Vrije Universiteit Brussel

Software Languages Lab

Programming Language Engineering
Master of Computer Science

Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

Section 1: Interpreters

Theo D'Hondt
Software Languages Lab

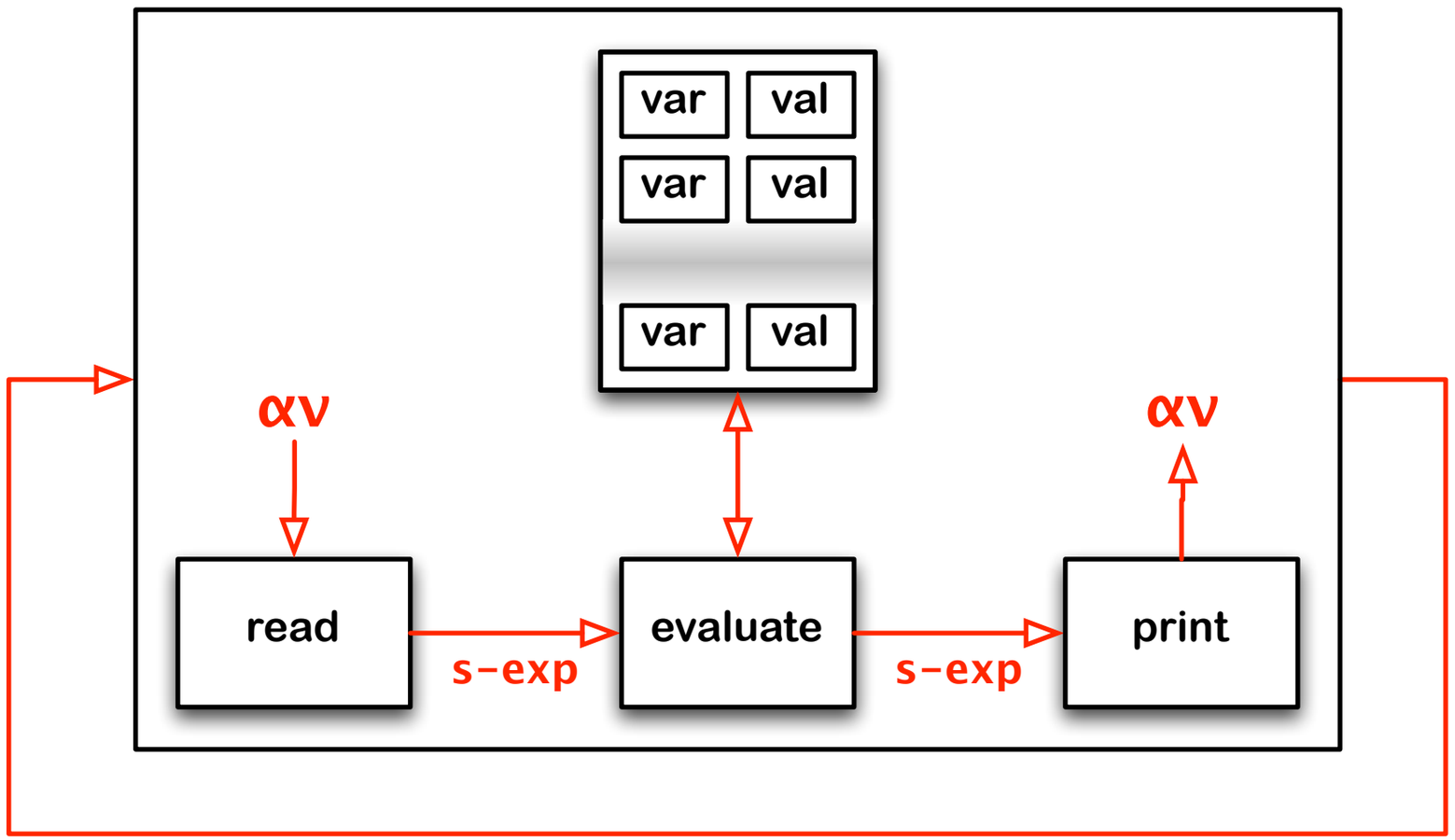
"...programs called interpreters provide the most direct, executable expression of program semantics..."

