Vrije Universiteit Brussel

Software Languages Lab

Programming Language Engineering
Master of Computer Science

Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

Section 11: Small-step Semantics

Theo D'Hondt
Software Languages Lab

CESK machines

- introduced by Matthias Felleisen
- formal specification: see *
- short for
  - Control
  - Environment
  - Store
  - Kontinuation
- easily mapped to a data-driven interpreter

*http://www.ccs.neu.edu/racket/pubs/dissertation-felleisen.pdf and http://matt.might.net/articles/cesk-machines/

Racket for CESK-based interpreters

```
(struct document (author title content))
```

```
> (match '(1 (x y z) 1)
    [(list a b a) (list a b)]
    [(list a b c) (list c b a)])
'(1 (x y z))
```

Racket for CESK-based interpreters

```
(define ns (make-base-namespace))
(struct ev (e ρ σ κ))
(struct ko (φ v σ κ))
(struct ap (v vs ρ σ κ))
(struct letk (a es ρ))
(struct setk (x ρ))
(struct ifk (e1 e2 ρ))
(struct seqk (es ρ))
(struct randk (xs vs ρ))
(struct haltk ())
(struct clo (λ ρ))
(struct lam (x es))
(define (eval-seq es ρ σ κ)
 (match es
   ((list e) (ev e ρ σ κ))
   ((cons e es) (ev e ρ σ (cons (seqk es ρ) κ)))))
(define (step state)
 (match state
   ((ev (and x (? symbol?)) ρ σ (cons φ κ))
    (ko φ (match (assoc x ρ)
            ((cons _ a) (cdr (assoc a σ)))
            (_ (eval x ns))) σ κ))
((ev `(lambda ,x ,es ...) ρ σ (cons φ κ))
 (ko φ (clo (lam x es) ρ) σ κ))
((ev `(quote ,e) ρ σ (cons φ κ))
 (ko φ e σ κ))
((ev `(if ,e ,e1 ,e2) ρ σ κ)
 (ev e ρ σ (cons (ifk e1 e2 ρ) κ)))
((ev `(letrec ((,x ,e)) ,es ...) ρ σ κ)
 (let* ((fresh (gensym))
        (ρ* (cons (cons x fresh) ρ)))
   (ev e ρ* σ (cons (letk fresh es ρ*) κ))))
((ev `(set! ,x ,e) ρ σ κ)
 (ev e ρ σ (cons (setk x ρ) κ)))
((ev `(begin ,es ...) ρ σ κ)
 (eval-seq es ρ σ κ))
((ev `(,rator . ,rands) ρ σ κ)
 (ev rator ρ σ (cons (randk rands '() ρ) κ)))
((ev e ρ σ (cons φ κ))
 (ko φ e σ κ))
((ko (letk a es ρ) v σ κ)
 (let ((σ* (cons (cons a v) σ)))
   (eval-seq es ρ σ* κ)))
((ko (setk x ρ) v σ (cons φ κ))
 (match (assoc x ρ)
   ((cons name a) (ko φ v (cons (cons a v) σ) κ))))
((ko (randk '() vs ρ) v σ κ)
 (let ((vs (reverse (cons v vs))))
   (ap (car vs) (cdr vs) ρ σ κ)))
((ap (clo (lam x es) ρ*) rands ρ σ κ)
 (let loop ((x x) (rands rands) (ρ ρ*) (σ σ))
   (match x
     ('() (eval-seq es ρ σ κ))
     ((cons x xs)
      (let ((fresh (gensym)))
        (loop xs (cdr rands)
              (cons (cons x fresh) ρ)
              (cons (cons fresh (car rands)) σ))))
     ((and x (? symbol?))
      (let ((fresh (gensym)))
        (eval-seq es
                  (cons (cons x fresh) ρ)
                  (cons (cons fresh rands) σ) κ))))))
((ap rator rands _ σ (cons φ κ))
 (ko φ (apply rator rands) σ κ))
((ko (randk rands vs ρ) v σ κ)
 (ev (car rands) ρ σ (cons (randk (cdr rands) (cons v vs) ρ) κ)))
((ko (ifk _ e2 ρ) #f σ κ)
 (ev e2 ρ σ κ))
((ko (ifk e1 _ ρ) _ σ κ)
 (ev e1 ρ σ κ))
((ko (seqk (list e) ρ) _ σ κ)
 (ev e ρ σ κ))
((ko (seqk (cons e exps) ρ) _ σ κ)
 (ev e ρ σ (cons (seqk exps ρ) κ)))
((ko (haltk) v _ _)
 #f)))
```

```
(define (inject e)
 (ev e '() '() `(,(haltk))))
(define (run s)
 (let ((next (step s)))
   (if next
       (run next)
       s)))
(define (extract s)
 (match s
   ((ko (haltk) v _ _) v)
   (_ 'error)))
(define (state-eval e)
 (extract (run (inject e))))
```

CESK-based interpreters for Slippy

```
(define ns (make-base-namespace))
(struct ev (e ρ σ κ))
(struct ko (φ v σ κ))
(struct ap (v vs ρ σ κ))
(struct letk (a es ρ))
(struct setk (x ρ))
(struct ifk (e1 e2 ρ))
(struct seqk (es ρ))
(struct randk (xs vs ρ))
(struct haltk ())
(struct clo (λ ρ))
(struct lam (x es))
(define (eval-seq es ρ σ κ)
 (match es
   ((list e) (ev e ρ σ κ))
   ((cons e es) (ev e ρ σ (cons (seqk es ρ) κ)))))
(define (step state)
 (match state
   ((ev (and x (? symbol?)) ρ σ (cons φ κ))
    (ko φ (match (assoc x ρ)
            ((cons _ a) (cdr (assoc a σ)))
            (_ (eval x ns))) σ κ))
((ev `(lambda ,x ,es ...) ρ σ (cons φ κ))
 (ko φ (clo (lam x es) ρ) σ κ))
((ev `(quote ,e) ρ σ (cons φ κ))
 (ko φ e σ κ))
((ev `(if ,e ,e1 ,e2) ρ σ κ)
 (ev e ρ σ (cons (ifk e1 e2 ρ) κ)))
((ev `(letrec ((,x ,e) ...) ,es ...) ρ σ κ)
 (let* ((fresh (gensym))
        (ρ* (cons (cons x fresh) ρ)))
   (ev e ρ* σ (cons (letk fresh es ρ*) κ))))
((ev `(set! ,x ,e) ρ σ κ)
 (ev e ρ σ (cons (setk x ρ) κ)))
((ev `(begin ,es ...) ρ σ κ)
 (eval-seq es ρ σ κ))
((ev `(,rator . ,rands) ρ σ κ)
 (ev rator ρ σ (cons (randk rands '() ρ) κ)))
((ev e ρ σ (cons φ κ))
 (ko φ e σ κ))
((ko (letk a es ρ) v σ κ)
 (let ((σ* (cons (cons a v) σ)))
   (eval-seq es ρ σ* κ)))
((ko (setk x ρ) v σ (cons φ κ))
 (match (assoc x ρ)
   ((cons name a) (ko φ v (cons (cons a v) σ) κ))))
((ko (randk '() vs ρ) v σ κ)
 (let ((vs (reverse (cons v vs))))
   (ap (car vs) (cdr vs) ρ σ κ)))
((ap (clo (lam x es) ρ*) rands ρ σ κ)
 (let loop ((x x) (rands rands) (ρ ρ*) (σ σ))
   (match x
     ('() (eval-seq es ρ σ κ))
     ((cons x xs)
      (let ((fresh (gensym)))
        (loop xs (cdr rands)
              (cons (cons x fresh) ρ)
              (cons (cons fresh (car rands)) σ))))
     ((and x (? symbol?))
      (let ((fresh (gensym)))
        (eval-seq es
                  (cons (cons x fresh) ρ)
                  (cons (cons fresh rands) σ) κ))))))
((ap rator rands _ σ (cons φ κ))
 (ko φ (apply rator rands) σ κ))
((ko (randk rands vs ρ) v σ κ)
 (ev (car rands) ρ σ (cons (randk (cdr rands) (cons v vs) ρ) κ)))
((ko (ifk _ e2 ρ) #f σ κ)
 (ev e2 ρ σ κ))
((ko (ifk e1 _ ρ) _ σ κ)
 (ev e1 ρ σ κ))
((ko (seqk (list e) ρ) _ σ κ)
 (ev e ρ σ κ))
((ko (seqk (cons e exps) ρ) _ σ κ)
 (ev e ρ σ (cons (seqk exps ρ) κ)))
((ko (haltk) v _ _)
 #f)))
```

```
(define (inject e)
 (ev e '() '() `(,(haltk))))
(define (run s)
 (let ((next (step s)))
   (if next
       (run next)
       s)))
(define (extract s)
 (match s
   ((ko (haltk) v _ _) v)
   (_ 'error)))
(define (state-eval e)
 (extract (run (inject e))))
```

