Interpreting `call/cc`
The CEK machine

CIS400 (Compiler Construction)

Kris Micinski, Fall 2021

- `call/cc` is a very powerful control construct
  - Can use it to implement exceptions, coroutines, etc…
- Implementing `call/cc` requires that we be able to materialize continuations as values at runtime
  - Just as lambdas require us to represent closures at runtime
  - Also need to handle continuation invocation

Normal Racket exceptions…

`with-handlers` adds an exception handler

```
(define (my-exception? e)
  (match e
    [`(my-exception ,(? number? n)) #t]
    [_ #f]))

(with-handlers ([my-exception?
                 (match-lambda [`(my-exception ,n) (displayln n)])])
    (+ 5 (raise `(my-exception 8))))
```

Within evaluation of body, exceptions may be `raise`d

Exceptions (one encoding) via `call/cc`

```
(define (my-exception? e)
  (match e
    [`(my-exception ,(? number? n)) #t]
    [_ #f]))

(with-handlers ([my-exception?
                 (match-lambda [`(my-exception ,n) (displayln n)])])
    (+ 5 (raise `(my-exception 8))))
```

- First, wrap entire thing in call/cc to get an "outer" continuation

```
(call/cc
 (lambda (k)




    ))
```

Exceptions via `call/cc`

```
(define (my-exception? e)
  (match e
    [`(my-exception ,(? number? n)) #t]
    [_ #f]))

(with-handlers ([my-exception?
                 (match-lambda [`(my-exception ,n) (displayln n)])])
    (+ 5 (raise `(my-exception 8))))
```

- First, wrap entire thing in call/cc to get an "outer" continuation
- Next, wrap body in call/cc and a match
  - Match will match the return value, either answer or exception

```
(call/cc
 (lambda (k)
   (match
       (call/cc
        (lambda (raise-exception)
          (+ 5 (raise-exception `(my-exception 8)))))
     [`(my-exception ,n) (displayln n)]
     ;; any non-exception value, terminate "normally"
     [anything-else (k anything-else)])))
```

- This is just one illustrative encoding I made up.
- You could also, say, take all “normal” return points and add a call to k (the original continuation)
- Broad point: continuations ala call/cc (or some other primitive control operator) add a lot of expressivity we want

```
(call/cc
 (lambda (k)
   (match
       (call/cc
        (lambda (raise-exception)
          (k (+ 5 (raise-exception `(my-exception 8))))))
     [`(my-exception ,n) (displayln n)])))
```

Continuations are very useful and a great foundation for control operators in our language

However, now we need to **implement** them

The next project will be having you implement call/cc by compiling **everything** to a specific style named CPS (we will soon talk about CPS and see why it is useful)