Slip: Lisp in 100 lines

```
(begin
  (define environment '())
  (define (loop output)
    (define rollback environment)
    (define (error message qualifier)
      (display message)
      (set! environment rollback)
      (loop qualifier))
    (define (bind-variable variable value)
      (define binding (cons variable value))
      (set! environment (cons binding environment)))
    (define (bind-parameters parameters arguments)
      (for-each bind-variable parameters arguments))
    (define (evaluate-sequence expressions)
      (define head (car expressions))
      (define tail (cdr expressions))
      (if (null? tail)
          (evaluate head)
          (evaluate-sequence tail)))
    (define (make-procedure parameters expression)
      (define lexical-scope environment)
      (lambda arguments
        (define dynamic-scope environment)
        (set! environment lexical-scope)
        (bind-parameters parameters arguments)
        (let ((value (evaluate expression)))
          (set! environment dynamic-scope)
          value)))
    (define (evaluate-application operator)
      (lambda operands
        (apply (evaluate operator) (map evaluate operands))))
    (define (evaluate-begin . expressions)
      (evaluate-sequence expressions))
    (define (evaluate-define variable expression)
      (define binding (cons variable '()))
      (set! environment (cons binding environment))
      (let ((value (evaluate expression)))
        (set-cdr! binding value)
        value))
    (define (evaluate-if predicate consequent alternative)
      (define boolean (evaluate predicate))
      (if (eq? boolean #f)
          (evaluate alternative)
          (evaluate consequent)))
    (define (evaluate-lambda parameters expression)
      (make-procedure parameters expression))
    (define (evaluate-set! variable expression)
      (define binding (assoc variable environment))
      (if binding
          (let ((value (evaluate expression)))
            (set-cdr! binding value)
            value)
          (error "inaccessible variable: " variable)))
    (define (evaluate-variable variable)
      (define binding (assoc variable environment))
      (if binding
          (cdr binding)
          (evaluate variable (interaction-environment))))
    (define (evaluate expression)
      (cond
        ((symbol? expression)
         (evaluate-variable expression))
        ((pair? expression)
         (let ((operator (car expression))
               (operands (cdr expression)))
           (apply
             (case operator
               ((begin)  evaluate-begin )
               ((define) evaluate-define)
               ((if)     evaluate-if    )
               ((lambda) evaluate-lambda)
               ((set!)   evaluate-set!  )
               (else     (evaluate-application operator))) operands)))
        (else
          expression)));
    (display output)
    (newline)
    (display ">>>")
    (loop (evaluate (read))))
  (loop "Slip version 0"))
```

Actually... 80 lines!

Slip$_0$ read-eval-print loop



Slip$_0$ read-eval-print loop (cont'd)

```
(define environment '())
(define (loop output)


  (display output)
  (newline)
  (display ">>>")
  (loop (evaluate (read))))

(loop "Slip version 0"))
Slip version 0
>>>(begin
     (define (factorial n continue)
       (define (continuation p)
         (continue (* n p)))
       (if (> n 1)
         (factorial (- n 1) continuation)
         (continue 1)))
     (factorial 10 display))
3628800
>>>
```

Slip$_0$ grammar

```
expression   ::= computation | lambda | variable |
                 literal | null
computation  ::= definition | assignment | sequence |
                 conditional | application
definition   ::= (define variable expression)
assignment   ::= (set! variable expression)
sequence     ::= (begin expression+)
conditional  ::= (if expression expression expression)
application  ::= (expression+)
lambda       ::= (lambda (variable*) expression)
variable     ::= symbol
literal      ::= number | character | string | #t | #f
null         ::= ()
```

"Growing a Language" by Guy Steele
keynote at the 1998 ACM OOPSLA conference

Slip$_0$ evaluation

```
(begin
  (define environment '())
  (define (loop output)
    (define rollback environment)
    (define (error message qualifier)
      (display message)
      (set! environment rollback)
      (loop qualifier))


    (define (evaluate expression)
      (cond
        ((symbol? expression)


        ((pair? expression)


        (else
          expression)))

    (display output)
    (newline)
    (display ">>>")
    (loop (evaluate (read)) environment))

  (loop "Slip version 0"))
```

Slip$_0$ evaluator

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((set!)   evaluate-set!  )
           (else     (evaluate-application operator))) operands)))
    (else
      expression)))
```

Evaluator: variables

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((set!)   evaluate-set!  )
           (else     (evaluate-application operator))) operands)))
    (else
      expression)))
```