CESK-based interpreters for Slip

states

```
(define ns (make-base-namespace))
(struct ev (e ρ σ κ))
(struct ko (φ v σ κ))
(struct ap (v vs ρ σ κ))
(struct letk (a es ρ))
(struct setk (x ρ))
(struct ifk (e1 e2 ρ))
(struct seqk (es ρ))
(struct randk (xs vs ρ))
(struct haltk ())
(struct clo (λ ρ))
(struct lam (x es))
(define (eval-seq es ρ σ κ)
 (match es
   ((list e) (ev e ρ σ κ))
   ((cons e es) (ev e ρ σ (cons (seqk es ρ) κ)))))
(define (step state)
 (match state
   ((ev (and x (? symbol?)) ρ σ (cons φ κ))
    (ko φ (match (assoc x ρ)
            ((cons _ a) (cdr (assoc a σ)))
            (_ (eval x ns))) σ κ))
   ((ev `(lambda ,x ,es ...) ρ σ (cons φ κ))
    (ko φ (clo (lam x es) ρ) σ κ))
   ((ev `(quote ,e) ρ σ (cons φ κ))
    (ko φ e σ κ))
   ((ev `(if ,e ,e1 ,e2) ρ σ κ)
    (ev e ρ σ (cons (ifk e1 e2 ρ) κ)))
   ((ev `(letrec ((,x ,e)) ,es ...) ρ σ κ)
    (let* ((fresh (gensym))
           (ρ* (cons (cons x fresh) ρ)))
      (ev e ρ* σ (cons (letk fresh es ρ*) κ))))
   ((ev `(set! ,x ,e) ρ σ κ)
    (ev e ρ σ (cons (setk x ρ) κ)))
   ((ev `(begin ,es ...) ρ σ κ)
    (eval-seq es ρ σ κ))
   ((ev `(,rator . ,rands) ρ σ κ)
    (ev rator ρ σ (cons (randk rands '() ρ) κ)))
   ((ev e ρ σ (cons φ κ))
    (ko φ e σ κ))
   ((ko φ e σ κ)
    (ko φ e σ κ))
   ((ko (letk a es ρ) v σ κ)
    (let ((σ* (cons (cons a v) σ)))
      (eval-seq es ρ σ* κ)))
   ((ko (setk x ρ) v σ (cons φ κ))
    (match (assoc x ρ)
      ((cons name a) (ko φ v (cons (cons a v) σ) κ))))
   ((ko (randk '() vs ρ) v σ κ)
    (let ((vs (reverse (cons v vs))))
      (ap (car vs) (cdr vs) ρ σ κ)))
   ((ap (clo (lam x es) ρ*) rands ρ σ κ)
    (let loop ((x x) (rands rands) (ρ ρ*) (σ σ))
      (match x
        ('() (eval-seq es ρ σ κ))
        ((cons x xs)
         (let ((fresh (gensym)))
           (loop xs (cdr rands)
                 (cons (cons x fresh) ρ)
                 (cons (cons fresh (car rands)) σ))))
        ((and x (? symbol?))
         (let ((fresh (gensym)))
           (eval-seq es
                     (cons (cons x fresh) ρ)
                     (cons (cons fresh rands) σ) κ))))))
   ((ap rator rands _ σ (cons φ κ))
    (ko φ (apply rator rands) σ κ))
   ((ko (randk rands vs ρ) v σ κ)
    (ev (car rands) ρ σ (cons (randk (cdr rands) (cons v vs) ρ) κ)))
   ((ko (ifk _ e2 ρ) #f σ κ)
    (ev e2 ρ σ κ))
   ((ko (ifk e1 _ ρ) _ σ κ)
    (ev e1 ρ σ κ))
   ((ko (seqk (list e) ρ) _ σ κ)
    (ev e ρ σ κ))
   ((ko (seqk (cons e exps) ρ) _ σ κ)
    (ev e ρ σ (cons (seqk exps ρ) κ)))
   ((ko (haltk) v _ _)
    #f)))
```

```
(define (inject e)
 (ev e '() '() `(,(haltk))))
(define (run s)
 (let ((next (step s)))
   (if next
       (run next)
       s)))
(define (extract s)
 (match s
   ((ko (haltk) v _ _) v)
   (_ 'error)))
(define (state-eval e)
 (extract (run (inject e))))
```

