((lambda (x) (cons x (cons (cons 'quote (cons x '())) '())) ' (lambda (x) (cons x (cons (cons 'quote (cons x '())) '()))))
  )

; This is the curry* function
; When given arguments it curry*'s another lambda that adds our arguments
; back in front before actually applying the function
(define (curry* f) (lambda args1
  (if (null? args1)
      (f)
      (curry* (lambda args2 (apply f (append args1 args2)))))))

; Part B)

; First we define some syntax rules to make creating objects easier (and generic)
; We could actually do this without syntax rules but using them is more efficient

; To make-object makes an object. Its first argument is a list of (method-name value) pairs
; and its (optional) second argument is a "default" method. If a default is not specified
; an unspecific value is returned when an unknown method is called.
(define-syntax make-object
  (syntax-rules ()
    ((make-object (methods ...)) (make-object (methods ...) (lambda x (if #f #f))))
    ((make-object (methods ...) default)
     (lambda args
       (if (pair? args)
           ; If an object is called with one or more argument see if the first is a method
           ; We use make-object-clauses (described below) to handle checking for and calling
           ; the method
           (let
              ((m (car args)) (margs (cdr args)))
              (make-object-clauses m margs (apply default args) methods ...))
           ; If not, just call the default method
           (default))))))

; make-object-clauses generates a series of if's to check for known methods (specified
; as the arguments after the third). If checks for the method named method_ and if found,
; passes it arguments args. If no match is found failure is returned.
(define-syntax make-object-clauses
  (syntax-rules ()
    ((make-object-clauses method_ args failure) failure)
    ((make-object-clauses method_ args failure (method value) rest ...)
     (if (eq? method method_)
         (apply value args) ; found the method, call it with the args
         (make-object-clauses method_ args failure rest ...))) ; keep looking
    ))

; Now, defining the conversion class (which is really a plain old object) is simply a
; matter of calling make-object with our methods
(define conversion (make-object
  ; The 'new method makes a conversion object with the a function in the form ax+b
  (('new (lambda (b a)
    (conversion 'new-with-functions (lambda (x) (+ (* a x) b)) (lambda (x) (/ (- x b) a))))))
  ; 'new-with-functions makes a new conversion object with a given function and its inverse
  ('new-with-functions (lambda (f inverse)
    ; The result of the call is a conversion "instance"
    (make-object
      ; This has an 'inverse method, which returns a new conversion instance
      ; with the function inverted
      (('inverse (lambda () (conversion 'new-with-functions inverse f))))
      ; The default method applies the function to all its arguments
      (lambda args (map f args)))))))

```

```
; We define bindings for new and inverse so we can use those names unquoted  
(define new 'new)  
(define inverse 'inverse)
```

```

; Brian Alliet
; Programming Language Theory
; 4005-710-01

(define (lexical-address expr)
  (call-with-current-continuation
    (λ (return)
      (lexical-address-helper (λ () (return #f)) 0 '() expr))))

(define (lexical-address-helper fail depth env expr)
  (match expr
    ((list 'λ args body)
     (list 'λ args (lexical-address-helper
                   fail (+ depth 1) (append
                                     (zip (λ (a p) (cons a (cons depth p))) args (from-to 0 (- (length args) 1)))
                                     env)
                   body)))
    ((xs @ (cons _ _)) (map (curry lexical-address-helper fail depth env) xs))
    (sym
     (if (symbol? sym)
         (match (lookup sym env)
             ((cons a p) (list ': a p))
             ('#f (list sym 'free)))
         (fail))))))

(define (un-lexical-address expr)
  (call-with-current-continuation
    (λ (return)
      (un-lexical-address-helper (λ () (return #f)) '() expr))))

(define (un-lexical-address-helper fail env expr)
  (match expr
    ((list 'λ args body)
     (list 'λ args (un-lexical-address-helper fail (append env (list args)) body)))
    ((list ': d p)
     (if (>= d (length env))
         (fail)
         (let ((names (list-ref env d)))
             (if (>= p (length names))
                 (fail)
                 (list-ref names p))))))
    ((list sym 'free) sym)
    ((xs @ (cons _ _)) (map (curry un-lexical-address-helper fail env ) xs))
    (_ (fail))))

```

```

; term: "2-unify.scm"
; This is just the datatype for term. There are variable terms ("x", "foo", etc),
; constant terms (1, 2, #f, etc), and application terms ((func arg), (bar quux))
(define-datatype term term?
  (var-term
   (id symbol?))
  (constant-term
   (datum constant?))
  (app-term
   (terms (list-of term?))))

; unify-term: "2-unify.scm"
; This is the function that does the actual unification
; Anything can be substituted for a variable term as long as the
; variable being substituted for doesn't appear in the substitution
; A constant term can be substituted for an identical constant
; An application term can be substituted with another application term
; if each component can also be unified
(define unify-term
  (λ (t u)
    (cases term t
      (var-term (tid)
        (if (or (var-term? u) (not (memv tid (all-ids u))))
            (unit-subst tid u)
            #f))
      (else
        (cases term u
          (var-term (uid) (unify-term u t))
          (constant-term (udatum)
            (cases term t
              (constant-term (tdatum)
                (if (equal? tdatum udatum) (empty-subst) #f))
              (else #f)))
          (app-term (us)
            (cases term t
              (app-term (ts) (unify-terms ts us))
              (else #f))))))))))

; constant?: "2-unify.scm"
; The constant? predicate checks if a value is a valid "constant"
; (number, boolean, string, or null)
(define constant?
  (λ (x)
    (or (boolean? x) (number? x) (string? x) (null? x))))

; var-term?: "2-unify.scm"
; The var-term? predicate checks if a term is a variable term
(define var-term?
  (λ (t)
    (cases term t
      (var-term (id) #t)
      (else #f))))

; all-ids: "2-unify.scm"
; all-ids returns a list of all the identifiers used in a term
(define all-ids
  (λ (t)
    (cases term t
      (var-term (id) (list id))
      (constant-term (datum) '())
      (app-term (ts) (all-ids-terms ts))))))

; all-ids-terms: "2-unify.scm"
; all-ids-terms returns a list of all the unique identifiers in a list of terms
(define all-ids-terms
  (λ (ts)
    (map-union all-ids ts)))

; unit-subst: "2-unify.scm"
; unit-subst returns a substitution for a single symbol with a term
(define unit-subst
  (λ (id new-term)
    (λ (id2)
      (if (eq? id2 id) new-term (var-term id2)))))

```

```

; empty-subst: "2-unify.scm"
; empty-subst is the empty substitution
(define empty-subst
  (λ ()
    (λ (id) (var-term id))))

; compose-subst: "2-unify.scm"
; compose-substs composes two substitutions
; it returns a substitution that applies s1 then s2
(define compose-substs
  (λ (s1 s2)
    (λ (id)
      (subst-in-term (apply-subst s1 id) s2))))

; apply-subst: "2-unify.scm"
; apply-subst applies a substitution to an identifier
(define apply-subst
  (λ (subst id)
    (subst id)))

; unify-terms: "2-unify.scm"
; unify-terms unifies a list of terms of equal length
; two empty lists result in an empty substitution
; an empty and non-empty list results in no substitution
; anything else results in the substitution for the car of each list
; composed with the substitution for the rest of each list
(define unify-terms
  (λ (ts us)
    (cond
      ((and (null? ts) (null? us)) (empty-subst))
      ((or (null? ts) (null? us)) #f)
      (else
       (let ((subst-car (unify-term (car ts) (car us))))
         (if (not subst-car)
             #f
             (let ((new-ts (subst-in-terms (cdr ts) subst-car))
                   (new-us (subst-in-terms (cdr us) subst-car)))
               (let ((subst-cdr (unify-terms new-ts new-us)))
                 (if (not subst-cdr)
                     #f
                     (compose-substs subst-car subst-cdr)))))))))))

; map-union: "2-unify.scm"
; map-union maps a function that returns a list to every element of a list
; the result is the union of the results of each application result
(define map-union
  (λ (f ls)
    (cond
      ((null? ls) '())
      (else (union (f (car ls)) (map-union f (cdr ls)))))))