Evaluator: variables

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((set!)   evaluate-set!  )
           (else     (evaluate-application operator))) operands)))
```

```
(define (evaluate-variable variable)
  (define binding (assoc variable environment))
    (if binding
      (cdr binding)
      (eval variable (interaction-environment))))
```

Evaluator: variables

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((set!)   evaluate-set!  )
           (else     (evaluate-application operator))) operands)))
```

```
(define (evaluate-variable variable)
  (define binding (assoc variable environment))
    (if binding
      (cdr binding)
      (eval variable (interaction-environment))))
```

reflective access to meta-variables

Evaluator: literals

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((set!)   evaluate-set!  )
           (else     (evaluate-application operator))) operands)))
    (else
      expression)))
```

Evaluator: literals

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((set!)   evaluate-set!  )
           (else     (evaluate-application operator))) operands)))
    (else
     expression)))
```

Evaluator: forms

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((set!)   evaluate-set!  )
           (else     (evaluate-application operator))) operands)))
    (else
      expression)))
```

Evaluation functions:

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: define

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: define

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
```

```
(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))
  (let ((value (evaluate expression)))
    (set-cdr! binding value)
    value))
```

Evaluation functions: define

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
```

```
(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))
  (let ((value (evaluate expression)))
    (set-cdr! binding value)
    value))
```

support for recursive functions

Evaluation functions: set!

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: set!

```
(define (evaluate-application operator)
  (lambda operands
```

```
(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
  (if binding
    (let ((value (evaluate expression)))
      (set-cdr! binding value)
      value)
    (error "inaccessible variable: " variable)))
```

```
(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: if

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: if

```
(define (evaluate-application operator)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

```
(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (if (eq? boolean #f)
    (evaluate alternative)
    (evaluate consequent)))
```

Evaluation functions: begin

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: begin

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (let ((value (evaluate head)))
    (if (null? tail)
      value
      (evaluate-sequence tail))))
```

```
  (define binding (assoc variable environment))
```

Evaluation functions: begin

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (let ((value (evaluate head)))
    (if (null? tail)
      value
      (evaluate-sequence tail))))
```

```
  (define binding (assoc variable environment))
```

no tail recursion ...

Evaluation functions: lambda

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: lambda

```
(define (make-procedure parameters expression)
  (define lexical-scope environment)
  (lambda arguments
    (define dynamic-scope environment)
    (set! environment lexical-scope)
    (bind-parameters parameters arguments)
    (let ((value (evaluate expression)))
      (set! environment dynamic-scope)
      value)))
```

```
(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: lambda

```
(define (make-procedure parameters expression)
  (define lexical-scope environment)
  (lambda arguments
    (define dynamic-scope environment)
    (set! environment lexical-scope)
    (bind-parameters parameters arguments)
    (let ((value (evaluate expression)))
      (set! environment dynamic-scope)
      value)))
```

```
(define (evaluate-define variable expression)
  (define binding (cons variable '())))
```

```
(define (bind-variable variable value)
  (define binding (cons variable value))
  (set! environment (cons binding environment)))

(define (bind-parameters parameters arguments)
  (for-each bind-variable parameters arguments))
```

```
(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: static scope

```
(define (make-procedure parameters expression)
  (define lexical-scope environment)
  (lambda arguments
    (define dynamic-scope environment)
    (set! environment lexical-scope)
    (bind-parameters parameters arguments)
    (let ((value (evaluate expression)))
      (set! environment dynamic-scope)
      value)))
```

① save defining environment outside the procedure

Evaluation functions: static scope

```
(define (make-procedure parameters expression)
  (define lexical-scope environment)
  (lambda arguments
    (define dynamic-scope environment)
    (set! environment lexical-scope)
    (bind-parameters parameters arguments)
    (let ((value (evaluate expression)))
      (set! environment dynamic-scope)
      value)))
```

② save calling environment inside the procedure

Evaluation functions: static scope

```
(define (make-procedure parameters expression)
  (define lexical-scope environment)
  (lambda arguments
    (define dynamic-scope environment)
    (set! environment lexical-scope)
    (bind-parameters parameters arguments)
    (let ((value (evaluate expression)))
      (set! environment dynamic-scope)
      value)))
```

③ switch to the defining environment

Evaluation functions: static scope

