Self-Definition of Programming Languages

Informal operational semantics

Investigate

- Lisp interpreter written in Lisp
- Prolog interpreter written in Prolog

Called metacircular interpreters

Self-definition of Lisp

Construct an interpreter for Scheme in Scheme.

Implement a subset of Scheme

List Operations

(car <list>)

(cdr <list>)

(cons <item> <list>)

Arithmetic Operations

(+ <e_1> <e_2>)

(– <e_1> <e_2>)

(* <e_1> <e_2>)

(/ <e_1> <e_2>)

Predicates

(null? <list>)

(equal? <s_1> <s_2>)

(atom? <s>)

Conditional

(cond (<p_1> <e_1>)
(<p_2> <e_2>)
: :
(<p_n> <e_n>))

Function Defn & Anonymous Functions

(define (<name> <formals>) <body>)

(lambda (<formals>) <body>)

(let (<var-bindings>) <body>)

Other

(quote <item>)

(display <expr>)

(newline)

Major Functions for the Interpreter

- Top-level function **micro-rep** reads an S-expr (an atom or a list), **e**valuates it, and **p**rints the result, beginning with an empty environment.
- Function **micro-eval** accepts an S-expr and environment and returns the value of the S-expr in the context of the environment.
- Function **micro-apply** accepts a function name or lambda function, a list of the results from evaluating the actual parameters, and an environment, and returns the result of applying the given function to the parameters in the given environment.

micro-rep

Prints a prompt
Reads an S-expression
Evaluates the expression
Prints the result
Calls itself with the new environment

```
(define (micro-rep env)
  (let ((prompt (display ">> ")) (s (read)))
    (if (equal? s 'quit)
        (begin (newline) (display "Goodbye") (newline))
        (cond
          ((atom? s) (begin (newline)
                            (display (micro-eval s env))
                            (newline)
                            (micro-rep env)))
          ((equal? (car s) 'define)
                   (let ((newenv
                            (updateEnv env (caadr s)
                              (list 'lambda (cdadr s) (caddr s)))))
                     (begin (newline)
                            (display (caadr s))
                            (newline)
                            (micro-rep newenv))))
          (#t  (begin (newline)
                      (display (micro-eval s env))
                      (newline)
                      (micro-rep env)))))))
```


updateEnv

```
(define
    (updateEnv env ide binding)
            (cons (list ide binding) env))
```

applyEnv

Uses the built-in function assoc to search for an identifier in an association list and return the first list entry that matches the identifier.

```
(define
    (applyEnv ide env)
            (cadr (assoc ide env)))
```

micro-eval

Deals with several forms of S-expressions:

- An atom is either a constant (#t, #f, or a numeral) or a variable, whose value is returned.
- A quoted expression is returned unevaluated.

- display evaluates its argument, displays that value, and returns the value of the expression printed.
- newline prints a carriage return.
- A conditional expression cond is handled separately since only some of its arguments need to be evaluated.
- A let expression augments the environment with the new variable bindings and executes the body of the let in this environment.

micro-apply

Processes function calls

Receives three arguments:

- A function object: an identifier bound to a function or a lambda expression
- Actual parameters after their evaluation, accomplished by mapping micro-eval over the actual parameter list
- Current environment.

micro-eval

```
(define (micro-eval s env)
    (cond
      ((atom? s)
               (cond ((equal? s #t) #t)
                     ((equal? s #f) #f)
                     ((number? s) s)
                     (else (applyEnv s env))))
      ((equal? (car s) 'quote) (cadr s))
      ((equal? (car s) 'lambda) s)
      ((equal? (car s) 'display)
               (let ((expr-value (micro-eval (cadr s) env)))
                     (display expr-value) expr-value))
      ((equal? (car s) 'newline) (begin (newline) '()))
      ((equal? (car s) 'cond) (micro-evalcond (cdr s) env))
      ((equal? (car s) 'let)
                  (micro-evallet (cddr s)
                                 (micro-let-bind (cadr s) env)))
      (else (micro-apply (car s)
                         (map
                            (lambda (x) (micro-eval x env))
                            (cdr s))
                         env))))
```

micro-evalcond

```
(define (micro-evalcond clauses env)
   (cond  ((null? clauses) #f)
          ((micro-eval (caar clauses) env)
                (micro-eval (cadar clauses) env))
          (else
              (micro-evalcond (cdr clauses) env))))
```