CESK-based interpreters for Slip

evaluation steps

```
(define ns (make-base-namespace))
(struct ev (e ρ σ κ))
(struct ko (φ v σ κ))
(struct ap (v vs ρ σ κ))
(struct letk (a es ρ))
(struct setk (x ρ))
(struct ifk (e1 e2 ρ))
(struct seqk (es ρ))
(struct randk (es vs ρ))
(struct haltk ())
(struct clo (λ ρ))
(struct lam (x es))
(define (eval-seq es ρ σ κ)
 (match es
   ((list e) (ev e ρ σ κ))
   ((cons e es) (ev e ρ σ (cons (seqk es ρ) κ)))))
(define (step state)
 (match state
  ((ev (and x (? symbol?)) ρ σ (cons φ κ))
   (ko φ (match (assoc x ρ)
           ((cons _ a) (cdr (assoc a σ)))
           (_ (eval x ns))) σ κ))
  ((ev `(lambda ,x ,es ...) ρ σ (cons φ κ))
   (ko φ (clo (lam x es) ρ) σ κ))
  ((ev `(quote ,e) ρ σ (cons φ κ))
   (ko φ e σ κ))
  ((ev `(if ,e ,e1 ,e2) ρ σ κ)
   (ev e ρ σ (cons (ifk e1 e2 ρ) κ)))
  ((ev `(letrec ((,x ,e)) ,es ...) ρ σ κ)
   (let* ((fresh (gensym))
          (ρ* (cons (cons x fresh) ρ)))
     (ev e ρ* σ (cons (letk fresh es ρ*) κ))))
  ((ev `(set! ,x ,e) ρ σ κ)
   (ev e ρ σ (cons (setk x ρ) κ)))
  ((ev `(begin ,es ...) ρ σ κ)
   (eval-seq es ρ σ κ))
  ((ev `(,rator . ,rands) ρ σ κ)
   (ev rator ρ σ (cons (randk rands '() ρ) κ)))
  ((ev e ρ σ (cons φ κ))
   (ko φ e σ κ))
  ((ko (letk a es ρ) v σ κ)
   (let ((σ* (cons (cons a v) σ)))
     (eval-seq es ρ σ* κ)))
  ((ko (setk x ρ) v σ (cons φ κ))
   (match (assoc x ρ)
     ((cons name a) (ko φ v (cons (cons a v) σ) κ))))
  ((ko (randk rands '() vs ρ) v σ κ)
   (let ((vs (reverse (cons v vs))))
     (ap (car vs) (cdr vs) ρ σ κ)))
  ((ap (clo (lam x es) ρ*) rands ρ σ κ)
   (let loop ((x x) (rands rands) (ρ ρ*) (σ σ))
     (match x
       ('() (eval-seq es ρ σ κ))
       ((cons x xs)
        (let ((fresh (gensym)))
          (loop xs (cdr rands)
                (cons (cons x fresh) ρ)
                (cons (cons fresh (car rands)) σ))))
       ((and x (? symbol?))
        (let ((fresh (gensym)))
          (eval-seq es
                    (cons (cons x fresh) ρ)
                    (cons (cons fresh rands) σ) κ))))))
  ((ap rator rands _ σ (cons φ κ))
   (ko φ (apply rator rands) σ κ))
  ((ko (randk rands vs ρ) v σ κ)
   (ev (car rands) ρ σ (cons (randk (cdr rands) (cons v vs) ρ) κ)))
  ((ko (ifk _ e2 ρ) #f σ κ)
   (ev e2 ρ σ κ))
  ((ko (ifk e1 _ ρ) _ σ κ)
   (ev e1 ρ σ κ))
  ((ko (seqk (list e) ρ) _ σ κ)
   (ev e ρ σ κ))
  ((ko (seqk (cons e exps) ρ) _ σ κ)
   (ev e ρ σ (cons (seqk exps ρ) κ)))
  ((ko (haltk) v _ _)
   #f)))
```

```
(define (inject e)
 (ev e '() '() `(,(haltk))))
(define (run s)
 (let ((next (step s)))
   (if next
       (run next)
       s)))
(define (extract s)
 (match s
   ((ko (haltk) v _ _) v)
   (_ 'error)))
(define (state-eval e)
 (extract (run (inject e))))
```

CESK-based interpreters for Slip

continuation steps

```
(define ns (make-base-namespace))
(struct ev (e ρ σ κ))
(struct ko (φ v σ κ))
(struct ap (v vs ρ σ κ))
(struct letk (a es ρ))
(struct setk (x ρ))
(struct ifk (e1 e2 ρ))
(struct seqk (es ρ))
(struct randk (xs vs ρ))
(struct haltk ())
(struct clo (λ ρ))
(struct lam (x es))
(define (eval-seq es ρ σ κ)
 (match es
   ((list e) (ev e ρ σ κ))
   ((cons e es) (ev e ρ σ (cons (seqk es ρ) κ)))))
(define (step state)
 (match state
   ((ev (and x (? symbol?)) ρ σ (cons φ κ))
    (ko φ (match (assoc x ρ)
            ((cons _ a) (cdr (assoc a σ)))
            (_ (eval x ns))) σ κ))
((ev `(lambda ,x ,es ...) ρ σ (cons φ κ))
 (ko φ (clo (lam x es) ρ) σ κ))
((ev `(quote ,e) ρ σ (cons φ κ))
 (ko φ e σ κ))
((ev `(if ,e ,e1 ,e2) ρ σ κ)
 (ev e ρ σ (cons (ifk e1 e2 ρ) κ)))
((ev `(letrec ((,x ,e)) ,es ...) ρ σ κ)
 (let* ((fresh (gensym))
        (ρ* (cons (cons x fresh) ρ)))
   (ev e ρ* σ (cons (letk fresh es ρ*) κ))))
((ev `(set! ,x ,e) ρ σ κ)
 (ev e ρ σ (cons (setk x ρ) κ)))
((ev `(begin ,es ...) ρ σ κ)
 (eval-seq es ρ σ κ))
((ev `(,rator . ,rands) ρ σ κ)
 (ev rator ρ σ (cons (randk rands '() ρ) κ)))
((ev e ρ σ (cons φ κ))
 (ko φ e σ κ))
((ko (letk a es ρ) v σ κ)
 (let ((σ* (cons (cons a v) σ)))
   (eval-seq es ρ σ* κ)))
((ko (setk x ρ) v σ (cons φ κ))
 (match (assoc x ρ)
   ((cons name a) (ko φ v (cons (cons a v) σ) κ))))
((ko (randk '() vs ρ) v σ κ)
 (let ((vs (reverse (cons v vs))))
   (ap (car vs) (cdr vs) ρ σ κ)))
((ap (clo (lam x es) ρ*) rands ρ σ κ)
 (let loop ((x x) (rands rands) (ρ ρ*) (σ σ))
   (match x
     ('() (eval-seq es ρ σ κ))
     (`(,x . ,xs)
      (let ((fresh (gensym)))
        (loop xs (cdr rands)
              (cons (cons x fresh) ρ)
              (cons (cons fresh (car rands)) σ))))
     ((and x (? symbol?))
      (let ((fresh (gensym)))
        (eval-seq es
                  (cons (cons x fresh) ρ)
                  (cons (cons fresh rands) σ) κ))))))
((ap rator rands _ σ (cons φ κ))
 (ko φ (apply rator rands) σ κ))
((ko (randk rands vs ρ) v σ κ)
 (ev (car rands) ρ σ (cons (randk (cdr rands) (cons v vs) ρ) κ)))
((ko (ifk _ e2 ρ) #f σ κ)
 (ev e2 ρ σ κ))
((ko (ifk e1 _ ρ) _ σ κ)
 (ev e1 ρ σ κ))
((ko (seqk (list e) ρ) _ σ κ)
 (ev e ρ σ κ))
((ko (seqk (cons e exps) ρ) _ σ κ)
 (ev e ρ σ (cons (seqk exps ρ) κ)))
((ko (haltk) v _ _)
 #f)))
```

```
(define (inject e)
 (ev e '() '() `(,(haltk))))
(define (run s)
 (let ((next (step s)))
   (if next
       (run next)
       s)))
(define (extract s)
 (match s
   ((ko (haltk) v _ _) v)
   (_ 'error)))
(define (state-eval e)
 (extract (run (inject e))))
```