```
(define (make-procedure parameters expression)
  (define lexical-scope environment)
  (lambda arguments
    (define dynamic-scope environment)
    (set! environment lexical-scope)
    (bind-parameters parameters arguments)
    (let ((value (evaluate expression)))
      (set! environment dynamic-scope)
      value)))
```

④ restore calling environment

Evaluation functions: apply

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment))

(define (evaluate-if predicate consequent alternative)
  (define boolean (evaluate predicate))
  (evaluate (if (eq? boolean #f)

(define (evaluate-lambda parameters expression)
  (make-procedure parameters expression))

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
```

Evaluation functions: apply

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))

(define (evaluate-define variable expression)
  (define binding (cons variable '()))
  (set! environment (cons binding environment)))
```

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))
```

```
  (define (evaluate-lambda parameters expression)
    (make-procedure parameters expression))

  (define (evaluate-set! variable expression)
    (define value (evaluate expression))
    (define binding (assoc variable environment))
```

Extensions: getting some practice

```
expression     ::=  computation | lambda | quote |
                    variable | literal | null
computation    ::=  definition | assignment | sequence |
                    conditional | iteration | application
definition     ::=  (define variable expression)
definition     ::=  (define pattern expression+)
pattern        ::=  (variable+)
pattern        ::=  (variable+ . variable)
assignment     ::=  (set! variable expression)
sequence       ::=  (begin expression+)
conditional    ::=  (if expression expression expression)
conditional    ::=  (if expression expression)
iteration      ::=  (while expression expression+)
application    ::=  (expression+)
lambda         ::=  (lambda (variable*) expression+)
lambda         ::=  (lambda (variable+ . variable) expression+)
lambda         ::=  (lambda variable expression+)
quote          ::=  ' s-expression
variable       ::=  symbol
literal        ::=  number | character | string | #t | #f
null           ::=  ()
```

Extensions: getting some practice

```
expression    ::=  computation | lambda | quote |
                   variable | literal | null
computation   ::=  definition | assignment | sequence |
                   conditional | iteration | application
definition    ::=  (define variable expression)
definition    ::=  (define pattern expression+)
pattern       ::=  (variable+)
pattern       ::=  (variable+ . variable)
assignment    ::=  (set! variable expression)
sequence      ::=  (begin expression+)
conditional   ::=  (if expression expression expression)
conditional   ::=  (if expression expression)
iteration     ::=  (while expression expression+)
application   ::=  (expression+)
lambda        ::=  (lambda (variable*) expression+)
lambda        ::=  (lambda (variable+ . variable) expression+)
lambda        ::=  (lambda variable expression+)
quote         ::=  ' s-expression
variable      ::=  symbol
literal       ::=  number | character | string | #t | #f
null          ::=  ()
```

single-branch conditional

Extensions: getting some practice

```
expression     ::=  computation | lambda | quote |
                    variable | literal | null
computation    ::=  definition | assignment | sequence |
                    conditional | iteration | application
definition     ::=  (define variable expression)
definition     ::=  (define pattern expression+)
pattern        ::=  (variable+)
pattern        ::=  (variable+ . variable)
assignment     ::=  (set! variable expression)
sequence       ::=  (begin expression+)
conditional    ::=  (if expression expression expression)
conditional    ::=  (if expression expression)
iteration      ::=  (while expression expression+)
application    ::=  (expression+)
lambda         ::=  (lambda (variable*) expression+)
lambda         ::=  (lambda (variable+ . variable) expression+)
lambda         ::=  (lambda variable expression+)
quote          ::=  ' s-expression
variable       ::=  symbol
literal        ::=  number | character | string | #t | #f
null           ::=  ()
```

multi-expression function bodies

Extensions: getting some practice

```
expression    ::=  computation | lambda | quote |
                   variable | literal | null
computation   ::=  definition | assignment | sequence |
                   conditional | iteration | application
definition    ::=  (define variable expression)
definition    ::=  (define pattern expression+)
pattern       ::=  (variable+)
pattern       ::=  (variable+ . variable)
assignment    ::=  (set! variable expression)
sequence      ::=  (begin expression+)
conditional   ::=  (if expression expression expression)
conditional   ::=  (if expression expression)
iteration     ::=  (while expression expression+)
application   ::=  (expression+)
lambda        ::=  (lambda (variable*) expression+)
lambda        ::=  (lambda (variable+ . variable) expression+)
lambda        ::=  (lambda variable expression+)
quote         ::=  ' s-expression
variable      ::=  symbol
literal       ::=  number | character | string | #t | #f
null          ::=  ()
```

