Class 21 CSc 335 April 11 2024

2nd Interpreter of the Term

(Due to Friedman and Felleisen, the authors of TLS)

An L-expression is an S-expression which is either:
- (AND l1 l2), or
- (OR l1 l2), or
- (NOT l1), or
- an arbitrary symbol, which we call a variable

Here l1 and l2 are arbitrary L-expressions, so this is an inductive definition once we add 'and nothing else is an L-expression'

(a) Write and certify a function lexp? which checks whether an S-expression is an L-expression.

(b) Write and certify a function covered? of an L-expression lexp and a list of symbols los that tests whether all the variables in lexp are in los.

(c) For the evaluation of L-expressions we need association lists, or alists. An alist for L-expressions is a list of (variable, value) pairs. The variable component is always a symbol, and the value component is either the number 0 (for false) or 1 (for true). Write and certify a function lookup of the symbol var and the association list al, so that (lookup var al) returns the value of the first pair in al whose car is eq? to var.

(d) If the list of symbols in an alist for L-expressions contains all the variables of an L-expression lexp, then lexp is called _closed_ with respect to this alist. A closed L-expression can be evaluated essentially by substituting the values of the variables given in the alist for the variable occurrences in the L-expression. You are asked to write and certify the function value of an L-expression lexp and an alist al, which, after verifying that lexp is closed with respect to al, determines whether lexp means true or false. If lexp is not closed wrt al, then (value lexp al) should return the symbol not-covered.

(a) comprises the pre-evaluation sanity check

(b) and (c) comprise the environment subsystem

(b)
```
(define (covered? lexp los)
  (cond ((symbol? lexp) (member? lexp los))
        ((member? (op lexp) '(AND OR))
         (and (covered? (first-arg lexp) los)
              (covered? (second-arg lexp) los)))

        (else (covered? (first-arg lexp) los))))
```

This is an example of recursive descent, exploiting the inductive structure of LEXP.

(c) an example a-list might be

((x 0) (y 1) (z 1))

ie, a flat list of bindings. There's no restriction on the number of times a given variable can occur — eg

((x 0) (y 1) (z 1) (z 0))

is also an a-list. — The lookup function will ignore all but the first binding for a variable.

```
(define (lookup var alist)
  (cond ((eq? var (caar alist)) (cadar alist))
        (else (lookup var (cdr alist)))))
```

This returns the value from the first (leftmost) binding of var.

Note that (AND x y) means one thing if alist is ((x 0) (y 1)), and means something else if alist is ((x 1) (y 1)) — explains the need for the alist argument.

(d) We separate checking from evaluation — we are assuming pre for value is satisfied — ie, lexp really is in LEXP, and also that (covered? lexp (map car alist)) is true.

alist is this system's representation of environments.

```
(define (value lexp alist)
  (cond ((symbol? lexp) (lookup lexp alist))
        ((and? lexp)
           (and (value (first-arg lexp) alist)
                (value (second-arg lexp) alist)))
        ((or? lexp)
           ...
        ((not? lexp)
           ...    )))

(define (check-and-evaluate lexp alist)
  (cond ((not (lexp? lexp)) 'syntax-error)
        ((not (covered? lexp (map car alist)))
           'not-covered )
```

; pre for value guaranteed at this point.

(else (value @exp alist))))

it will be useful to notice that a list can be thought of as a stack –

list	stack
cons	push
car	top
cdr	pop

(The left end of the list is the top of the stack)

We will make use of this to implement local variables (and, generally, scope) in the TLS interpreter ⟶ ch 10 of The Little Schemer ⟶ starting next.

Note that all variables in the LEXP system are global

(historical reference: This was true of BASIC in the 1960s as well as of the first versions of FORTRAN)

;;

; For scheme, we need somewhat more complex environment structures.

;;

; to see this, let's consider some lambda forms and their associated environments

; example 1

```
((lambda (a b)
   (+ a b))
 2 3)
```

system responds to this call by extending the base environment by adding the rib.

; the expression (+ a b) is evaluated in this environment

; (a b)----|----(2 3)

vars values ; positional correspondence is assumed.

; essentially the same flat structure used for L-exps, perhaps represented as a pair of lists -- the
; first is a list of names, i.e., (a b), and the second is the list of corresponding values, (2 3)

The lookup procedure here would be to simultaneously cdr down the vars and values lists

But this simple one-rib envirnment is not always adequate

Consider:

```
; example 2

((lambda (a b)
   (+ a (* b ((lambda (a c) (+ (* a c) b))
          3 4)
       )))
 1 2)
```

system sees ((lambda (a b) ~~~) (1 2)))
and adds the rib
(a b) ——— (1 2)
Using this rib values for are read out. But then it sees another call —— so it adds a second rib, namely (a c) ——— (3 4) so that the env. now has 2 ribs and the values of a and c and b are now determined in a 2-rib environment.

```
; the expression (+ (* a c) b) is evaluated in this environment

; (a c)---|---(3 4)
;          |
; (a b)---|---(1 2)

; and can be seen to have value (+ (* 3 4) 2) = 14.  The search for a given symbol begins with t
; proceeding to the second rib only if the symbol does not occur in the first rib.  What happens
; not occur in any rib?

; when this expression has been evaluated, the outer expression (+ a (* b 14)) is evaluated in

; (a b)---|---(1 2)

; to give (+ 1 (* 2 14)) = 29

; these environment structures are usually described as 'rib-cage environments': you can see th
; sternum, suggested in our diagrams

; clearly, any interpreter is going to need to 'push' and 'pop' ribs at appropriate times.
```