CESK-based interpreters for Slip

```
(define ns (make-base-namespace))
(struct ev (e ρ σ κ))
(struct ko (φ v σ κ))
(struct ap (v vs ρ σ κ))
(struct letk (a es ρ))
(struct setk (x ρ))
(struct ifk (e1 e2 ρ))
(struct seqk (es ρ))
(struct randk (xs vs ρ))
(struct haltk ())
(struct clo (λ ρ))
(struct lam (x es))
(define (eval-seq es ρ σ κ)
 (match es
   ((list e) (ev e ρ σ κ))
   ((cons e es) (ev e ρ σ (cons (seqk es ρ) κ)))))
(define (step state)
 (match state
   ((ev (and x (? symbol?)) ρ σ (cons φ κ))
    (ko φ (match (assoc x ρ)
            ((cons _ a) (cdr (assoc a σ)))
            (_ (eval x ns))) σ κ))
((ev `(lambda ,x ,es ...) ρ σ (cons φ κ))
 (ko φ (clo (lam x es) ρ) σ κ))
((ev `(quote ,e) ρ σ (cons φ κ))
 (ko φ e σ κ))
((ev `(if ,e ,e1 ,e2) ρ σ κ)
 (ev e ρ σ (cons (ifk e1 e2 ρ) κ)))
((ev `(letrec ((,x ,e)) ,es ...) ρ σ κ)
 (let* ((fresh (gensym))
        (ρ* (cons (cons x fresh) ρ)))
   (ev e ρ* σ (cons (letk fresh es ρ*) κ))))
((ev `(set! ,x ,e) ρ σ κ)
 (ev e ρ σ (cons (setk x ρ) κ)))
((ev `(begin ,es ...) ρ σ κ)
 (eval-seq es ρ σ κ))
((ev `(,rator . ,rands) ρ σ κ)
 (ev rator ρ σ (cons (randk rands '() ρ) κ)))
((ev e ρ σ (cons φ κ))
 (ko φ e σ κ))
((ko (letk a es ρ) v σ κ)
 (let ((σ* (cons (cons a v) σ)))
   (eval-seq es ρ σ* κ)))
((ko (setk x ρ) v σ (cons φ κ))
 (match (assoc x ρ)
   ((cons name a) (ko φ v (cons (cons a v) σ) κ))))
((ko (randk '() vs ρ) v σ κ)
 (let ((vs (reverse (cons v vs))))
   (ap (car vs) (cdr vs) ρ σ κ)))
((ap (clo (lam x es) ρ*) rands ρ σ κ)
 (let loop ((x x) (rands rands) (ρ ρ*) (σ σ))
   (match x
     ('() (eval-seq es ρ σ κ))
     ((cons x xs)
      (let ((fresh (gensym)))
        (loop xs (cdr rands)
              (cons (cons x fresh) ρ)
              (cons (cons fresh (car rands)) σ))))
     ((and x (? symbol?))
      (let ((fresh (gensym)))
        (eval-seq es
                  (cons (cons x fresh) ρ)
                  (cons (cons fresh rands) σ) κ))))))
((ap rator rands _ σ (cons φ κ))
 (ko φ (apply rator rands) σ κ))
((ko (randk rands vs ρ) v σ κ)
 (ev (car rands) ρ σ (cons (randk (cdr rands) (cons v vs) ρ) κ)))
((ko (ifk _ e2 ρ) #f σ κ)
 (ev e2 ρ σ κ))
((ko (ifk e1 _ ρ) _ σ κ)
 (ev e1 ρ σ κ))
((ko (seqk (list e) ρ) _ σ κ)
 (ev e ρ σ κ))
((ko (seqk (cons e exps) ρ) _ σ κ)
 (ev e ρ σ (cons (seqk exps ρ) κ)))
((ko (haltk) v _ _)
 #f)))
```

```
(define (inject e)
 (ev e '() '() `(,(haltk))))
(define (run s)
 (let ((next (step s)))
   (if next
       (run next)
       s)))
(define (extract s)
 (match s
   ((ko (haltk) v _ _) v)
   (_ 'error)))
(define (state-eval e)
 (extract (run (inject e))))
```

application steps

CESK-based interpreter for Slip

```
(struct ev (e ρ σ κ))
(struct ko (φ v ρ* σ*))
(struct defk (fresh))
(struct setk (x))
(struct ifk (e))
(struct ifek (e1 e2))
(struct seqk (es))
(struct randk (rands vs))
(struct thkk (ρ))
(struct repk ())
(struct clo (λ ρ))
(struct lam (x es))

(define (eval-seq es ρ σ κ)
  (match es
    ((list e) (ev e ρ σ κ))
    ((cons e es) (ev e ρ σ (cons (seqk es) κ)))))

(define (step state)
  (match state
    ((ev (and x (? symbol?)) ρ σ (cons φ κ))
     (ko φ (match (assoc x ρ)
             ((cons _ a) (cdr (assoc a σ)))
             (_ (eval x ns))) ρ σ κ))

   ((ev `(lambda ,x ,es ...) ρ σ (cons φ κ))
    (let ((clo (clo (lam x es) ρ)))
      (ko φ clo ρ σ κ)))

   ((ev `(quote ,e) ρ σ (cons φ κ))
    (ko φ e ρ σ))    ; <--- e should be cloned

   ((ev `(if ,e ,e1) ρ σ κ)
    (ev e ρ σ (cons (ifk e1) κ)))

   ((ev `(if ,e ,e1 ,e2) ρ σ κ)
    (ev e ρ σ (cons (ifek e1 e2) κ)))

   ((ev `(define ,(cons x xs) ,es ...) ρ σ (cons φ κ))
    (let* ((fresh (gensym))
           (ρ* (cons (cons x fresh) ρ))
           (clo (clo (lam xs es) ρ*))
           (σ* (cons (cons fresh clo) σ)))
      (ko φ clo ρ* σ* κ)))
```

```
σ κ)
)
s x fresh) ρ))
efk fresh) κ))))

 κ)
 x) κ)))

ρ σ κ)


) ρ σ κ)
randk rands '()) κ)))



σ (cons φ κ))
 fresh v) σ)))


ons φ κ))


φ v ρ (cons (cons a v) σ) κ))))

ρ σ (cons φ κ))
 v vs))
es) ρ*) rands)
rands rands) (ρ ρ*) (σ σ))

 es ρ σ (cons (thkk ρ) (cons φ κ))))

(gensym)))
dr rands)
 (cons x fresh) ρ)
 (cons fresh (car rands)) σ))))
ool?))
 (gensym))
ons (cons x fresh) ρ))
ons (cons fresh rands) σ)))
s ρ* σ* (cons (thkk ρ) (cons φ κ))))))))
)
r rands) ρ σ κ))))
```

```
cdr rands) (cons v vs)) κ)))
```

```
') (ev (read) ρ σ (list

ρ Slip" '() '() '())))
```

Slip

```
(struct ev (e ρ σ κ))
(struct ko (φ v ρ* σ κ))
(struct defk (fresh))
(struct setk (x))
(struct ifk (e))
(struct ifek (e1 e2))
(struct seqk (es))
(struct randk (rands vs))
(struct thkk (ρ))
(struct repk ())
(struct clo (λ ρ))
(struct lam (x es))

(define (eval-seq es ρ σ κ)
  (match es
    ((list e) (ev e ρ σ κ))
    ((cons e es) (ev e ρ σ (cons (seqk es) κ)))))

(define (step state)
  (match state
    ((ev (and x (? symbol?)) ρ σ (cons φ κ))
     (ko φ (match (assoc x ρ)
             ((cons _ a) (cdr (assoc a σ)))
             (_ (eval x ns))) ρ σ κ))

   ((ev `(lambda ,x ,es ...) ρ σ (cons φ κ))
    (let ((clo (clo (lam x es) ρ)))
      (ko φ clo ρ σ κ)))

   ((ev `(quote ,e) ρ σ (cons φ κ))
    (ko φ e ρ σ))    ; <--- e should be cloned

   ((ev `(if ,e ,e1) ρ σ κ)
    (ev e ρ σ (cons (ifk e1) κ)))

   ((ev `(if ,e ,e1 ,e2) ρ σ κ)
    (ev e ρ σ (cons (ifek e1 e2) κ)))

   ((ev `(define ,(cons x xs) ,es ...) ρ σ (cons φ κ))
    (let* ((fresh (gensym))
           (ρ* (cons (cons x fresh) ρ))
           (clo (clo (lam xs es) ρ*))
           (σ* (cons (cons fresh clo) σ)))
      (ko φ clo ρ* σ* κ)))
```

```
 σ κ)
))
s x fresh) ρ)))
efk fresh) κ))))

 κ)
 x) κ)))

ρ σ κ)

) ρ σ κ)
randk rands '()) κ)))

σ (cons φ κ))
 fresh v) σ)))

ons φ κ))

φ v ρ (cons (cons a v) σ) κ))))

ρ σ (cons φ κ))
 v vs))
es) ρ*) rands)
rands rands) (ρ ρ*) (σ σ))

 es ρ σ (cons (thkk ρ) (cons φ κ))))

(gensym)))
dr rands)
 (cons x fresh) ρ)
 (cons fresh (car rands)) σ))))
ool?))
 (gensym))
ons (cons x fresh) ρ))
ons (cons fresh rands) σ)))
s ρ* σ* (cons (thkk ρ) (cons φ κ))))))))
)
r rands) ρ σ κ))))
```

```
ands) (cons v vs)) κ)))
```

```
v (read) ρ σ (list (repk))))))

p" '() '() '()))))
```