quoted expressions

Extensions: getting some practice

```
expression      ::=  computation | lambda | quote |
                     variable | literal | null
computation     ::=  definition | assignment | sequence |
                     conditional | iteration | application
definition      ::=  (define variable expression)
definition      ::=  (define pattern expression+)
pattern         ::=  (variable+)
pattern         ::=  (variable+ . variable)
assignment      ::=  (set! variable expression)
sequence        ::=  (begin expression+)
conditional     ::=  (if expression expression expression)
conditional     ::=  (if expression expression)
iteration       ::=  (while expression expression+)
application     ::=  (expression+)
lambda          ::=  (lambda (variable*) expression+)
lambda          ::=  (lambda (variable+ . variable) expression+)
lambda          ::=  (lambda variable expression)
quote           ::=  ' s-expression
variable        ::=  symbol
literal         ::=  number | character | string | #t | #f
null            ::=  ()
```

function defines

Extensions: getting some practice

```
expression    ::=  computation | lambda | quote |
                   variable | literal | null
computation   ::=  definition | assignment | sequence |
                   conditional | iteration | application
definition    ::=  (define variable expression)
definition    ::=  (define pattern expression+)
pattern       ::=  (variable+)
pattern       ::=  (variable+ . variable)
assignment    ::=  (set! variable expression)
sequence      ::=  (begin expression+)
conditional   ::=  (if expression expression expression)
conditional   ::=  (if expression expression)
iteration     ::=  (while expression expression+)
application   ::=  (expression+)
lambda        ::=  (lambda (variable*) expression)
lambda        ::=  (lambda (variable+ . variable) expression+)
lambda        ::=  (lambda variable expression+)
quote         ::=  ' s-expression
variable      ::=  symbol
literal       ::=  number | character | string | #t | #f
null          ::=  ()
```

variable arity functions

Extensions: getting some practice

```
expression     ::=  computation | lambda | quote |
                    variable | literal | null
computation    ::=  definition | assignment | sequence |
                    conditional | iteration | application
definition     ::=  (define variable expression)
definition     ::=  (define pattern expression+)
pattern        ::=  (variable+)
pattern        ::=  (variable+ . variable)
assignment     ::=  (set! variable expression)
sequence       ::=  (begin expression+)
conditional    ::=  (if expression expression expression)
conditional    ::=  (if expression expression)
iteration      ::=  (while expression expression+)
application    ::=  (expression+)
lambda         ::=  (lambda (variable*) expression+)
lambda         ::=  (lambda (variable+ . variable) expression+)
lambda         ::=  (lambda variable expression+)
quote          ::=  ' s-expression
variable       ::=  symbol
literal        ::=  number | character | string | #t | #f
null           ::=  ()
```

iterations

Extension: dual-branch conditional

```
(define (evaluate-if predicate consequent . alternative)
  (define boolean (evaluate predicate))
  (if (eq? boolean #f)
    (if (null? alternative)
      '()
      (evaluate (car alternative)))
    (evaluate consequent)))
```

Extension: multi-expression function bodies

```
(define (make-procedure parameters expressions)
  (define lexical-environment environment)
  (lambda arguments
    (define dynamic-environment environment)
    (set! environment lexical-environment)
    (bind-parameters parameters arguments)
    (let ((value (evaluate-sequence expressions)))
      (set! environment dynamic-environment)
      value)))
```

```
(define (evaluate-define pattern . expressions)
  (define binding (cons pattern '()))
  (set! environment (cons binding environment))
```

```
(define (evaluate-lambda parameters . expressions)
  (make-procedure parameters expressions))
```

Extension: quoted expressions

```
(define (evaluate-quote expression)
  expression)
```

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((quote)  evaluate-quote )
           ((set!)   evaluate-set!  )
           (else     (evaluate-application operator))) operands)))
    (else
      expression)))
```

Extension: quoted expressions