; union: "2-unify.scm"
; union finds the union of two sets (represented as lists)
(define union
  (λ (set1 set2)
    (cond
      ((null? set1) set2)
      ((memv (car set1) set2) (union (cdr set1) set2))
      (else (cons (car set1) (union (cdr set1) set2))))))

; subst-in-terms: "2-unify.scm"
; subst-in-terms maps a substitution to every term in a list of terms
(define subst-in-terms
  (λ (ts subst)
    (map (λ (t) (subst-in-term t subst)) ts)))

; subst-in-term: "2-unify.scm"
; subst-in-term runs a substitution on a single term
; var-terms get their symbol mapped with apply-substs
; constant terms are unchanged
; application terms have each component run against the substitution
(define subst-in-term
  (λ (t subst)
    (cases term t
      (var-term (id)

```

```
(apply-subst subst id))
(constant-term (datum)
  (constant-term datum))
(app-term (ts)
  (app-term (subst-in-terms ts subst))))))

; valid-substs checks is a substitution is "valid"
; this is used for sanity checks in the test framework
; for THIS implementation a substituton is a procedure so this is the
; best check we can do
(define valid-subst procedure?)
```

```
; Brian Alliet
; Programming Language Theory
; 4005-710-01

; This is the substitution datatype. There are three substitution
; The empty substitution (which does nothing)
; The unit substitution which substitutes a single term for a single symbol
; The composition substitution which combines two substitution into one
; it applies the first then the second
(define-datatype subst subst?
  (empty-subst)
  (unit-subst (id symbol?) (new-term term?))
  (compose-substs (s1 subst?) (s2 subst?)))

; apply-subst applies a substitution to an identifier
(define (apply-subst s id)
  (cases subst s
    (empty-subst () (var-term id))
    (unit-subst (id_ new) (if(eq? id id_) new (var-term id)))
    (compose-substs (s1 s2) (subst-in-term (apply-subst s1 id) s2))))

; In THIS implementation every valid substitution is of type subst
(define valid-subst subst?)
```

```
; Brian Alliet
; Programming Language Theory
; 4005-710-01

(define (test)
  (let
    ((try (λ (t u unified)
            (let*
              ((s (unify-term t u))
               (if
                (if unified
                 (and
                  (valid-subst s)
                  (equal? unified (subst-in-term t s))
                  (equal? unified (subst-in-term u s))
                  (not s))
                 #t
                 (error "Test failed")))))
            (foo (var-term 'foo))
            (bar (var-term 'bar))
            (baz (var-term 'baz))
            (quux (var-term 'quux))
            (one (constant-term 1))
            (two (constant-term 2))
            )
          )
    (begin
      ; Simple variable substitution
      (try foo bar bar)
      ; Substituting an expression for a variable
      (try foo (app-term (list bar baz)) (app-term (list bar baz)))
      ; Equal constants are ok
      (try one one one)
      ; Equal applications are ok (as long as each component can be substituted)
      (try (app-term (list foo bar)) (app-term (list baz quux)) (app-term (list baz quux)))
      ; Substituting an expression for a variable isn't ok always
      (try foo (app-term (list foo bar)) #f)
      ; Unequal constants are not ok
      (try one two #f)
      ; Mismatched non variable types aren't ok
      (try one (app-term (list foo two)) #f)
      )))
```

```
; Brian Alliet
; Programming Language Theory
; 4005-710-01

(define client1
  (let*
    ((x 0)
     (sequence (λ () (begin (set! x (+ x 1)) x)))
     (y (sequence))
     (z (sequence)))
    )
    (list x y z)
  ) )

(define (make-sequence x) (λ ()
  (begin (set! x (+ x 1)) x)))

(define (apply-sequence s) (s))

(define client2
  (let*
    ((x 0)
     (sequence (make-sequence x))
     (y (apply-sequence sequence))
     (z (apply-sequence sequence)))
    (list x y z)))

(define-datatype sequence2 sequence2?
  (make-sequence2 (x number?)))

(define (apply-sequence2 s)
  (cases sequence2 s
    (make-sequence2 (x)
      (begin (set! x (+ x 1)) x))))

(define client3
  (let*
    ((x 0)
     (sequence (make-sequence2 x))
     (y (apply-sequence2 sequence))
     (z (apply-sequence2 sequence)))
    (list x y z)))
```

```

; Brian Alliet
; Programming Language Theory
; 4005-710-01

; Turn any sexpression into a string
(define (sexp->string e)
  (letrec
    ((helper (λ (e rest)
              (cond
                ((null? e) (cons "" rest))
                ((pair? e) (cons "(" (finishpair e rest)))
                ((number? e) (cons (number->string e) rest))
                ((symbol? e) (cons (symbol->string e) rest))
                ; FIXME: These don't escape chars correctly (but this doesn't matter for hw5)
                ((char? e) (cons (list->string (list #\# #\\ e)) rest))
                ((string? e) (cons "\"" (cons e (cons "\"" rest))))
                ((vector? e) (cons "#" (helper (vector->list e) rest)))
                ((eq? e #t) (cons "#t" rest))
                ((eq? e #f) (cons "#f" rest))
                (else (cons "--FIXME--" rest))))))
    (finishpair (λ (e rest)
                 (helper (car e)
                        (cond
                          ((null? (cdr e)) (cons "" rest))
                          ((pair? (cdr e)) (cons " " (finishpair (cdr e) rest)))
                          (else (cons "." (helper (cdr e) (cons "" rest))))))))))
    (apply string-append (helper e '()))))

; Our expressions are just a normal sexpressions
(define expr->string sexp->string)

; Datatype that can old a la-expressions or ula-expression
(define-datatype expression expression?
  (la-expr (e la-expression?))
  (ula-expr (e ula-expression?)))

; Lexical spec for lambda expressions
(define lexical-spec
  '((white-sp (whitespace) skip)
    (id (letter (arbno (or letter digit "?")) symbol)
      (number (digit (arbno digit)) number)))

; Grammar for un-lexical-address expressions
(define ula-grammar
  '((ula-expression (id) ula-symbol-exp)
    (ula-expression "(" ula-p-expression ")") ula-p-exp)
  (ula-expression ("lambda" "(" (arbno id) ")") ula-expression) ula-lambda-exp)
  (ula-expression (ula-expression (arbno ula-expression)) ula-app-exp)))
(sllgen:make-define-datatypes lexical-spec ula-grammar)

; Grammar for lexical-address expressions
(define la-grammar
  '((la-expression "(" la-p-expression ")") la-p-exp)
  (la-expression ":" number number) la-bound-symbol-exp)
  (la-expression (id "free") la-free-symbol-exp)
  (la-expression ("lambda" "(" (arbno id) ")") la-expression) la-lambda-exp)
  (la-expression (la-expression (arbno la-expression)) la-app-exp)))
(sllgen:make-define-datatypes lexical-spec la-grammar)

; To parse the above convert them to strings, run them though the sllgen parser, then wrap them
; in an expression data type
(define (parse-la e) (la-expr ((sllgen:make-string-parser lexical-spec la-grammar) (expr->string e))))
(define (parse-ula e) (ula-expr ((sllgen:make-string-parser lexical-spec ula-grammar) (expr->string e))))

; Recursively convert a ula expression data type to a normal ula-expression
(define (ula-expression->sexp e)
  (cases ula-expression e
    (ula-symbol-exp (e) e)
    (ula-p-exp (e) (cases ula-p-expression e
                      (ula-lambda-exp (ids body) (list 'λ ids (ula-expression->sexp body)))
                      (ula-app-exp (f args) (map ula-expression->sexp (cons f args))))))

; Same for la
(define (la-expression->sexp e)

```

```

(cases la-expression e
  (la-p-exp (e) (cases la-p-expression e
    (la-bound-symbol-exp (x y) (list ': x y))
    (la-free-symbol-exp (x) (list x 'free))
    (la-lambda-exp (ids body) (list 'λ ids (la-expression->sexp body)))
    (la-app-exp (f args) (map la-expression->sexp (cons f args))))))

; To unparse an expressions run it though the above functions (after possibly changing its
; representation)
(define (unparse-ula e)
  (cases expression e
    (la-expr (e) (un-lexical-address (la-expression->sexp e)))
    (ula-expr (e) (ula-expression->sexp e))))