Example

```
(cond  ((null? L) 0)  ((atom? (car L)) 5)  (#t 8))
```


micro-evallet

```
(define (micro-evallet exprlist env)
       (if  (null? (cdr exprlist))
            (micro-eval (car exprlist) env)
            (begin (micro-eval (car exprlist) env)
                   (micro-evallet (cdr exprlist) env))))
```

micro-let-bind

```
(define (micro-let-bind pairlist env)
   (if (null? pairlist)
      env
      (updateEnv
         (micro-let-bind (cdr pairlist) env)
            (caar pairlist)
               (micro-eval (cadar pairlist) env)) )))
```

Example

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


micro-apply

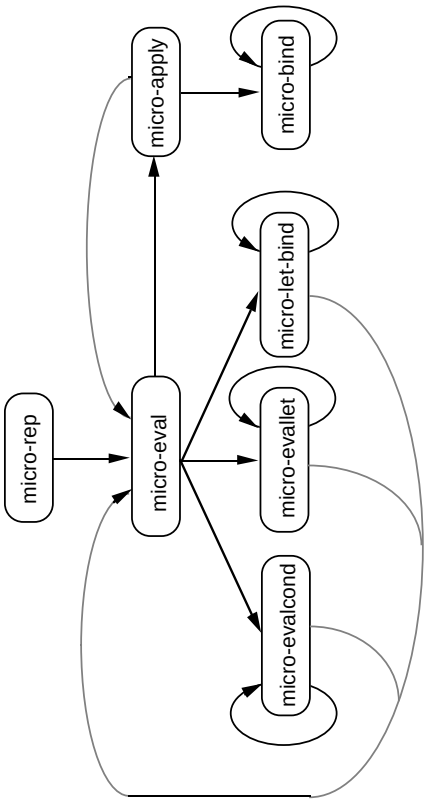
```
(define (micro-apply fn args env)
   (if (atom? fn)
      (cond  ((equal? fn 'car) (caar args))
             ((equal? fn 'cdr) (cdar args))
             ((equal? fn 'cons)
                  (cons (car args) (cadr args)))
             ((equal? fn 'atom?)
                  (atom? (car args)))
             ((equal? fn 'null?) (null? (car args)))
             ((equal? fn 'equal?)
                  (equal? (car args) (cadr args)))
             ((equal? fn '+)
                  (+ (car args) (cadr args)))
             ((equal? fn '–)
                  (– (car args) (cadr args)))
             ((equal? fn '*)
                  (* (car args) (cadr args)))
             ((equal? fn '/)
                  (/ (car args) (cadr args)))
             (else (micro-apply
                       (micro-eval fn env)
                                   args env)))
      (micro-eval
             (caddr fn)
                  (micro-bind (cadr fn) args env))))
```


micro-bind key-list

```
(define (micro-bind  key-list  value-list env)
   (if (or (null? key-list) (null? value-list))
      env
      (updateEnv
         (micro-bind (cdr key-list)
                           (cdr value-list) env)
            (car key-list)
               (car value-list) )))
```

Start Interpreter by Entering

```
(micro-rep '()).
```



Illustration

```
>>(define (first lst) (car lst))
first
```

Consider the execution of the function call:

```
>>(first (quote (a b c)))
a
```

This S-expression is passed to micro-apply with three arguments:

first	function identifier
((a b c))	eval of actual params
((first (lambda (lst) (car lst))))	current environment

Actual parameters are evaluated using map.

Actual parameter is an expression that calls the function quote, which is handled by micro-eval.

Since micro-apply does not recognize the object first, it appeals to micro-eval to evaluate first.

So micro-eval looks up a value for first in the environment and returns a lambda expression to micro-apply, which then calls itself with the following arguments:

((lambda (lst) (car lst)))	a function object
((a b c))	eval of the actual param
((first (lambda (lst) (car lst))))	the current environment

micro-eval is called with the function body as first parameter and the environment, augmented by the binding of formal parameters to actual values.

(car lst)	S-expr to be evaluated
((lst (a b c)) (first (lambda (lst) (car lst))))	the current environment

micro-eval now calls micro-apply with the first parameter as car, the evaluation of the actual parameters, and the environment.

car	a function identifier
((a b c))	evaluation of the actual param
((lst (a b c)) (first (lambda (lst) (car lst))))	the current environment

The actual parameter value is supplied when micro-eval evaluates the actual parameter lst.

The function car is something that micro-apply knows how to deal with directly; it returns the caar of the arguments, namely the atom a.

This result is returned through all the function calls back to micro-rep, which displays the result.

Strategies for Evaluating Nonlocal Variables

Scheme metacircular interpreter uses dynamic scoping.

Nonlocal variables are evaluated in the environment of the caller.

This strategy results in the *funarg* problem.

Example

Define a function twice

```
>> (define (twice func val) (func (func val)))
twice
```

Define a function double

```
>>(define (double n) (* n 2))
double
```


Call twice passing double as the function and 3 as the value

```
>> (twice double 3)
12
```

The value returned is 12, as expected.