```
(define (evaluate-quote expression)
  expression)
```

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((quote)  evaluate-quote )
           ((set!)   evaluate-set!  )
           (else     (evaluate-application operator))) oper
    (else
      expression)))
```

#$@*!

```
(define (f t)
  (define z '(1 2 3))
  (if t (set-car! z 9))
  z)

(f #t)

(f #f)
```

Using quotes

```
Slip version 1
>>>(begin
  (define counter
    (lambda (count)
      (begin
        (define self
          (lambda (message)
            (if (string=? message "+")
                (begin
                  (set! count (+ count 1))
                  self)
                (if (string=? message "-")
                    (begin
                      (set! count (- count 1))
                      self)
                    (if (string=? message "?")
                       count
                       "error")))))
          self)))
      (define c (counter 10))
      ((((c "+") "+") "-") "?"))
11
>>>
```

Extension: function defines

```
(define (evaluate-define pattern . expressions)
  (define binding (cons pattern '()))
  (set! environment (cons binding environment))
  (if (symbol? pattern)
    (let ((value (evaluate (car expressions))))
      (set-cdr! binding value)
      value)
    (let ((procedure (make-procedure (cdr pattern) expressions)))
      (set-car! binding (car pattern))
      (set-cdr! binding procedure)
      procedure)))
```

Extension: function defines

```
(define (evaluate-define pattern . expressions)
  (define binding (cons pattern '()))
  (set! environment (cons binding environment))
  (if (symbol? pattern)
    (let ((value (evaluate (car expressions))))
      (set-cdr! binding value)
      value)
    (let ((procedure (make-procedure (cdr pattern) expressions)))
      (set-car! binding (car pattern))
      (set-cdr! binding procedure)
```

```
(define (evaluate-define pattern . expressions)
  (if (symbol? pattern)
    (let* ((value (evaluate (car expressions)))
           (binding (cons pattern value)))
      (set! environment (cons binding environment))
      value)
    (let* ((binding (cons (car pattern) '())))
      (set! environment (cons binding environment))
      (let ((procedure (make-procedure (cdr pattern) expressions)))
        (set-cdr! binding procedure)
        procedure))))
```

recursive support for function defines only

Extension: variable arity functions

```
(define (bind-variable variable value)
  (define binding (cons variable value))
  (set! environment (cons binding environment)))

(define (bind-parameters parameters arguments)
  (if (symbol? parameters)
    (bind-variable parameters arguments))
  (if (pair? parameters)
    (let
      ((variable (car parameters))
       (value    (car arguments )))
      (bind-variable variable value)
      (bind-parameters (cdr parameters) (cdr arguments)))))
```

Using variable arity

```
Slip version 2
>>>(begin
  (define empty   0)
  (define full    1)
  (define push    2)
  (define pop     3)

  (define (Stack n)
    (define stack (make-vector n))
    (define top -1)

    (define (empty)
      (< top 0))

    (define (full)
      (>= top n))

    (define (push item)
      (set! top (+ top 1))
      (vector-set! stack top item)
      ())

    (define (pop)
      (define item (vector-ref stack top))
      (set! top (- top 1))
      item)

    (define (self message . arguments)
      (define methods (vector empty full push pop))
      (apply (vector-ref methods message) arguments))

    self))
<procedure>
```

```
>>(begin
(define S (Stack 10))
  (define T (Stack 20))
  (if (S full)
    (display 'Overflow)
    (S push 123))
  (T push 456)
  (if (S empty)
    (display 'Underflow)
    (S pop))
  (display (T pop))
  (newline)
  (if (S empty)
    (display 'Underflow)
    (S pop)))456
Underflow<unspecified>
>>>
```