(define (unparse-la e)
  (cases expression e
    (la-expr (e) (la-expression->sexp e))
    (ula-expr (e) (lexical-address (ula-expression->sexp e)))))

; To test for equality just convert them both to the same form and compare
(define (equal-expression? e1 e2) (equal? (unparse-ula e1) (unparse-ula e2)))

; This is just the lexical-address code from hw3
(define (lexical-address expr)
  (call-with-current-continuation
    (λ (return)
      (lexical-address-helper (λ () (return #f)) 0 '() expr))))

(define (lexical-address-helper fail depth env expr)
  (match expr
    ((list 'λ args body)
     (list 'λ args (lexical-address-helper
       fail (+ depth 1) (append
         (zip (λ (a p) (cons a (cons depth p))) args (from-to 0 (- (length args) 1))
         env)
         body))))
    ((xs @ (cons _ _)) (map (curry lexical-address-helper fail depth env) xs))
    (sym
     (if (symbol? sym)
         (match (lookup sym env)
           ((cons a p) (list ': a p))
           ('#f (list sym 'free)))
         (fail))))))

(define (un-lexical-address expr)
  (call-with-current-continuation
    (λ (return)
      (un-lexical-address-helper (λ () (return #f)) '() expr))))

(define (un-lexical-address-helper fail env expr)
  (match expr
    ((list 'λ args body)
     (list 'λ args (un-lexical-address-helper fail (append env (list args)) body))))
    ((list ': d p)
     (if (>= d (length env))
         (fail)
         (let ((names (list-ref env d)))
           (if (>= p (length names))
               (fail)
               (list-ref names p))))))
    ((list sym 'free) sym)
    ((xs @ (cons _ _)) (map (curry un-lexical-address-helper fail env ) xs))
    (_ (fail))))

```

```

; Brian Alliet
; Programming Language Theory
; 4005-710-01

; Lexical spec and grammar for the lambda calculus format used in this assignment
(define lex-spec
  '((white-sp (whitespace) skip)
    (id ((or letter digit) (arbno (or letter digit "?" "-")) symbol)))

(define grammar
  '((parsed-prog ((arbno "$" id "=" parsed-expr) parsed-expr) parsed-prog_)
    (parsed-expr (id) parsed-var-expr)
    (parsed-expr ("(" parsed-pexpr ")") parsed-pexpr_)
    (parsed-pexpr ("lambda" "(" id ")") parsed-lambda-expr)
    (parsed-pexpr (parsed-expr parsed-expr) parsed-app-expr)))
(sllgen:make-define-datatypes lex-spec grammar)
(define parse (sllgen:make-string-parser lex-spec grammar))

; inline-macors inlines all macros defined in dict in the parsed expression pe
(define (inline-macros dict pe)
  (let
    ((fix (λ (sym) (or (lookup sym dict) (parsed-var-expr sym))))
     (inline (curry inline-macros dict)))
    (cases parsed-expr pe
      (parsed-var-expr (id) (fix id))
      (parsed-pexpr_ (ppe) (parsed-pexpr_
        (cases parsed-pexpr ppe
          (parsed-lambda-expr (id body) (parsed-lambda-expr id (inline body)))
          (parsed-app-expr (e1 e2) (parsed-app-expr (inline e1) (inline e2))))))))))

; fix-dict inlines all macros previously defined within the body of all subsequent macros
(define-match fix-dict
  ((cons x xs) (cons x (fix-dict (map (lambda-match ((cons k v) (cons k (inline-macros (list x) v)
    ))) xs))))
  ('() '()))

; expand inlines all the macros in a parsed program and returns the resulting expressions
(define (expand p)
  (cases parsed-prog p
    (parsed-prog_ (names values body)
      (inline-macros (fix-dict (zip names values)) body))))

; We represent variable names as (name, subscript) pairs
; The subscript is used to generate unique names during alpha conversion
(define (varname n) (cons n 0))
(define-match (varname? x)
  ((cons n s) (and (symbol? n) (number? s)))
  (_ #f))

; This is our actual term data type, it is easier to work with than
; the parsed-expr output by sllgen
(define-datatype term term?
  (var (v varname?)) ; variable
  (app (e1 term?) (e2 term?)); aplication
  (lam (id varname?) (body term?)); lambda

; convert a parsed expression to a lambda term
(define (pexpr->term pe)
  (cases parsed-expr pe
    (parsed-var-expr (id) (var (varname id)))
    (parsed-pexpr_ (ppe) (cases parsed-pexpr ppe
      (parsed-lambda-expr (id body) (lam (varname id) (pexpr->term body)))
      (parsed-app-expr (e1 e2) (app (pexpr->term e1) (pexpr->term e2))))))

; Unparse a variable name. variables with subscripts use underscores
(define-match unparse-var
  ((cons n '0) n)
  ((cons n s) (string->symbol (string-append (symbol->string n) "_" (number->string s)))))

; Unparse a lambda expression (turn it back into nested lists)
(define (unparse t)
  (cases term t
    (var (v) (unparse-var v))
    (lam (v body) (list 'λ (list (unparse-var v)) (unparse body)))
    (app (t1 t2) (list (unparse t1) (unparse t2))))))

```

```

; Main term evaluation function. This simply calls teval_ with an empty stack (see below)
(define (teval t) (teval_ t '()))

; The lambda term evaluator always evaluates one term at a time. When an
; application is encountered we evaluate the left side of the application
; and push the right side to a stack so we can use it later

; When a lambda term is evaluated we first check the stack for a term
; If there is a term on the stack we pop it and, do a beta reduction,
; and evaluate the resulting expression
; If the stack is empty we simply return the lambda expression after
; evaluating its body (since we can't do beta reduction)

; If a variable term is evaluated we first check the stack for a term
; If there is a term on the stack we need to unwind the stack and
; turn it back into a series of applications. We can't do beta reduction
; without a lambda expression. This will happen when an expression's normal
; form is an application. If the stack is empty then the variable is just
; returned, it is in normal form

; I didn't come up with this technique. It is based on a lambda calculus
; evaluator by Oleg Kiselyov which I based a lambda calculus evaluator of
; my own on a while ago.

(define (teval_ t stack)
  (cases term t
    (var (name) (if (null? stack) t (unwind t stack)))
    (lam (v b)
      (if (null? stack)
          (lam v (teval b))
          (teval_ (subst b v (car stack)) (cdr stack))))
    (app (t1 t2) (teval_ t1 (cons t2 stack)))))

(define (unwind t stack)
  (if (null? stack)
      t
      (unwind (app t (teval (car stack))) (cdr stack))))

; The substitution function substitutes st for the variable v in the term t
(define (subst t v st)
  ; See if we can get off easy, if we're replacing with ourselves do nothing
  (or (cases term st (var (v_) (if (equal? v v_) t #f)) (else #f))
      (cases term t
        ; If the variable equals the one we're substituting return st, else t
        (var (x) (if (equal? x v) st t))
        ; Simply substitute both sides of the application
        (app (t1 t2) (app (subst t1 v st) (subst t2 v st)))
        ; If we're substituting for the variable bound by the lambda expression we're done
        ; (since it shadows the one we're substituting). If not things get messy...
        (lam (x body) (if (equal? x v) t
                          (match
                           ; Check if the bound variable occurs in our substitution
                           (match (occures st x)
                             ; If not we can leave the bound variable and the body alone
                             ('#f (cons x body))
                             ; If it does we have to rename the bound variable (alpha conversion)
                             ; s is the highest subscript that appears on a variable with the same
                             ; name as x in st
                             (s (let*
                                  ; The name of x, name of v, and subscript of v
                                  ((xn (car x))
                                   (vn (car v))
                                   (vs (cdr v))
                                   ; unique1 is a unique name for x within st and v
                                   (unique1 (+ (if (eqv? vn xn) (max s vs) s) 1))
                                   ; unique2 is a unique name for unique1 within the body
                                   (unique2 (match (occures body (cons xn unique1))
                                                ('#f unique1) ; not in body, we're done
                                                (s (+ (max s unique1) 1)))) ; if it is, increment the highest
                                   ; Change x's subscript to unique2 and substitute for x in the body
                                   (cons (cons xn unique2) (subst body x (var (cons xn unique2))))))
                                  ; This is the result of the above two calls. Now we can substitute in the body
                                  ; and return the new lambda expression with a possibly renamed bound variable
                                  ((cons x_ body_) (lam x_ (subst body_ v st))))))))))))))