Generalize the double function by writing a function, called times, that multiplies its argument by a preset value, called val.

```
>> (define (times x)  (* x val))
```

If val is set to 2, we expect our times function to perform just like our double function.

```
>> (let ((val 2)) (twice times 3))
27
```

Surprisingly, the value of 27 is returned rather than the value 12.

Explanation

Environmment at the time of the function call:

```
((times (lambda (x) (* x val)))
   (twice (lambda (func val) (func (func val))))
      (val 2))
```

The execution of twice adds its parameter bindings to the environment before executing the body of the function.

```
((val 3)
  (func times)
    (times (lambda (x) (* x val)))
      (twice (lambda (func val) (func (func val))))
        (val 2))
```

Difficulty

When executing the function body for times and it fetches a value for val, it fetches 3 instead of 2.

times became a tripling function, and tripling 3 twice gives 27.

Self-definition of Prolog

First handle the conjunction of goals and the chaining of goals.

A goal succeeds for one of three reasons:

- Goal is true
- Goal is conjunction with both conjuncts true
- Goal is head of a clause whose body is true.

All other goals fail.

Predefined Prolog predicate “clause” searches the user data base for a clause whose head matches the first argument; the body of the clause is returned as the second argument.

```
prove(true).

prove((Goal1, Goal2)) :-  prove(Goal1),
                          prove(Goal2).

prove(Goal) :- clause(Goal, Body),
               prove(Body).

prove(Goal) :- fail.
```

Example

memb(X,[X|Rest]).
memb(X,[Y|Rest]) :- memb(X,Rest).

Results:

```
:- prove((memb(X,[a,b,c]),memb(X,[b,c,d])))
X = b ;
X = c ;
no
:-
prove((memb(X,[a,b,c]),memb(X,[d,e,f]))).
no
:- prove(((memb(X,[a,b,c]),memb(X,[b,c,d]))
                  memb(X,[c,d,e]))).
X = c ;
no
```


Return a “proof tree” for clauses that succeed.

- Proof for true is simply true
- Proof of a conjunction of goals is a conjunction of the individual proofs
- Proof of a clause whose head is true because the body is true will be represented as “Goal<==Proof”.
- Proof tree for failure is simply fail.

```
:- op(500,xfy,<==).

prove(true, true).

prove((Goal1, Goal2),(Proof1, Proof2)) :-
                        prove(Goal1,Proof1),
                        prove(Goal2,Proof2).

prove(Goal, Goal<==Proof) :-
                        clause(Goal, Body),
                        prove(Body, Proof).

prove(Goal,fail) :- fail.
```


Example

```
:- prove((memb(X,[a,b,c]),
          memb(X,[b,c,d])),Proof).
X = b
Proof = memb(b,[a,b,c])<==memb(b,[b,c])
   <==true, memb(b,[b,c,d])<==true
:- prove((memb(X,[a,b,c]),
          memb(X,[d,e,f])), Proof).
no
:- prove(((memb(X,[a,b,c]),
             memb(X,[b,c,d])),
                memb(X,[c,d,e])), Proof
X = c
Proof =
   (memb(c,[a,b,c])<==memb(c,[b,c])
   <==memb(c,[c])<==true,
   memb(c,[b,c,d])<==memb(c,[c,d])
   <==true), memb(c,[c,d,e])<==true
```


Add a trace facility to show each step in a proof, whether it succeeds or fails.

Return to the first version of the program and add a tracing facility.

Indent our trace by one space each time we chain a rule from a goal to a body.

Print “Call: ” and the goal before the application of a user-defined rule.

Print “Exit: ” and the goal if we exit from the body of the goal successfully.

A subsequent goal may fail so we have to backtrack and retry a goal that previously succeeded.

Add a predicate retry that is true the first time it is called but prints “Retry: ”, the goal, and fails on subsequent calls.

Print “Fail: ” and the goal when a goal fails.

New Meta-interpreter

```
prove(Goal) :-  prove(Goal, 0).

prove(true, _).

prove((Goal1, Goal2), Level) :-
                    prove(Goal1, Level),
                    prove(Goal2, Level).

prove(Goal, Level) :-  tab(Level), write('Call: '),
                       write(Goal), nl,
                       clause(Goal, Body),
                       NewLevel is Level + 2,
                       prove(Body, NewLevel),
                       tab(Level), write('Exit: '),
                       write(Goal), nl,
                       retry(Goal, Level).

prove(Goal, Level) :-  tab(Level), write('Fail: '),
                       write(Goal), nl, fail.

retry(Goal, Level) :-  true ;
                       tab(Level),
                       write('Retry: '),
                       write(Goal), nl,
                       fail.
```