Using variable arity (cont'd)

```
Slip version 2
>>>(begin
  (define empty   0)
  (define full    1)
  (define push    2)
  (define pop     3)
  (define protect 4)

  (define (Stack n)
    (define stack (make-vector n))
    (define top -1)

    (define (empty)
      (< top 0))

    (define (full)
      (>= top n))

    (define (push item)
      (set! top (+ top 1))
      (vector-set! stack top item)
      ())

    (define (pop)
      (define item (vector-ref stack top))
      (set! top (- top 1))
      item)
```

```
    (define (protect)

      (define (p-push item)
        (if (full)
            (display 'Overflow)
            (push item)))

      (define (p-pop)
        (if (empty)
            (display 'Underflow)
            (pop)))

      (define (self message . arguments)
        (define methods (vector empty full p-push p-pop protect))
        (apply (vector-ref methods message) arguments))

      self)

    (define (self message . arguments)
      (define methods (vector empty full push pop protect))
      (apply (vector-ref methods message) arguments))

    self))
<procedure>
>>>(begin
 (define S (Stack 10))
  (define T (Stack 20))
  (S push 123)
  (display (S pop))
  (newline)
  (define R (S protect))
  (R push 1)
  (R pop)
  (R pop))
123
Underflow<unspecified>
>>>
```

Extension: iteration

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((quote)  evaluate-quote )
           ((set!)   evaluate-set!  )
           ((while)  evaluate-while )
           (else     (evaluate-application operator))) operands)))
    (else
      expression)))
```

Extension: iteration

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
```

```
  (define (evaluate-while predicate . expressions)
    (define (iterate value)
      (define boolean (evaluate predicate))
      (if (eq? boolean #f)
        value
        (iterate (evaluate-sequence expressions))))
    (iterate '()))
```

```
          ((lambda) evaluate-lambda)
          ((quote)  evaluate-quote )
          ((set!)   evaluate-set!  )
          ((while)  evaluate-while )
          (else     (evaluate-application operator))) operands)))
   (else
     expression)))
```

this is tail-recursive ...

Extension: iteration

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (define (evaluate-while predicate . expressions)
       (define (iterate value)
         (define boolean (evaluate predicate))
         (if (eq? boolean #f)
             value
             (iterat
       (iterate '(
                  ((l
                  ((q
                  ((s
                  ((w
                  (el
     (else
       expression)))
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (if (null? tail)
    (evaluate head)
    (begin
      (evaluate head)
      (evaluate-sequence tail))))
```

...provided we do this

Using iteration

```
Slip version 3
>>>(begin
  (define (QuickSort V Low High)
    (define Left Low)
    (define Right High)
    (define Pivot (vector-ref V (quotient (+ Left Right) 2)))
    (define Save 0)
    (while (< Left Right)
      (while (< (vector-ref V Left) Pivot)
        (set! Left (+ Left 1)))
      (while (> (vector-ref V Right) Pivot)
        (set! Right (- Right 1)))
      (if (<= Left Right)
        (begin
          (set! Save (vector-ref V Left))
          (vector-set! V Left (vector-ref V Right))
          (vector-set! V Right Save)
          (set! Left (+ Left 1))
          (set! Right (- Right 1)))))
    (if (< Low Right)
      (QuickSort V Low Right))
    (if (> High Left)
      (QuickSort V Left High)))
  (define V (make-vector 100 0))
  (define Low 0)
  (define High (- (vector-length V) 1))
  (define x 0)
  (define y 1)
  (while (<= x High)
    (vector-set! V x y)
    (set! x (+ x 1))
    (set! y (remainder (+ y 4253171) 1235711)))
  (QuickSort V Low High)
  (vector-ref V High))
1209460
>>>
```

Slip in Slip

```
(begin
  (define circularity-level (+ circularity-level 1))
  (define meta-level-eval eval)
  (define eval ())

  (define environment ())
  (define (loop output)
    (define rollback environment)
    (define (evaluate expression)

      (define (error message qualifier)
        (display message)
        (set! environment rollback)
        (loop qualifier))

      (define (bind-variable variable value)
        (define binding (cons variable value))
        (set! environment (cons binding environment)))

      (define (bind-parameters parameters arguments)
        (if (symbol? parameters)
          (bind-variable parameters arguments)
          (if (pair? parameters)
            (begin
              (define variable (car parameters))
              (define value (car arguments ))
              (bind-variable variable value)
              (bind-parameters (cdr parameters) (cdr arguments))))))

      (define (evaluate-sequence expressions)
        (define head (car expressions))
        (define tail (cdr expressions))
        (if (null? tail)
          (evaluate head)
          (begin
            (evaluate head)
            (evaluate-sequence tail))))
```

Slip in Slip (cont'd)