; easier to use wrapper around occures_ (below)

```

```

(define (occures t v)
  (match (occures_ t v)
    ((cons '#f _) #f)
    ((cons '#t s) s)))

; occures looks for a variable in a term. it returns if it was found, and if it was,
; the highest subscript of all variables with that name in the term
(define (occures_ t v)
  (cases term t
    (var (v_) (cond
      ; if it is a different name we didn't find it
      ((not (eqv? (car v) (car v_))) (cons #f (cdr v)))
      ; same name but different subscripts, take the highest, but still not found
      ((not (eqv? (cdr v) (cdr v_))) (cons #f (max (cdr v) (cdr v_))))
      ; same name, found, and with the same subscript
      (else (cons #t (cdr v))))))
    (lam (x body) (if (equal? x v) (cons #f (cdr v)) (occures_ body v)))
    ; Just get the union of the two sides of the application
    (app (t1 t2)
      (let
        ((r1 (occures_ t1 v))
         (r2 (occures_ t2 v)))
        (cons (or (car r1) (car r2)) (max (cdr r1) (cdr r2))))))

; To run a lambda expression parse it, expand it, make it a term, evaluate
; it, and unparse the result
(define (run s) (unparse (teval (pexpr->term (expand (parse s))))))

; Some test data (they compute factorials in a few different ways)
(define test
  "(((lambda (true) (lambda (false) (((lambda (and) (lambda (or) (lambda (not) (lambda (xor) ((lambda (equ) ((lambda (id) (lambda (const) (((lambda (succ) (lambda (pred) (lambda (iszero) ((lambda (c0) ((lambda (c1) ((lambda (c2) ((lambda (c3) ((lambda (c4) ((lambda (c5) ((lambda (plus) ((lambda (mult) ((lambda (exp) ((lambda (y) ((lambda (fac) (fac c3)) (y (lambda (f) (lambda (x) ((iszero x) c1) ((mult x) (f (pred x))))))) (lambda (g) ((lambda (x) (g (x x))) (lambda (x) (g (x x)))))) (lambda (m) (lambda (n) ((n (mult m) c1)))) (lambda (m) (lambda (n) ((m (plus n) c0)))) (lambda (m) (lambda (n) ((m succ n)))) (succ c4))) (succ c3))) (succ c2))) (succ c1))) (succ c0))) (lambda (f) (lambda (x) x)))) (lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))) (lambda (n) (lambda (f) (lambda (x) ((n (lambda (g) (lambda (h) (h (g f)))) (const x) id)))) (lambda (n) ((n (lambda (x) false)) true)))) (lambda (x) x) (lambda (x) (lambda (y) x))) (lambda (p) (lambda (q) (not (xor p q)))))) (lambda (p) (lambda (q) ((p q) false))) (lambda (p) (lambda (q) ((p true) q))) (lambda (p) ((p false) true))) (lambda (p) (lambda (q) ((p (q false) true) q)))) (lambda (x) (lambda (y) x)) (lambda (x) (lambda (y) y)))"
)

(define test2
  "(((lambda (g) ((lambda (x) (g (x x))) (lambda (x) (g (x x)))) (lambda (f) (lambda (x) (((lambda (n) ((n (lambda (x) (lambda (x) (lambda (y) y)))) (lambda (x) (lambda (y) x))) x) ((lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))) (lambda (f) (lambda (x) x))) ((lambda (m) (lambda (n) ((m (lambda (m) (lambda (n) ((m (lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))) n)) n)) (lambda (f) (lambda (x) x))) x) (f ((lambda (n) (lambda (f) (lambda (x) (((n (lambda (g) (lambda (h) (h (g f)))) (lambda (x) (lambda (y) x)) x) (lambda (x) x)))))) (lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))))) (lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))) (lambda (f) (lambda (x) x))))"
)

(define test3
  "$ true = (lambda (x) (lambda (y) x)) $ false = (lambda (x) (lambda (y) y)) $ if-then-else = (lambda (cond) (lambda (then) (lambda (else) ((cond the n) else))) $ 0 = (lambda (f) (lambda (x) x)) $ 1 = (lambda (f) (lambda (x) (f x))) $ 2 = (lambda (f) (lambda (x) (f (f x)))) $ 3 = (lambda (f) (lambda (x) (f (f (f x))))) $ repeat = (lambda (n) (lambda (x) ((n (lambda (g) (g x))) (lambda (y) y)))) $ succ = (lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))) $ pred = (lambda (n) (((n (lambda (p) (lambda (z) ((z (succ (p true))) (p true)))) (lambda (z) ((z 0) 0))) false)) $ sum = (lambda (m) (lambda (n) (lambda (f) (lambda (x) ((m f) ((n f) x)))))) $ product = (lambda (m) (lambda (n) (lambda (f) (m (n f)))) $ isZero = (lambda (n) ((n (lambda (x) false)) true)) $ g = (lambda (f) (lambda (n) (((if-then-else (isZero n)) 1) ((product n) (f (pred n)))))) $ ycomb = (lambda (y) ((lambda (x) (y (x x))) (lambda (x) (y (x x))))) $ factorial = (ycomb g) (repeat (factorial 3)) hello"
)

```

```

;(let ((time-stamp "Time-stamp: <2001-05-10 16:12:14 dfried>"))
; (eopl:printf "3-5.scm: language with procedures ~a~%"
; (substring time-stamp 13 29))

;;;;;;;;;;;;;;;;; top level and tests ;;;;;;;;;;;;;;;;;;

(define run
  (lambda (string)
    (eval-program (scan&parse string))))

(define run-all
  (lambda ()
    (run-experiment run use-execution-outcome
      '(lang3-1 lang3-5) all-tests)))

(define run-one
  (lambda (test-name)
    (run-test run test-name)))

;; needed for testing
(define equal-external-reps? equal?)

;;;;;;;;;;;;;;;;; grammatical specification ;;;;;;;;;;;;;;;;;;

(define the-lexical-spec
  '((whitespace (whitespace) skip)
    (comment ("% " (arbno (not #\newline)))) skip)
    (identifier
      (letter (arbno (or letter digit "_" "-" "?")))
      symbol)
    (number (digit (arbno digit)) number)))

(define the-grammar
  '((program ((arbno funcdecl) expression) a-program)
    (funcdecl ("func" identifier "(" (separated-list identifier ",") ")" expression) a-funcdecl)
    (expression (number) lit-exp)
    (expression (identifier) var-exp)
    (expression
      (primitive "(" (separated-list expression ",") ")")
      primapp-exp)
    (expression
      ("if" expression "then" expression "else" expression)
      if-exp)
    (expression
      ("let" (arbno identifier "=" expression) "in" expression)
      let-exp)
    (expression
      ("(" identifier (arbno expression) ")")
      app-exp)
    (expression
      ("begin" expression (arbno ";" expression) "end")
      begin-exp)

    (primitive "+") add-prim)
    (primitive "-") subtract-prim)
    (primitive "*") mult-prim)
    (primitive "add1") incr-prim)
    (primitive "sub1") decr-prim)
    (primitive "zero?") zero-test-prim)

  ))

(sllgen:make-define-datatypes the-lexical-spec the-grammar)

(define show-the-datatypes
  (lambda () (sllgen:list-define-datatypes the-lexical-spec the-grammar)))

(define scan&parse
  (sllgen:make-string-parser the-lexical-spec the-grammar))

(define just-scan
  (sllgen:make-string-scanner the-lexical-spec the-grammar))

(define read-eval-print
  (sllgen:make-rep-loop "--> "
    (lambda (pgm) (eval-program pgm)) ; wrapped to avoid load

```

; dependency

```

(sllgen:make-stream-parser
 the-lexical-spec
 the-grammar)))

;;;;;;;;;;;;; the interpreter ;;;;;;;;;;;;;;

(define (make-func-env xs)
  (map (λ (x)
        (cases funcdecl x
              (a-funcdecl (name args body) (cons name x)))) xs))