First Test

```
prove((memb(X,[a,b,c]),
       memb(X,[b,c,d]))).
```

```
Call: memb(_483,[a,b,c])
Exit: memb(a,[a,b,c])
Call: memb(a,[b,c,d])
  Call: memb(a,[c,d])
    Call: memb(a,[d])
      Call: memb(a,[])
      Fail: memb(a,[])
    Fail: memb(a,[d])
  Fail: memb(a,[c,d])
Fail: memb(a,[b,c,d])
Retry: memb(a,[a,b,c])
  Call: memb(_483,[b,c])
  Exit: memb(b,[b,c])
Exit: memb(b,[a,b,c])
Call: memb(b,[b,c,d])
Exit: memb(b,[b,c,d])
```


Second Test

```
prove((memb(X,[a,b,c]),
       memb(X,[d,e,f]))).
```

```
Call: memb(_483,[a,b,c])
Exit: memb(a,[a,b,c])
Call: memb(a,[d,e,f])
  Call: memb(a,[e,f])
    Call: memb(a,[f])
      Call: memb(a,[])
      Fail: memb(a,[])
    Fail: memb(a,[f])
  Fail: memb(a,[e,f])
Fail: memb(a,[d,e,f])
Retry: memb(a,[a,b,c])
  Call: memb(_483,[b,c])
  Exit: memb(b,[b,c])
Exit: memb(b,[a,b,c])
Call: memb(b,[d,e,f])
  Call: memb(b,[e,f])
    Call: memb(b,[f])
      Call: memb(b,[])
      Fail: memb(b,[])
    Fail: memb(b,[f])
  Fail: memb(b,[e,f])
Fail: memb(b,[d,e,f])
Retry: memb(b,[a,b,c])
```



```
  Retry: memb(b,[b,c])
    Call: memb(_483,[c])
    Exit: memb(c,[c])
  Exit: memb(c,[b,c])
Exit: memb(c,[a,b,c])
Call: memb(c,[d,e,f])
  Call: memb(c,[e,f])
    Call: memb(c,[f])
      Call: memb(c,[])
      Fail: memb(c,[])
    Fail: memb(c,[f])
  Fail: memb(c,[e,f])
Fail: memb(c,[d,e,f])
Retry: memb(c,[a,b,c])
  Retry: memb(c,[b,c])
    Retry: memb(c,[c])
      Call: memb(_483,[])
      Fail: memb(_483,[])
    Fail: memb(_483,[c])
  Fail: memb(_483,[b,c])
Fail: memb(_483,[a,b,c])
```

Third Test

```
prove(((memb(X,[a,b,c]),
        memb(X,[b,c,d])),
          memb(X,[c,d,e]))).
```

```
Call: memb(_486,[a,b,c])
Exit: memb(a,[a,b,c])
Call: memb(a,[b,c,d])
  Call: memb(a,[c,d])
    Call: memb(a,[d])
      Call: memb(a,[])
      Fail: memb(a,[])
    Fail: memb(a,[d])
  Fail: memb(a,[c,d])
Fail: memb(a,[b,c,d])
Retry: memb(a,[a,b,c])
  Call: memb(_486,[b,c])
  Exit: memb(b,[b,c])
Exit: memb(b,[a,b,c])
Call: memb(b,[b,c,d])
Exit: memb(b,[b,c,d])
Call: memb(b,[c,d,e])
  Call: memb(b,[d,e])
    Call: memb(b,[e])
      Call: memb(b,[])
      Fail: memb(b,[])
    Fail: memb(b,[e])
```



```
  Fail: memb(b,[d,e])
Fail: memb(b,[c,d,e])
Retry: memb(b,[b,c,d])
  Call: memb(b,[c,d])
    Call: memb(b,[d])
      Call: memb(b,[])
      Fail: memb(b,[])
    Fail: memb(b,[d])
  Fail: memb(b,[c,d])
Fail: memb(b,[b,c,d])
Retry: memb(b,[a,b,c])
  Retry: memb(b,[b,c])
    Call: memb(_486,[c])
    Exit: memb(c,[c])
  Exit: memb(c,[b,c])
Exit: memb(c,[a,b,c])
Call: memb(c,[b,c,d])
  Call: memb(c,[c,d])
  Exit: memb(c,[c,d])
Exit: memb(c,[b,c,d])
Call: memb(c,[c,d,e])
Exit: memb(c,[c,d,e])
```

Prolog interpreter written in Prolog does not explicitly implement the built-in backtracking of Prolog or show the unification process.

Trace facility allows us to follow the backtracking, but does not illustrate its implementation.