```
(define (make-procedure parameters expressions)
  (define lexical-environment environment)
  (lambda arguments
    (define dynamic-environment environment)
    (set! environment lexical-environment)
    (bind-parameters parameters arguments)
    (define value (evaluate-sequence expressions))
    (set! environment dynamic-environment)
    value))

(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))

(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))
```

```
(define (evaluate-define pattern . expressions)
  (if (symbol? pattern)
    (begin
      (define value (evaluate (car expressions)))
      (define binding (cons pattern value))
      (set! environment (cons binding environment))
      value)
    (begin
      (define binding (cons (car pattern) ()))
      (set! environment (cons binding environment))
      (define procedure (make-procedure (cdr pattern) expressions))
      (set-cdr! binding procedure)
      procedure)))
```

Slip in Slip (cont'd)

```
(define (evaluate-if predicate consequent . alternative)
  (define boolean (evaluate predicate))
  (if (eq? boolean #f)
    (if (null? alternative)
      '()
      (evaluate (car alternative)))
    (evaluate consequent)))

(define (evaluate-lambda parameters . expressions)
  (make-procedure parameters expressions))

(define (evaluate-quote expression)
  expression)

(define (evaluate-set! variable expression)
  (define value (evaluate expression))
  (define binding (assoc variable environment))
  (if (pair? binding)
    (begin
      (define value (evaluate expression))
      (set-cdr! binding value)
      value)
    (error "inaccessible variable: " variable)))

(define (evaluate-variable variable)
  (define binding (assoc variable environment))
  (if (pair? binding)
    (cdr binding)
    (meta-level-eval variable)))

(define (evaluate-while predicate . expressions)
  (define (iterate value)
    (define boolean (evaluate predicate))
    (if (eq? boolean #f)
      value
      (iterate (evaluate-sequence expressions))))
  (iterate ()))
```

Slip in Slip (cont'd)

```
(if (symbol? expression)
    (evaluate-variable expression)
    (if (pair? expression)
      (begin
        (define operator (car expression))
        (define operands (cdr expression))
        (apply
          (if (eq? operator 'begin) evaluate-begin
            (if (eq? operator 'define) evaluate-define
              (if (eq? operator 'if) evaluate-if
                (if (eq? operator 'lambda) evaluate-lambda
                  (if (eq? operator 'quote) evaluate-quote
                    (if (eq? operator 'set!) evaluate-set!
                      (if (eq? operator 'while) evaluate-while
                        (evaluate-application operator))))))))) operands))
            expression)))

  (display output)
  (newline)
  (display "level ")
  (display circularity-level)
  (display ">")
  (set! eval evaluate)
  (loop (evaluate (read))))

(loop "Meta-Circular Slip" ())
```

Of Chickens and Eggs

```
    (begin
      (define circularity-level 0)
      (define meta-level-eval eval)
      (define eval '())
        (loop (evaluate (read))))
      (loop "Root-Level Slip" '()))
    Slip version 3
    >>>
    (begin
      (define circularity-level (+ circularity-level 1))
      (define meta-level-eval eval)
      (define eval ())
      (define environment ())
      (define (loop output)
        (define rollback environment)
      (loop "Meta-Circular Slip" ()))
    Meta-Circular Slip
    level 1>
    (begin
      (define circularity-level (+ circularity-level 1))
      (loop "Meta-Circular Slip" ()))
    Meta-Circular Slip
    level 2>
    (begin
      (define circularity-level (+ circularity-level 1))
      (loop "Meta-Circular Slip" ()))
    Meta-Circular Slip
    level 3>(+ 1 2)
    3
    level 3>
```

What's next ...

- iteration syntax was introduced to avoid needing to solve the tail-recursion issue!
- we can change the language to make the interpreter properly tail recursive ...
- ... or we can look for a more structural solution and also lay the foundation for more general language processors

What's next ...

- iteration syntax was introduced to
  to solve the tail-recursion issue!
- we can change the language to ma
  interpreter properly tail recursive
- ... or we can look for a more structural solution and also lay the foundation for more general language processors

make everything first class and as a benefit introduce objects

What's next ...

- iteration syntax was introduced to avoid needing to solve the tail-recursion issue!
- we can change the language to make the interpreter properly tail recursive ...
- ... or we can look for a more structural solution and also lay the foundation for more general language processors

move to a continuation passing style