Slip

```
(struct ev (e ρ σ κ))
(struct ko (φ v ρ* σ κ))
(struct defk (fresh))
(struct setk (x))
(struct ifk (e))
(struct ifek (e1 e2))
(struct seqk (es))
(struct randk (rands vs))
(struct thkk (ρ))
(struct repk ())
(struct clo (λ ρ))
(struct lam (x es))

(define (eval-seq es ρ σ κ)
  (match es
    ((list e) (ev e ρ σ κ))
    ((cons e es) (ev e ρ σ (co

(define (step state)
  (match state
    ((ev (and x (? symbol?)) ρ
     (ko φ (match (assoc x ρ)
             ((cons _ a) (cdr
             (_ (eval x ns)))

    ((ev `(lambda ,x ,es ...)
     (let ((clo (clo (lam x es
       (ko φ clo ρ σ κ)))

    ((ev `(quote ,e) ρ σ (cons
     (ko φ e ρ σ))    ; <--- e

    ((ev `(if ,e ,e1) ρ σ κ)
     (ev e ρ σ (cons (ifk e1)

    ((ev `(if ,e ,e1 ,e2) ρ σ
     (ev e ρ σ (cons (ifek e1

    ((ev `(define ,(cons x xs)
     (let* ((fresh (gensym))
            (ρ* (cons (cons x
            (clo (clo (lam xs
            (σ* (cons (cons fr
       (ko φ clo ρ* σ* κ)))
```

```
    ((ev `(define ,x ,e) ρ σ κ)
     (let* ((fresh (gensym))
            (ρ* (cons (cons x fresh) ρ)))
       (ev e ρ* σ (cons (defk fresh) κ))))

    ((ev `(set! ,x ,e) ρ σ κ)
     (ev e ρ σ (cons (setk x) κ)))

    ((ev `(begin ,es ...) ρ σ κ)
     (eval-seq es ρ σ κ))

    ((ev `(,rator . ,rands) ρ σ κ)
     (ev rator ρ σ (cons (randk rands '()) κ)))

    ((ev e ρ σ (cons φ κ))
     (ko φ e ρ σ κ))

    ((ko (defk fresh) v ρ σ (cons φ κ))
     (let ((σ* (cons (cons fresh v) σ)))
       (ko φ v ρ σ* κ)))

    ((ko (setk x) v ρ σ (cons φ κ))
     (match (assoc x ρ)
       ((cons name a) (ko φ v ρ (cons (cons a v) σ) κ))))

    ((ko (randk '() vs) v ρ σ (cons φ κ))
     (match (reverse (cons v vs))
       ((cons (clo (lam x es) ρ*) rands)
        (let loop ((x x) (rands rands) (ρ ρ*) (σ σ))
          (match x
            ('() (eval-seq es ρ σ (cons (thkk ρ) (cons φ κ))))
            ((cons x xs)
             (let ((fresh (gensym)))
               (loop xs (cdr rands)
                     (cons (cons x fresh) ρ)
                     (cons (cons fresh (car rands)) σ))))
            ((and x (? symbol?))
             (let* ((fresh (gensym))
                    (ρ* (cons (cons x fresh) ρ))
                    (σ* (cons (cons fresh rands) σ)))
               (eval-seq es ρ* σ* (cons (thkk ρ) (cons φ κ))))))))
       ((cons rator rands)
        (ko φ (apply rator rands) ρ σ κ))))
```

```
ands) (cons v vs)) κ)))
```

```
v (read) ρ σ (list (repk))))))

p" '() '() '())))
```

Small-Step Slip

```
(struct ev (e ρ σ κ))
(struct ko (φ v ρ* σ κ))
(struct defk (fresh))
(struct setk (x))
(struct ifk (e))
(struct ifek (e1 e2))
(struct seqk (es))
(struct randk (rands vs))
(struct thkk (ρ))
(struct repk ())
(struct clo (λ ρ))
(struct lam (x es))

(define (eval-seq es ρ σ κ)
  (match es
    ((list e) (ev e ρ σ κ))
    ((cons e es) (ev e ρ σ (co

(define (step state)
  (match state
    ((ev (and x (? symbol?)) ρ
     (ko φ (match (assoc x ρ)
             ((cons _ a) (cdr
             (_ (eval x ns)))

    ((ev `(lambda ,x ,es ...)
     (let ((clo (clo (lam x es
       (ko φ clo ρ σ κ)))

    ((ev `(quote ,e) ρ σ (cons
     (ko φ e ρ σ))    ; <--- e

    ((ev `(if ,e ,e1) ρ σ κ)
     (ev e ρ σ (cons (ifk e1)

    ((ev `(if ,e ,e1 ,e2) ρ σ
     (ev e ρ σ (cons (ifek e1

    ((ev `(define ,(cons x xs)
     (let* ((fresh (gensym))
            (ρ* (cons (cons x
            (clo (clo (lam xs
            (σ* (cons (cons fr
       (ko φ clo ρ* σ* κ)))
```

```
    ((ev `(define ,x ,e) ρ σ κ)
     (let* ((fresh (gensym))
            (ρ* (cons (cons x fresh) ρ)))
       (ev e ρ* σ (cons (defk fresh) κ))))

    ((ev `(set! ,x ,e) ρ σ κ)
     (ev e ρ σ (cons (se

    ((ev `(begin ,es ...
     (eval-seq es ρ σ κ)

    ((ev `(,rator . ,ran
     (ev rator ρ σ (cons

    ((ev e ρ σ (cons φ κ
     (ko φ e ρ σ κ))

    ((ko (defk fresh) v
     (let ((σ* (cons (co
      (ko φ v ρ σ* κ)))

    ((ko (setk x) v ρ σ
     (match (assoc x ρ)
       ((cons name a) (k

    ((ko (randk '() vs)
     (match (reverse (co
       ((cons (clo (lam
        (let loop ((x x)
          (match x
            ('() (eval-s
            ((cons x xs)
             (let ((fres
               (loop xs
                     (co
                     (co
            ((and x (? s
             (let* ((fre
                    (ρ* (cons (cons x fresh) ρ))
                    (σ* (cons (cons fresh rands) σ)))
               (eval-seq es ρ* σ* (cons (thkk ρ) (cons φ κ))))))))
       ((cons rator rands)
        (ko φ (apply rator rands) ρ σ κ))))
```

```
  ((ko (randk rands vs) v ρ σ κ)
    (ev (car rands) ρ σ (cons (randk (cdr rands) (cons v vs)) κ)))

   ((ko (ifk _) #f ρ σ (cons φ κ))
    (ko φ '() ρ σ κ))

   ((ko (ifk e) _ ρ σ κ)
    (ev e ρ σ (cons (thkk ρ) κ)))

   ((ko (ifek _ e2) #f ρ σ κ)
    (ev e2 ρ σ (cons (thkk ρ) κ)))

   ((ko (ifek e1 _) _ ρ σ κ)
    (ev e1 ρ σ (cons (thkk ρ) κ)))

   ((ko (seqk (list e)) _ ρ σ κ)
    (ev e ρ σ κ))

   ((ko (seqk (cons e exps) ) _ ρ σ κ)
    (ev e ρ σ (cons (seqk exps) κ)))

   ((ko (thkk ρ) v _ σ (cons φ κ))
    (ko φ v ρ σ κ))

   ((ko (repk) v ρ σ _)
    (display v) (newline) (display "> ") (ev (read) ρ σ (list (repk))))))

(let loop ((state (ko (repk) "Small-Step Slip" '() '() '())))
  (loop (step state)))
```