(define eval-program
  (λ (pgm)
    (cases program pgm
      (a-program (funcdecls body)
        (eval-expression body (init-env) (make-func-env funcdecls))))))

(define eval-expression
  (λ (exp env func-env)
    (cases expression exp
      (lit-exp (datum) datum)
      (var-exp (id) (apply-env env id))
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands env func-env)))
          (apply-primitive prim args)))
      (if-exp (test-exp true-exp false-exp) ;\new4
        (if (true-value? (eval-expression test-exp env func-env))
            (eval-expression true-exp env func-env)
            (eval-expression false-exp env func-env)))
      (begin-exp (exp1 exps)
        (let loop ((acc (eval-expression exp1 env func-env))
                   (exps exps))
          (if (null? exps) acc
              (loop (eval-expression (car exps) env func-env) (cdr exps)))))
      (let-exp (ids rands body) ;\new3
        (let ((args (eval-rands rands env func-env)))
          (eval-expression body (extend-env ids args env) func-env)))
      (app-exp (name rands) ;\new7
        (let ((func (lookup name func-env))
              (args (eval-rands rands env func-env)))
          (if func
              (apply-func func args func-env)
              (eopl:error 'eval-expression
                           "Attempt to apply non-procedure ~s" name))))
      ;&
      (else (eopl:error 'eval-expression "Not here:~s" exp)
            )))

;;; Right now a prefix must appear earlier in the file.

(define eval-rands
  (λ (rands env func-env)
    (map (λ (x) (eval-rand x env func-env)) rands)))

(define eval-rand
  (λ (rand env func-env)
    (eval-expression rand env func-env)))

(define apply-primitive
  (λ (prim args)
    (cases primitive prim
      (add-prim () (+ (car args) (cadr args)))
      (subtract-prim () (- (car args) (cadr args)))
      (mult-prim () (* (car args) (cadr args)))
      (incr-prim () (+ (car args) 1))
      (decr-prim () (- (car args) 1))
      ;&
      (zero-test-prim () (if (zero? (car args)) 1 0))
      )))

(define init-env
  (λ ()
    (extend-env
     '(i v x)
     '(1 5 10)
     ())))

```

```

    (empty-env)))

;;;;;;;;;;;;;; booleans ;;;;;;;;;;;;;;

(define true-value?
  (λ (x)
    (not (zero? x))))

;;;;;;;;;;;;;; procedures ;;;;;;;;;;;;;;

(define-datatype procval procval?
  (closure
   (ids (list-of symbol?))
   (body expression?)
   (env environment?)))

(define apply-func
  (λ (func args func-env)
    (cases funcdecl func
      (a-funcdecl (name ids body)
        (eval-expression body (extend-env ids args (init-env)) func-env))))))

;;;;;;;;;;;;;; environments ;;;;;;;;;;;;;;

(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
   (syms (list-of symbol?))
   (vec vector?)           ; can use this for anything.
   (env environment?))
  )

(define empty-env
  (λ ()
    (empty-env-record)))

(define extend-env
  (λ (syms vals env)
    (extended-env-record syms (list->vector vals) env)))

(define apply-env
  (λ (env sym)
    (cases environment env
      (empty-env-record ()
        (eopl:error 'apply-env "No binding for ~s" sym))
      (extended-env-record (syms vals env)
        (let ((position (rib-find-position sym syms)))
          (if (number? position)
              (vector-ref vals position)
              (apply-env env sym)))))))

(define rib-find-position
  (λ (sym los)
    (list-find-position sym los)))

(define list-find-position
  (λ (sym los)
    (list-index (λ (syml) (eqv? syml sym)) los)))

(define list-index
  (λ (pred ls)
    (cond
      ((null? ls) #f)
      ((pred (car ls)) 0)
      (else (let ((list-index-r (list-index pred (cdr ls))))
              (if (number? list-index-r)
                  (+ list-index-r 1)
                  #f))))))

(define iota
  (λ (end)
    (let loop ((next 0))
      (if (>= next end) '()
          (cons next (loop (+ 1 next))))))

(define difference

```

```
(λ (set1 set2)
  (cond
    ((null? set1) '())
    ((memv (car set1) set2)
     (difference (cdr set1) set2))
    (else (cons (car set1) (difference (cdr set1) set2))))))
```

```

;; 3-7.scm : interp with variable assignment

;(let ((time-stamp "Time-stamp: <2001-05-10 16:13:41 dfried>"))
; (eopl:printf "3-7.scm - interp with variable assignment ~a~%"
; (substring time-stamp 13 29))

;;;;;;;;;;;;; top level and tests ;;;;;;;;;;;;;;

(define run
  (lambda (string)
    (eval-program (scan&parse string))))

(define run-all
  (lambda ()
    (run-experiment run use-execution-outcome
      '(lang3-1 lang3-5 lang3-6 lang3-7) all-tests)))

(define run-one
  (lambda (test-name)
    (run-test run test-name)))

;; needed for testing
(define equal-external-reps? equal?)

;;;;;;;;;;;;; grammatical specification ;;;;;;;;;;;;;;

(define the-lexical-spec
  '((whitespace (whitespace) skip)
    (comment ("% (arbno (not #\newline))) skip)
    (identifier
     (letter (arbno (or letter digit "_" "-" "?")))
     symbol)
    (number (digit (arbno digit)) number)))

(define the-grammar
  '((program (expression) a-program)
    (expression (number) lit-exp)
    (expression (identifier) var-exp)
    (expression
     (primitive "(" (separated-list expression ",") ")")
     primapp-exp)
    (expression
     ("if" expression "then" expression "else" expression)
     if-exp)
    (expression
     ("let" (arbno identifier "=" expression) "in" expression)
     let-exp)
    (expression
     ("proc" "(" (separated-list identifier ",") ")" expression)
     proc-exp)
    (expression
     "(" expression (arbno expression) ")")
     app-exp)
    (expression ("set" identifier "=" expression) varassign-exp) ; new for 3-7
    (expression
     ("begin" expression (arbno ";" expression) "end")
     begin-exp)
    (expression
     ("letrec"
      (arbno identifier "(" (separated-list identifier ",") ")"
       "=" expression)
      "in" expression)
     letrec-exp)

    (primitive "+") add-prim)
    (primitive "-") subtract-prim)
    (primitive "*") mult-prim)
    (primitive "add1") incr-prim)
    (primitive "sub1") decr-prim)
    (primitive "zero?") zero-test-prim)

  ))

(sllgen:make-define-datatypes the-lexical-spec the-grammar)

(define list-the-datatypes

```

```

(λ () (sllgen:list-define-datatypes the-lexical-spec the-grammar)))

(define scan&parse
  (sllgen:make-string-parser the-lexical-spec the-grammar))

(define just-scan
  (sllgen:make-string-scanner the-lexical-spec the-grammar))

;;;;;;;;;;;;; the interpreter ;;;;;;;;;;;;;;

(define eval-program
  (λ (pgm)
    (cases program pgm
      (a-program (body)
        (eval-expression body (init-env))))))

(define eval-expression ;^exp x env -> expval
  (λ (exp env)
    (cases expression exp
      (lit-exp (datum) datum)
      (var-exp (id) (apply-env env id))
      (varassign-exp (id rhs-exp) ;\new6
        (begin
          (setref!
            (apply-env-ref env id)
            (eval-expression rhs-exp env))
          1))
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands env)))
          (apply-primitive prim args)))
      (if-exp (test-exp true-exp false-exp) ;\new4
        (if (true-value? (eval-expression test-exp env))
            (eval-expression true-exp env)
            (eval-expression false-exp env)))
      (proc-exp (ids body) (closure ids body env)) ;\new1
      (let-exp (ids rands body) ;\new3
        (let ((args (eval-rands rands env)))
          (eval-expression body (extend-env ids args env))))
      (begin-exp (expl exps)
        (let loop ((acc (eval-expression expl env))
                   (exps exps))
          (if (null? exps) acc
              (loop (eval-expression (car exps) env) (cdr exps)))))
      (app-exp (rator rands) ;\new7
        (let ((proc (eval-expression rator env))
              (args (eval-rands rands env)))
          (if (procedure? proc)
              (apply proc args)
              (if (procval? proc)
                  (apply-procval proc args)
                  (eopl:error 'eval-expression
                               "Attempt to apply non-procedure ~s" proc))))
      (letrec-exp (proc-names idss bodies letrec-body) ;\new4
        (eval-expression letrec-body
          (extend-env-recursively
            proc-names idss bodies env)))
      )))