“EFKES” semantics for Slip

- frame-based environments
- inclusion of higher-order native functions
- short for
  - Expression
  - Frame
  - Kontinuation
  - Environment
  - Store
- inspired by denotaional semantics

“EFKES” semantics for Slip

```
#lang racket

(require compatibility/mlist)

(struct ap (π vs ρ ρs σ κs τ ρ0))        ; application pattern
(struct ev (e ρ ρs σ κs τ ρ0))           ; evaluation pattern
(struct ko (κ v ρ ρs σ κs))              ; continuation pattern

(struct defk (id))                       ; define form continuation
(struct evalk (τ ρ0))                    ; evaluate form continuation
(struct fchk (π vs ρ0))                  ; for-each native continuation
(struct ifk (e τ ρ0))                    ; if form continuation
(struct ifek (e1 e2 τ ρ0))               ; if-else form continuation
(struct mppk (π vs vs* ρ0))              ; map native  continuation
(struct multiplek (v vs τ ρ0))           ; multiple argument continuation
(struct nullk (τ ρ0))                    ; no argument continuation
(struct randk (π rands vs τ ρ0))         ; operand continuation
(struct ratork (rands τ ρ0))             ; operator continuation
(struct repk ())                         ; REP continuation
(struct retk (ρ ρs))                     ; return continuation
(struct seqk (es τ ρ0))                  ; sequence continuation
(struct setk (id ρ0))                    ; set form continuation
(struct singlek (v τ ρ0))                ; single argument continuation

(struct apl ())                          ; apply native
(struct ccc ())                          ; call-with-current-continuation native
(struct clo (ps es ρs))                  ; closure
(struct con (ρ ρs κs))                   ; continuation
(struct fch ())                          ; for-each native
(struct mpp ())                          ; map native
(struct nat (π))                         ; regular native
(struct nyi ())                          ; not yet implemented native
(struct rea ())                          ; read native
(struct undefined ())                    ; undefined pattern
(struct unspecified ())                  ; unspecified pattern
```

“EFKES” semantics for Slip

```
(define (step state)
  (match state

    ; variable reference

    ((ev (and e (? symbol?)) ρ ρs σ (cons κ κs) τ ρ0)
     (let loop ((ρ* ρ)
                (ρs* ρs))
       (match (assoc e ρ*)
         ((cons _ a)
          (ko κ (cdr (assoc a σ)) ρ ρs σ κs))
         (_
          (match ρs*
            ((cons ρ ρs)
             (loop ρ ρs))
            (_
             (ko κ (undefined) ρ ρs σ κs)))))))


    . . .


; read-eval-print

((ko (repk) v ρ ρs σ _)
 (display v)
 (newline)
 (display "› ")
 (ev (read) ρ ρs σ (list (repk)) #f ρ))))
```

“EFKES” semantics for Slip

```
; expression sequence

((ev `(begin ,es ...) ρ ρs σ κs τ ρ0)
 (seq es ρ ρs σ κs τ ρ0))

((ko (seqk es τ ρ0) _ ρ ρs σ κs)
 (seq es ρ ρs σ κs τ ρ0))
```

```
; expression sequence

(define (seq es ρ ρs σ κs τ ρ0)
  (match es
    ((list e)
     (ev e ρ ρs σ κs τ ρ0))
    ((cons e es)
     (ev e ρ ρs σ (cons (seqk es τ ρ0) κs) #f ρ0))
    (_
     (rep "expression sequence required" σ ρ0))))
```

“EFKES” semantics for Slip

```
; definition

((ev `(define (,id . ,ps) ,es ...) ρ ρs σ (cons κ κs) _ ρ0)
 (match (assoc id ρ)
   ((cons _ a)
    (let* ((clo (clo ps es (cons ρ ρs)))
           (σ (cons (cons a clo) σ)))
      (ko κ clo ρ ρs σ κs)))
   (_
    (let* ((fresh (gensym))
           (ρ (cons (cons id fresh) ρ))
           (clo (clo ps es (cons ρ ρs)))
           (σ (cons (cons fresh clo) σ)))
      (ko κ clo ρ ρs σ κs)))))

; ---

((ev `(define ,id ,e) ρ ρs σ κs _ ρ0)
 (ev e ρ ρs σ (cons (defk id) κs) #f ρ0))

((ko (defk id) v ρ ρs σ (cons κ κs))
 (match (assoc id ρ)
   ((cons _ a)
    (let ((σ* (cons (cons a v) σ)))
      (ko κ v ρ ρs σ* κs)))
   (_
    (let* ((fresh (gensym))
           (ρ* (cons (cons id fresh) ρ))
           (σ* (cons (cons fresh v) σ)))
      (ko κ v ρ* ρs σ* κs)))))
```

“EFKES” semantics for Slip

```
; evaluation

((ev `(evaluate ,e) ρ ρs σ κs τ ρ0)
 (ev e ρ ρs σ (cons (evalk τ ρ0) κs) τ ρ0))

((ko (evalk τ ρ0) v ρ ρs σ κs)
 (ev v ρ ρs σ κs τ ρ0))
```

“EFKES” semantics for Slip

```
; conditional

((ev `(if ,e ,e1) ρ ρs σ κs τ ρ0)
 (ev e ρ ρs σ (cons (ifk e1 τ ρ0) κs) #f ρ0))

((ko (ifk _ _ _) #f ρ ρs σ (cons κ κs))
 (ko κ (unspecified) ρ ρs σ κs))

((ko (ifk e1 τ ρ0) _ ρ ρs σ κs)
 (thk e1 ρ ρs σ κs τ ρ0))

; ---

((ev `(if ,e ,e1 ,e2) ρ ρs σ κs τ ρ0)
 (ev e ρ ρs σ (cons (ifek e1 e2 τ ρ0) κs) #f ρ0))

((ko (ifek _ e2 τ ρ0) #f ρ ρs σ κs)
 (thk e2 ρ ρs σ κs τ ρ0))

((ko (ifek e1 _ τ ρ0) _ ρ ρs σ κs)
 (thk e1 ρ ρs σ κs τ ρ0))
```

```
; thunk

(define (thk e ρ ρs σ κs τ ρ0)
  (if τ
      (ev e '() (cons ρ ρs) σ κs #t ρ0)
      (ev e '() (cons ρ ρs) σ (cons (retk ρ ρs) κs) #t ρ0)))
```

“EFKES” semantics for Slip

```
; abstraction

((ev `(lambda ,ps ,es ...) ρ ρs σ (cons κ κs) _ _)
 (let ((clo* (clo ps es (cons ρ ρs))))
   (ko κ clo* ρ ρs σ κs)))
```

“EFKES” semantics for Slip

```
; assignment

((ev `(set! ,id ,e) ρ ρs σ κs _ ρ0)
 (ev e ρ ρs σ (cons (setk id ρ0) κs) #f ρ0))

((ko (setk id ρ0) v ρ ρs σ (cons κ κs))
 (let loop ((ρ* ρ)
            (ρs* ρs))
   (match (assoc id ρ*)
     ((cons _ a)
      (ko κ v ρ ρs (cons (cons a v) σ) κs))
     (_
      (match ρs*
        ((cons ρ ρs)
         (loop ρ ρs))
        (_
         (rep (msg "variable not found: " id) σ ρ0)))))))
```

```
; error message with symbol

(define (msg ms id)
  (string-append ms (symbol->string id)))
```

```
; escape to read-eval-print

(define (rep ms ρ σ)
  (ko (repk) ms ρ '() σ '()))
```

“EFKES” semantics for Slip

```
; application

((ev `(,rator) ρ ρs σ κs τ ρ0)
 (ev rator ρ ρs σ (cons (nullk τ ρ0) κs) τ ρ0))

((ko (nullk τ ρ0) π ρ ρs σ κs)
 (ap π '() ρ ρs σ κs τ ρ0))

; ---

((ev `(,rator . ,rands) ρ ρs σ κs τ ρ0)
 (ev rator ρ ρs σ (cons (ratork rands τ ρ0) κs) #f ρ0))

((ko (ratork (list rand) τ ρ0) π ρ ρs σ κs)
 (ev rand ρ ρs σ (cons (singlek π τ ρ0) κs) τ ρ0))

((ko (singlek π τ ρ0) v ρ ρs σ κs)
 (ap π (list v) ρ ρs σ κs τ ρ0))

((ko (ratork (cons rand rands) τ ρ0) π ρ ρs σ κs)
 (ev rand ρ ρs σ (cons (randk π rands '() τ ρ0) κs) #f ρ0))

((ko (randk π (list rand) vs τ ρ0) v ρ ρs σ κs)
 (ev rand ρ ρs σ (cons (multiplek π (cons v vs) τ ρ0) κs) τ ρ0))

((ko (randk π (cons rand rands) vs τ ρ0) v ρ ρs σ κs)
 (ev rand ρ ρs σ (cons (randk π rands (cons v vs) τ ρ0) κs) #f ρ0))

((ko (multiplek π vs τ ρ0) v ρ ρs σ κs)
 (ap π (reverse (cons v vs)) ρ ρs σ κs τ ρ0))

; ---
```

```
((ap (clo ps es ρs#) vs ρ ρs σ κs τ ρ0)
 (let ((κs* (if τ κs (cons (retk ρ ρs) κs))))
   (let loop ((ps* ps)
              (vs* vs)
              (ρ# '())
              (σ* σ))
     (match (cons ps* vs*)
       ((cons '() '())
        (seq es ρ# ρs# σ* κs* #t ρ0))
       ((cons (cons p ps*) (cons v vs*))
        (let* ((fresh (gensym))
               (ρ# (cons (cons p fresh) ρ#))
               (σ* (cons (cons fresh v) σ*)))
          (loop ps* vs* ρ# σ*)))
       ((cons (and ps* (? symbol?)) vs*)
        (let* ((fresh (gensym))
               (ρ# (cons (cons ps* fresh) ρ#))
               (σ* (cons (cons fresh (list->mlist vs*)) σ*)))
          (seq es ρ# ρs# σ* κs* #t ρ0)))
       (_
        (rep "non-matching argument list" σ* ρ0))))))

((ap (con ρ ρs (cons κ κs)) (mlist v) _ _ σ _ _ ρ0)
 (ko κ v ρ ρs σ κs))

((ap (con _ _ _) _ _ _ σ _ _ ρ0)
 (rep "invalid arguments for: call/cc" σ ρ0))

((ap (nat π) vs ρ ρs σ (cons k κs) _ _)
 (ko k (apply π vs) ρ ρs σ κs))

((ko (retk ρ ρs) v _ _ σ (cons κ κs))
 (ko κ v ρ ρs σ κs))
```

“EFKES” semantics for Slip

```
; natives

 ((ap (apl) (mlist π vs) ρ ρs σ κs τ ρ0)
  (ap π vs ρ ρs σ κs τ ρ0))

 ((ap (ccc) (mlist π) ρ ρs σ κs τ ρ0)
  (ap π (mlist (con ρ ρs κs)) ρ ρs σ κs τ ρ0))

 ((ap (fch) (mlist π '()) ρ ρs σ κs _ ρ0)
  (ko (car κs) '() ρ ρs σ (cdr κs)))

 ((ap (fch) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

 ((ko (fchk π '() ρ0) _ ρ ρs σ (cons κ κs))
  (ko κ (unspecified) ρ ρs σ κs))

 ((ko (fchk π (mcons v vs) ρ0) _ ρ ρs σ κs)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

((ap (mpp) (mlist π '()) ρ ρs σ κs _ ρ0)
 (ko (car κs) '() ρ ρs σ (cdr κs)))

((ap (mpp) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs '() ρ0) κs) #f ρ0))

((ko (mppk π '() vs* _) v* ρ ρs σ (cons κ κs))
 (ko κ (mreverse (mcons v* vs*)) ρ ρs σ κs))

((ko (mppk π (mcons v vs) vs* ρ0) v* ρ ρs σ κs)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs (mcons v* vs*) ρ0) κs) #f ρ0))

((ap (nyi) _ _ _ σ _ _ ρ0)
 (rep "not yet implemented" σ ρ0))

((ap (rea) '() ρ ρs σ κs _ ρ0)
 (ko (car κs) (lit (read)) ρ ρs σ κs))

((ap _ _ _ _ σ _ _ ρ0)
 (rep "invalid application" σ ρ0))
```

“EFKES” semantics for Slip

```
; literal
((ev e ρ ρs σ (cons κ κs) _ _)
 (ko κ (lit e) ρ ρs σ κs))
```

```
; literal cloning

(define (lit e)
  (match e
    ((and e (? string?))
     (string-copy e))
    ((cons car cdr)
     (mcons (lit car) (lit cdr)))
    ((vector es ...)
     (list->vector (map lit es)))
    (_
     e)))
```

“EFKES” semantics for Slip

```
(define native-names '(-
                       *
                       /
                       +
                       <
                       <=
                       =
                       >
                       >=
                       abs
                       acos
                       ;angle
                       append
                       apply

                       . . .

                       false
                       format
                       gensym
                       random
                       true))
```

```
(define native-implementations (list (nat -)
                                     (nat *)
                                     (nat /)
                                     (nat +)
                                     (nat <)
                                     (nat <=)
                                     (nat =)
                                     (nat >)
                                     (nat >=)
                                     (nat abs)
                                     (nat acos)
                                     ;(nat angle)
                                     (nat append)

                                        . . .

                                     #f
                                     (nyi)
                                     (nyi)
                                     (nyi)
                                     #t))
```

```
(define ρ (map cons native-names native-names))
(define σ (map cons native-names native-implementations))

(let loop ((state (rep "Small-Step Slip" ρ σ)))
  (loop (step state)))
```