(define eval-rands
  (λ (rands env)
    (map (λ (x) (eval-rand x env)) rands)))

(define eval-rand
  (λ (rand env)
    (eval-expression rand env)))

(define apply-primitive
  (λ (prim args)
    (cases primitive prim
      (add-prim () (+ (car args) (cadr args)))
      (subtract-prim () (- (car args) (cadr args)))
      (mult-prim () (* (car args) (cadr args)))
      (incr-prim () (+ (car args) 1))
      (decr-prim () (- (car args) 1))
      (zero-test-prim () (if (zero? (car args)) 1 0))
      )))

```

```

(define (array-proc n) (make-vector n 0))
(define get-proc vector-ref)
(define (put-proc v n x) (begin (vector-set! v n x) x))
(define length-proc vector-length)

(define init-env
  (λ ()
    (extend-env
      '(i v x array get put length)
      (list 1 5 10 array-proc get-proc put-proc length-proc)
      (empty-env))))

;;;;;;;;;;;;;; booleans ;;;;;;;;;;;;;;

(define true-value?
  (λ (x)
    (not (zero? x))))

;;;;;;;;;;;;;; procedures ;;;;;;;;;;;;;;

(define-datatype procval procval?
  (closure
    (ids (list-of symbol?))
    (body expression?)
    (env environment?)))

(define apply-procval
  (λ (proc args)
    (cases procval proc
      (closure (ids body env)
        (eval-expression body (extend-env ids args env))))))

;;;;;;;;;;;;;; references ;;;;;;;;;;;;;;

(define-datatype reference reference?
  (a-ref
    (position integer?)
    (vec vector?)))

(define primitive-deref
  (λ (ref)
    (cases reference ref
      (a-ref (pos vec) (vector-ref vec pos)))))

(define primitive-setref!
  (λ (ref val)
    (cases reference ref
      (a-ref (pos vec) (vector-set! vec pos val)))))

(define deref
  (λ (ref)
    (primitive-deref ref)))

(define setref!
  (λ (ref val)
    (primitive-setref! ref val)))

;;;;;;;;;;;;;; environments ;;;;;;;;;;;;;;

(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
    (syms (list-of symbol?))
    (vec vector?) ; can use this for anything.
    (env environment?))
  )

(define empty-env
  (λ ()
    (empty-env-record)))

;;; Right now a prefix must appear earlier in the file.

(define extend-env
  (λ (syms vals env)

```

```

    (extended-env-record syms (list->vector vals) env)))

;;; Right now a prefix must appear earlier in the file.

(define apply-env
  (λ (env sym)
    (deref (apply-env-ref env sym))))

(define apply-env-ref
  (λ (env sym)
    (cases environment env
      (empty-env-record ()
        (eopl:error 'apply-env-ref "No binding for ~s" sym))
      (extended-env-record (syms vals env)
        (let ((pos (rib-find-position sym syms)))
          (if (number? pos)
              (a-ref pos vals)
              (apply-env-ref env sym)))))))

(define extend-env-recursively
  (λ (proc-names idss bodies old-env)
    (let ((len (length proc-names)))
      (let ((vec (make-vector len)))
        (let ((env (extended-env-record proc-names vec old-env)))
          (for-each
            (λ (pos ids body)
              (vector-set! vec pos (closure ids body env)))
            (iota len) idss bodies)
          env))))))

(define rib-find-position
  (λ (sym los)
    (list-find-position sym los)))

(define list-find-position
  (λ (sym los)
    (list-index (λ (syml) (eqv? syml sym)) los)))

(define list-index
  (λ (pred ls)
    (cond
      ((null? ls) #f)
      ((pred (car ls)) 0)
      (else (let ((list-index-r (list-index pred (cdr ls))))
              (if (number? list-index-r)
                  (+ list-index-r 1)
                  #f))))))

(define iota
  (λ (end)
    (let loop ((next 0))
      (if (>= next end) '()
          (cons next (loop (+ 1 next)))))))

(define difference
  (λ (set1 set2)
    (cond
      ((null? set1) '())
      ((memv (car set1) set2)
       (difference (cdr set1) set2))
      (else (cons (car set1) (difference (cdr set1) set2))))))

```

```

; Brian Alliet
; Programming Language Theory

(define lex-spec
  '((white-sp (whitespace) skip)
    (id (
      (or letter      "?" "+" "-" "*" "/" "' " ":" "!" "@" "$" "%" "^" "&" "=" "|" "_")
      (arbno (or letter digit  "?" "+" "-" "*" "/" "' " ":" "!" "@" "$" "%" "^" "&" "=" "|" "_")))
    symbol)
    (comment ("#" (arbno (not #\newline)))) skip)
    (number (digit (arbno digit)) number)
    (char ("'" (not #\') "'") string)
    (string ("\" (arbno (not #\")) "\") string)))

; The grammar is a little confusing. There are three different types of expressions so we can get
; left associative application working correctly with sllgen
; parsed-expr are all expressions that extend as far to the right as possible
; this includes application after an identifier and after a parenthesized expression
; parsed-expr2 is every expression that can appear as a component of a series of applications
; this is just identifiers, parenthesized expressions and constants
; parsed-expr3 are the constants they can appear in place of parsed-exprs or parsed-expr2s
(define grammar '(
  (parsed-expr ("\" id "->" parsed-expr) parsed-lambda-expr)
  (parsed-expr ("if" parsed-expr "then" parsed-expr "else" parsed-expr) parsed-if-expr)
  (parsed-expr ("let" "{" (arbno id "=" parsed-expr ";") "}" "in" parsed-expr) parsed-let-expr)
  (parsed-expr (id (arbno parsed-expr2)) parsed-app-expr)
  (parsed-expr ("(" parsed-expr ")") (arbno parsed-expr2)) parsed-paren-app-expr)
  (parsed-expr (parsed-expr3) parsed-expr3-expr)

  (parsed-expr2 ("(" parsed-expr ")") parsed-paren-expr2)
  (parsed-expr2 (id) parsed-id-expr2)
  (parsed-expr2 (parsed-expr3) parsed-expr3-expr2)

  (parsed-expr3 (number) parsed-const-num-expr3)
  (parsed-expr3 (string) parsed-const-string-expr3)
  (parsed-expr3 (char) parsed-const-char-expr3)
  (parsed-expr3 ("[" (separated-list parsed-expr ",") "]" ) parsed-list-expr3)
  ))

(sllgen:make-define-datatypes lex-spec grammar)
(define parse2 (sllgen:make-string-parser lex-spec grammar))

; There are several different types of values
; unit has only one (non-bottom) value
; bool, char, number are what you'd expect
; tuple is an ordered pair
; cons-cell is a cons cell in a list (this looks like a tuple but it is a different
; type so we don't have to worry about improper lists, etc)
; null is the empty list
; io-monad is a pure value representing a computation to be performed in the io monad
(define-datatype value value?
  (unit)
  (bool (b boolean?))
  (char (c char?))
  (number (n number?))
  (lam (l procedure?))
  (tuple (a thunk?) (b thunk?))
  (cons-cell (h thunk?) (t thunk?))
  (null)
  (io-monad (m io?))
  (io-ref (v vector?))
  )

; There are 4 types of io action, the standard monad bind and return operations,
; get-char, and put-char
(define-datatype io io?
  (bind-io (m thunk?) (f thunk?))
  (return-io (f thunk?))
  (put-char-io (c thunk?))
  (get-char-io)
  (new-io-ref-io (v thunk?))
  (read-io-ref-io (r thunk?))
  (write-io-ref-io (r thunk?) (v thunk?))
  )

(define type? list?)

```

```

(define (make-type bt . args) (cons bt args))
(define int-type (make-type 'int))
(define char-type (make-type 'char))
(define bool-type (make-type 'bool))
(define (lam-type a b) (make-type 'lam a b))
(define (tuple-type a b) (make-type 'tuple a b))
(define (list-type a) (make-type 'list a))