“EFKES” semantics for Slip

```
; natives

 ((ap (apl) (mlist π vs) ρ ρs σ κs τ ρ0)
  (ap π vs ρ ρs σ κs τ ρ0))

 ((ap (ccc) (mlist π) ρ ρs σ κs τ ρ0)
  (ap π (mlist (con ρ ρs κs)) ρ ρs σ κs τ ρ0))

 ((ap (fch) (mlist π '()) ρ ρs σ κs _ ρ0)
  (ko (car κs) '() ρ ρs σ (cdr κs)))

 ((ap (fch) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

 ((ko (fchk π '() ρ0) _ ρ ρs σ (cons κ κs))
  (ko κ (unspecified) ρ ρs σ κs))

 ((ko (fchk π (mcons v vs) ρ0) _ ρ ρs σ κs)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

((ap (mpp) (mlist π '()) ρ ρs σ κs _ ρ0)
 (ko (car κs) '() ρ ρs σ (cdr κs)))

((ap (mpp) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs '() ρ0) κs) #f ρ0))

((ko (mppk π '() vs* _) v* ρ ρs σ (cons κ κs))
 (ko κ (mreverse (mcons v* vs*)) ρ ρs σ κs))

((ko (mppk π (mcons v vs) vs* ρ0) v* ρ ρs σ κs)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs (mcons v* vs*) ρ0) κs) #f ρ0))

((ap (nyi) _ _ _ σ _ _ ρ0)
 (rep "not yet implemented" σ ρ0))

((ap (rea) '() ρ ρs σ κs _ ρ0)
 (ko (car κs) (lit (read)) ρ ρs σ κs))

((ap _ _ _ _ σ _ _ ρ0)
 (rep "invalid application" σ ρ0))
```

“EFKES” semantics for Slip

```
; natives

 ((ap (apl) (mlist π vs) ρ ρs σ κs τ ρ0)
  (ap π vs ρ ρs σ κs τ ρ0))

 ((ap (ccc) (mlist π) ρ ρs σ κs τ ρ0)
  (ap π (mlist (con ρ ρs κs)) ρ ρs σ κs τ ρ0))

 ((ap (fch) (mlist π '()) ρ ρs σ κs _ ρ0)
  (ko (car κs) '() ρ ρs σ (cdr κs)))

 ((ap (fch) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

 ((ko (fchk π '() ρ0) _ ρ ρs σ (cons κ κs))
  (ko κ (unspecified) ρ ρs σ κs))

 ((ko (fchk π (mcons v vs) ρ0) _ ρ ρs σ κs)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

((ap (mpp) (mlist π '()) ρ ρs σ κs _ ρ0)
 (ko (car κs) '() ρ ρs σ (cdr κs)))

((ap (mpp) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs '() ρ0) κs) #f ρ0))

((ko (mppk π '() vs* _) v* ρ ρs σ (cons κ κs))
 (ko κ (mreverse (mcons v* vs*)) ρ ρs σ κs))

((ko (mppk π (mcons v vs) vs* ρ0) v* ρ ρs σ κs)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs (mcons v* vs*) ρ0) κs) #f ρ0))

((ap (nyi) _ _ _ σ _ _ ρ0)
 (rep "not yet implemented" σ ρ0))

((ap (rea) '() ρ ρs σ κs _ ρ0)
 (ko (car κs) (lit (read)) ρ ρs σ κs))

((ap _ _ _ _ σ _ _ ρ0)
 (rep "invalid application" σ ρ0))
```

“EFKES” semantics for Slip

```
; natives

 ((ap (apl) (mlist π vs) ρ ρs σ κs τ ρ0)
  (ap π vs ρ ρs σ κs τ ρ0))

 ((ap (ccc) (mlist π) ρ ρs σ κs τ ρ0)
  (ap π (mlist (con ρ ρs κs)) ρ ρs σ κs τ ρ0))

 ((ap (fch) (mlist π '()) ρ ρs σ κs _ ρ0)
  (ko (car κs) '() ρ ρs σ (cdr κs)))

 ((ap (fch) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

 ((ko (fchk π '() ρ0) _ ρ ρs σ (cons κ κs))
  (ko κ (unspecified) ρ ρs σ κs))

 ((ko (fchk π (mcons v vs) ρ0) _ ρ ρs σ κs)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

((ap (mpp) (mlist π '()) ρ ρs σ κs _ ρ0)
 (ko (car κs) '() ρ ρs σ (cdr κs)))

((ap (mpp) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs '() ρ0) κs) #f ρ0))

((ko (mppk π '() vs* _) v* ρ ρs σ (cons κ κs))
 (ko κ (mreverse (mcons v* vs*)) ρ ρs σ κs))

((ko (mppk π (mcons v vs) vs* ρ0) v* ρ ρs σ κs)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs (mcons v* vs*) ρ0) κs) #f ρ0))

((ap (nyi) _ _ _ σ _ _ ρ0)
 (rep "not yet implemented" σ ρ0))

((ap (rea) '() ρ ρs σ κs _ ρ0)
 (ko (car κs) (lit (read)) ρ ρs σ κs))

((ap _ _ _ _ σ _ _ ρ0)
 (rep "invalid application" σ ρ0))
```

“EFKES” semantics for Slip

```
; natives

 ((ap (apl) (mlist π vs) ρ ρs σ κs τ ρ0)
  (ap π vs ρ ρs σ κs τ ρ0))

 ((ap (ccc) (mlist π) ρ ρs σ κs τ ρ0)
  (ap π (mlist (con ρ ρs κs)) ρ ρs σ κs τ ρ0))

 ((ap (fch) (mlist π '()) ρ ρs σ κs _ ρ0)
  (ko (car κs) '() ρ ρs σ (cdr κs)))

 ((ap (fch) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

 ((ko (fchk π '() ρ0) _ ρ ρs σ (cons κ κs))
  (ko κ (unspecified) ρ ρs σ κs))

 ((ko (fchk π (mcons v vs) ρ0) _ ρ ρs σ κs)
  (ap π (mlist v) ρ ρs σ (cons (fchk π vs ρ0) κs) #f ρ0))

((ap (mpp) (mlist π '()) ρ ρs σ κs _ ρ0)
 (ko (car κs) '() ρ ρs σ (cdr κs)))

((ap (mpp) (mlist π (mcons v vs)) ρ ρs σ κs _ ρ0)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs '() ρ0) κs) #f ρ0))

((ko (mppk π '() vs* _) v* ρ ρs σ (cons κ κs))
 (ko κ (mreverse (mcons v* vs*)) ρ ρs σ κs))

((ko (mppk π (mcons v vs) vs* ρ0) v* ρ ρs σ κs)
 (ap π (mlist v) ρ ρs σ (cons (mppk π vs (mcons v* vs*) ρ0) κs) #f ρ0))

((ap (nyi) _ _ _ σ _ _ ρ0)
 (rep "not yet implemented" σ ρ0))

((ap (rea) '() ρ ρs σ κs _ ρ0)
 (ko (car κs) (lit (read)) ρ ρs σ κs))

((ap _ _ _ _ σ _ _ ρ0)
 (rep "invalid application" σ ρ0))
```