(define a-type (make-type 'a))
(define b-type (make-type 'b))

; There are 6 expression types. Constants, lambda expressions, if expressions,
; identifiers, application, and let binding. "let" here is really letrec
(define-datatype expr expr?
  (constant-expr (v value?) (t type?))
  (lam-expr (x symbol?) (body expr?))
  (if-expr (test expr?) (e1 expr?) (e2 expr?))
  (id-expr (id symbol?))
  (app-expr (e1 expr?) (e2 expr?))
  (let-expr (bindings (curry all pair?)) (e expr?))
)

; parsed-expr->expr converts a parsed expression into our real expression datatype
; (which doesn't carry all the baggage associated with the sllgen hacks)
(define (parsed-expr->expr e)
  (cases parsed-expr e
    (parsed-lambda-expr (x body) (lam-expr x (parsed-expr->expr body)))
    (parsed-if-expr (test e1 e2) (if-expr (parsed-expr->expr test) (parsed-expr->expr e1) (parse
d-expr->expr e2)))
    (parsed-let-expr (ids bodies e) (let-expr (zip ids (map parsed-expr->expr bodies)) (parsed-e
xpr->expr e)))
    (parsed-app-expr (id apps) (parsed-app-expr->expr (id-expr id) (map parsed-expr2->expr apps)
))
    (parsed-paren-app-expr (e apps) (parsed-app-expr->expr (parsed-expr->expr e) (map parsed-exp
r2->expr apps)))
    (parsed-expr3-expr (e) (parsed-expr3->expr e))))

(define (parsed-app-expr->expr e es) (if (null? es) e (foldl app-expr e es)))

(define (parsed-expr2->expr e)
  (cases parsed-expr2 e
    (parsed-paren-expr2 (e) (parsed-expr->expr e))
    (parsed-id-expr2 (id) (id-expr id))
    (parsed-expr3-expr2 (e) (parsed-expr3->expr e))))

(define (parsed-expr3->expr e)
  (cases parsed-expr3 e
    (parsed-const-num-expr3 (n) (constant-expr (number n) int-type))
    (parsed-const-char-expr3 (c) (constant-expr (char (string-ref c 1)) char-type))
    ; We convert string constants into lists of chars
    (parsed-const-string-expr3 (s)
      (constant-expr (foldr (lambda (h r) (cons-cell (non-thunked h) (non-thunked r))) (null)
        (map char (string->list (substring s 1 (- (string-length s) 1))))) (list-type char-t
ype))))
    ; And list constants into nested calls to cons
    (parsed-list-expr3 (elems) (foldr
      (lambda (h t) (app-expr (app-expr (id-expr 'cons) h) t))
      (constant-expr (null) (list-type a-type))
      (map parsed-expr->expr elems))))
  ))

(define (parse s) (parsed-expr->expr (parse2 s)))

; The environment is a simple associated list. This should be turned into some kind of
; binary tree to make it faster.
(define empty-env '())
(define (add-env k v env) (cons (cons k v) env))
(define lookup-env lookup)

; new-thunk creates a thunk. A thunk is a possibly unevaluated expressin.
; the expression is only evaluated once and only after a call to force-thunk
; We represent thunks as procedures. The first call to the procedure evaluates the
; expression and saves the result, subsequent calls will use the saved result.
(define (new-thunk f)
  (let

```

```

    ((evald #f) (realval #f))
    (λ ()
      (if evald
          realval
          (begin (set! realval (f)) (set! evald #t) realval))))))
; A thunk is forced by just calling the procedure
(define (force-thunk t) (t))
; non-thunked is used when an already evaluated expression needs to be used as a thunk
(define (non-thunked x) (λ () x))
; thunk? returns true if a value is a thunk (but it could return true for other values too)
(define thunk? procedure?)

; thunkify is just syntactic sugar for new-thunk with a lambda argument
(define-syntax thunkify
  (syntax-rules ()
    ((thunkify expr) (new-thunk (λ () expr)))))

; The following function force evaluation of thunks and returns a value as a
; specific type. This is currently where most of the typechecking is done
; but it could be done statically in the future
(define (force-num v)
  (cases value (force-thunk v)
    (number (n) n)
    (else (error "type error, non-number used as number"))))

(define (force-list v)
  (cases value (force-thunk v)
    (null () '())
    (cons-cell (h t) (cons (force-thunk h) (force-list t)))
    (else (error "type error, force-list: non-list used as list"))
  ))

(define (force-char v)
  (cases value (force-thunk v)
    (char (c) c)
    (else (error "type error, non-char used as a char"))))

(define (to-io-monad v)
  (cases value v
    (io-monad (m) m)
    (else (error "type error, non-io used as io"))))

(define (force-io-monad v) (to-io-monad (force-thunk v)))

(define (force-io-ref v)
  (cases value (force-thunk v)
    (io-ref (v) v)
    (else (error "type error, non-io-ref used as ioref"))))

(define (to-lam v)
  (cases value v
    (lam (l) l)
    (else (error "type error, non-lambda used as lambda"))))

(define (force-lam v) (to-lam (force-thunk v)))

(define (to-bool v)
  (cases value v
    (bool (b) b)
    (else (error "type error, non-bool used as bool"))))

; eval-expr evaluates an expression. eval-expr always evaluates an expression
; to weak head normal form
(define (eval-expr e env)
  (cases expr e
    (constant-expr (v t) v) ; constants are easy
    ; lambda expressions are represented by scheme procedures. when called they
    ; add their argument to the environment and evaluate themselves
    (lam-expr (x body) (lam (λ (arg) (eval-expr body (add-env x arg env))))))
    ; if expressions evaluate the test then either the first or second clause
    (if-expr (test e1 e2) (if (to-bool (eval-expr test env)) (eval-expr e1 env) (eval-expr e2 env)))
  ))

; letrec expressions are a little tricky. A new environment is created containing binding
; for everything in the letrec expression. This new environment itself is also passed to
; eval-expr for each binding. Thanks to letrec's magic behind the scenes and lazy evaluation

```

```

; this all works how it is supposed to.
(let-expr (bindings e)
  (letrec
    ((newenv (foldr
              (λ (binding env)
                (match binding
                  ((cons id expr) (add-env id (thunkify (eval-expr expr newenv)) env)))
                env
                bindings)))
      (eval-expr e newenv)
    ))
  ; Application expressions are evaluated by applying the expression on the right to
  ; the expression on the left (which must be a lambda). The argument is passed as a thunk
  ; and not evaluated yet
  (app-expr (e1 e2) ((to-lam (eval-expr e1 env)) (thunkify (eval-expr e2 env))))
  ; Identifiers are looked up in the environment and their evaluation forced
  (id-expr (id)
    (match (lookup-env id env)
      ('#f (error (string-append (symbol->string id) " not found in environment")))
      (t (force-thunk t))))
  ))

; The following are helpers for some common built-in primitives (see below)

; 0, 1 and 2 argument primitives
(define (prim0 v) (non-thunked v))
(define (prim1 v) (prim0 (lam (λ (x) (v x)))))
(define (prim2 v) (prim1 (λ (x) (lam (λ (y) (v x y))))))

; Primitives which return io actions
(define (ioprim0 v) (prim0 (io-monad v)))
(define (ioprim1 v) (prim1 (λ (x) (io-monad (v x)))))
(define (ioprim2 v) (prim2 (λ (x y) (io-monad (v x y)))))

; Arithmetic primitives
(define (arith-prim1 f) (prim1 (λ (x) (number (f (force-num x))))))
(define (arith-prim2 f) (prim2 (λ (x y) (number (f (force-num x) (force-num y))))))

; Here are all the primitives built into the language
(define initial-env `(
  (+ . ,(arith-prim2 +))
  (- . ,(arith-prim2 -))
  (* . ,(arith-prim2 *))
  (/ . ,(arith-prim2 quotient))
  (% . ,(arith-prim2 modulo))
  (negate . ,(arith-prim1 -))
  (true . ,(prim0 (bool #t)))
  (false . ,(prim0 (bool #f)))
  (newline . ,(prim0 (char #\newline)))
  (cons . ,(prim2 (λ (x y) (cons-cell x y))))
  (pair . ,(prim2 (λ (x y) (tuple x y))))
  (isnull . ,(prim1 (λ (x) (bool
    (cases value (force-thunk x)
      (null () #t)
      (cons-cell (h t) #f)
      (else (error "type error: isnull: non-list used as list"))))))))
  (head . ,(prim1 (λ (x)
    (cases value (force-thunk x)
      (null () (error "tried to take the head of the empty list"))
      (cons-cell (h t) (force-thunk h))
      (else (error "type error: head: non-list used as list"))))))))
  (tail . ,(prim1 (λ (x)
    (cases value (force-thunk x)
      (null () (error "tried to take the head of the empty list"))
      (cons-cell (h t) (force-thunk t))
      (else (error "type error: tail: non-list used as list"))))))))
  (fst . ,(prim1 (λ (x)
    (cases value (force-thunk x)
      (tuple (a b) (force-thunk a))
      (else (error "type error: non-tuple used as tuple"))))))))
  (snd . ,(prim1 (λ (x)
    (cases value (force-thunk x)
      (tuple (a b) (force-thunk b))
      (else (error "type error: non-tuple used as tuple"))))))))
  (== . ,(prim2 (λ (x y) (bool

```

```

    (cases value (force-thunk x)
      (number (n) (cases value (force-thunk y) (number (n2) (= n n2)) (else "type error in ==")))
      (else "type error in =="))))
  (==c . ,(prim2 (λ (x y) (bool
    (cases value (force-thunk x)
      (char (n) (cases value (force-thunk y) (char (n2) (eq? n n2)) (else "type error in ==")))
      (else "type error in ==")))))
  (show . ,(prim1 (λ (x) (foldr
    (λ (h r) (cons-cell (non-thunked (char h)) (non-thunked r)))
    (null)
    (string->list (value->string (force-thunk x)))))))
  (unit . ,(prim0 (unit)))
  (return . ,(ioprim1 return-io))
  (bind . ,(ioprim2 bind-io))
  (get-char . ,(ioprim0 (get-char-io)))
  (put-char . ,(ioprim1 put-char-io))
  (new-io-ref . ,(ioprim1 new-io-ref-io))
  (read-io-ref . ,(ioprim1 read-io-ref-io))
  (write-io-ref . ,(ioprim2 write-io-ref-io))
  ))

; Run runs a string containing an expression and displays the result
; (or runs the io action if an action is returned)
(define (run s) (run_ (parse s)))
(define (run_ e)
  (let
    ((v (eval-expr e initial-env)))
    (cases value v
      (io-monad (io)
        (let
          ((ret (value->string (eval-io io))))
          (begin
            (newline)
            (display ret)
            (newline))))
      (else (begin
        (display (value->string v))
        (newline)
        )))))

; Eval-io evaluates an io action. IO pctions are pure values that represent
; operations with side effects. Because the language is pure we can't have side
; effects in the language itself but we can return values that describe actions
; with side effects.
(define (eval-io m)
  (cases io m
    (return-io (t) (force-thunk t))
    (bind-io (m f) (eval-io (to-io-monad
      ((force-lam f) (non-thunked (eval-io (force-io-monad m)))))))
    (put-char-io (c) (begin (display (force-char c)) (unit)))
    (get-char-io () (let ((c (read-char)))
      (if (eof-object? c)
        (error "hit eof on getchar")
        (char c))))
    (new-io-ref-io (v) (io-ref (vector v)))
    (read-io-ref-io (r) (force-thunk (vector-ref (force-io-ref r) 0)))
    (write-io-ref-io (r v) (begin
      (vector-set! (force-io-ref r) 0 v)
      (unit)))
  ))

; Runfile runs an expression contained in a file
(define (runfile f)
  (begin
    (display "Reading...") (newline)
    (let ((s (call-with-input-file f (λ (h)
      (begin
        (let loop ((cs '()))
          (let ((c (read-char h)))
            (if (eof-object? c) (list->string (reverse cs)) (loop (cons c cs))))))))
      (begin
        (display "Parsing...") (newline)
        (let ((e (parse s)))
          (begin
            (display "Evaluating...") (newline)
            (run_ e)))))))
  ))

```

```

; value->string turns a value into a string
(define (value->string_ v rest)
  (cases value v
    (unit () (cons "<unit>" rest))
    (bool (b) (cons (if b "true" "false") rest))
    (number (n) (cons (number->string n) rest))
    (char (c) (cons (list->string (list #\' c #\')) rest))
    (tuple (a b) (cons "(" (value->string_
      (force-thunk a)
      (cons "," (value->string_
        (force-thunk b)
        (cons ")" rest))))))
    (null () (cons "[]" rest))
    (cons-cell (h t)
      (cons "[" (tail (foldr
        (lambda (x r) (cons "," (value->string_ x r))
          (cons "]" rest)
          (cons (force-thunk h) (force-list t))))))
    (lam (p) (cons "<lambda>" rest))
    (io-monad (p) (cons "<io action>" rest))
    (io-ref (r) (cons "<ioref>" rest))
  ))

(define (value->string x) (apply string-append (value->string_ x '())))

(define initial-tenv
  `((undefined . ,a-type)
    (+ . ,(lam-type int-type (lam-type int-type int-type)))
    (- . ,(lam-type int-type (lam-type int-type int-type)))
    (* . ,(lam-type int-type (lam-type int-type int-type)))
    (/ . ,(lam-type int-type (lam-type int-type int-type)))
    (true . ,bool-type)
    (false . ,bool-type)
    (== . ,(lam-type int-type (lam-type int-type bool-type)))
    (cons . ,(lam-type a-type (lam-type (list-type a-type) (list-type a-type)))
    (isnull . ,(lam-type (list-type a-type) bool-type))
  ))

(define (type-var? t) (eqv? (string-length (symbol->string (car t))) 1))

(define (unify-error a b)
  (error (string-append "cannot unify: " (show a) " and " (show b))))

(define (unify-types a b)
  (cond
    ((eq? (type-var? a) (type-var? b))
     (match (cons a b)
       ((cons (cons ab aa) (cons bb ba))
        (if (and (eq? ab bb) (eqv? (length aa) (length ba)))
            (cons ab (map unify-types aa ba))
            (unify-error a b))))
    ((type-var? a) b)
    ((type-var? b) a)))

(define (typecheck-expr e env)
  (cases expr e
    (constant-expr (v t) t)
    (if-expr (t e1 e2) (begin
      (unify-types (typecheck-expr t env) bool-type)
      (unify-types (typecheck-expr e1 env) (typecheck-expr e2 env))))
    (id-expr (id) (match (lookup-env id env)
      ('#f (error (string-append "variable: " (show id) " not in scope")))
      (t t)))
    (app-expr (e1 e2)
      (let*
        ((t1 (unify-types (typecheck-expr e1 env) (lam-type a-type a-type)))
         (t2 (unify-types (cadr t1) (typecheck-expr e2 env)))
         (caddr t1)))
        (let-expr (bindings e)
          (error "fixme")))
        (lam-expr (x body) (error "fixme")))))

```

```
; Copyright 2005 Brian Alliet
```

```
(define (main args) (unsafe-perform-io (do io:  
  (put-str-lin "Hello, World")  
  (put-str-lin (show (e maybe: '(1 2 5 10) 12)))  
  (put-str-lin (show (e list: '(1 2 5 10) 12)))  
  (put-str-lin (show (e maybe: '(2 5 10) 11)))  
  (put-str-lin (show (e list: '(2 5 10) 11)))  
  (cs <- (read-file "monad.scm"))  
  (write-file "dump.txt" cs)  
  (io:return 0)  
  )))
```

```
; This function is parameterized over any MonadPlus instance
```

```
; It returns a combination of numbers in xs whose sum is t
```

```
; Running it under the Maybe monad yields 1 or 0 results
```

```
; Running it under the List monad yields all results
```

```
(define (e d: xs t)  
  (match xs  
    ((cons x xs) (cond  
      ((= x t) (return d: (list x)))  
      ((< x t) (msum d: (list  
        ((lift-m d: (left-section cons x)) (e d: xs (- t x)))  
        ((lift-m d: (left-section cons (- x)) (e d: xs (+ t x)))  
        (e d: xs t))))  
      (else (mzero d:))))  
    (_ (mzero d:))))
```