In today's lecture we will build an **interpreter** for LC + call/cc

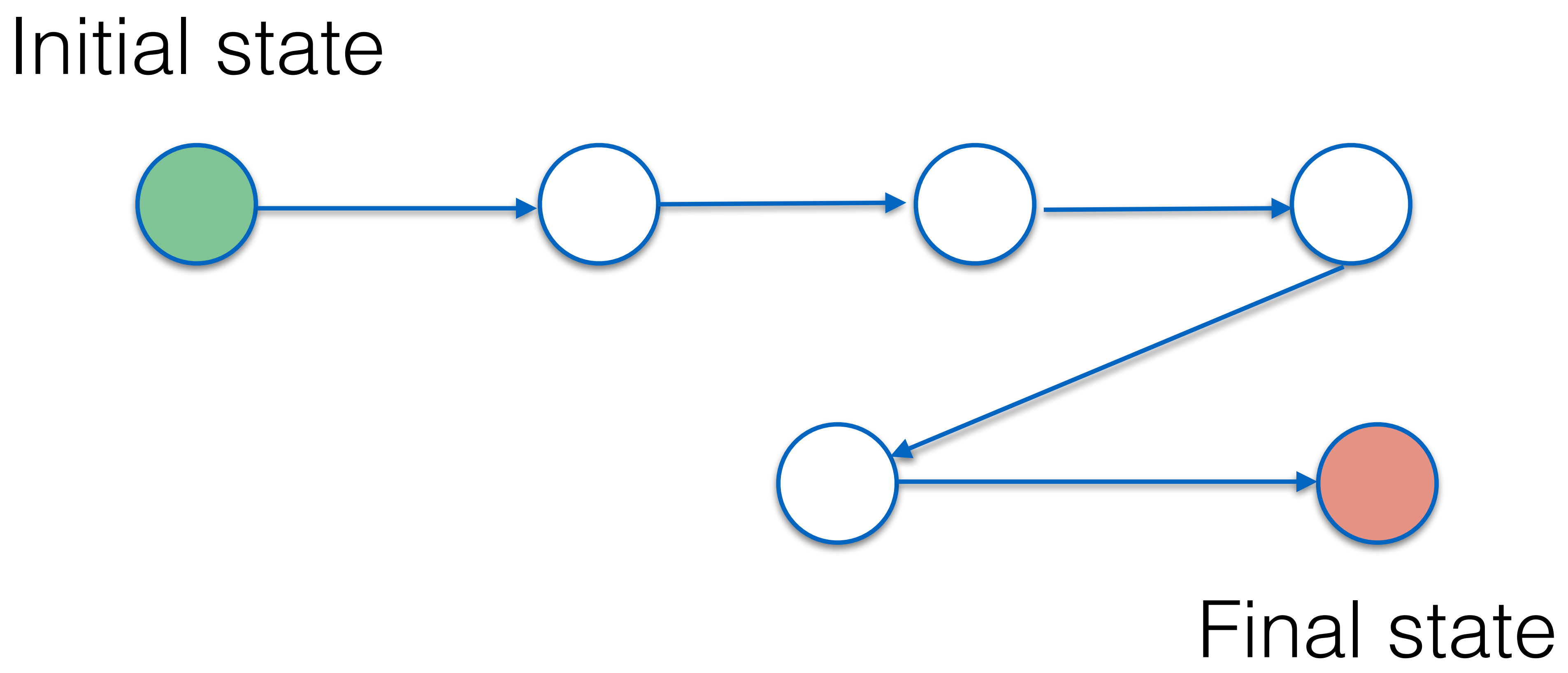
Stack-passing (CEK) semantics

(implementing first-class continuations)

Abstract Machine Semantics

One common semantics (we have been touching upon) is the **abstract machine** style

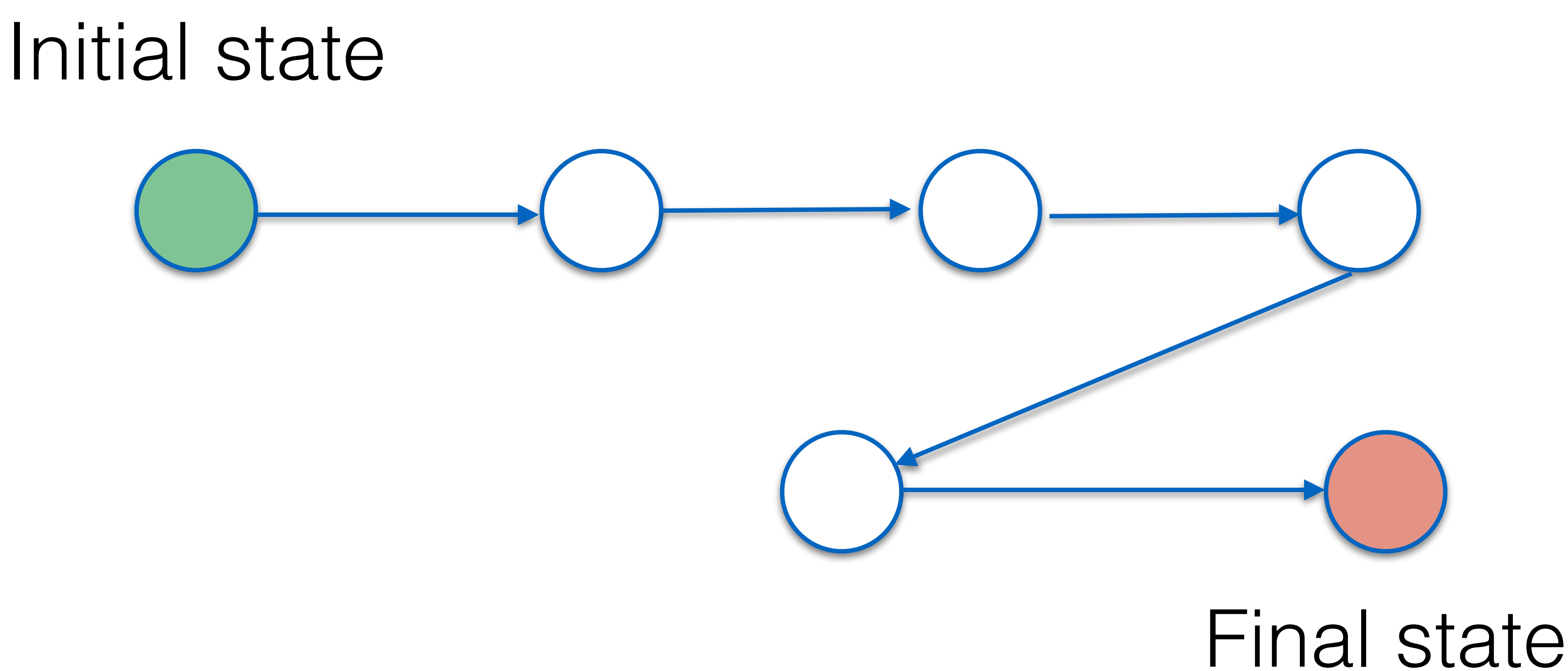
In style style of semantics, we define an “abstract machine” (like a VM) that takes a sequence of *steps* to compute an answer



Abstract Machine Semantics

To define an abstract machine, we must specify:

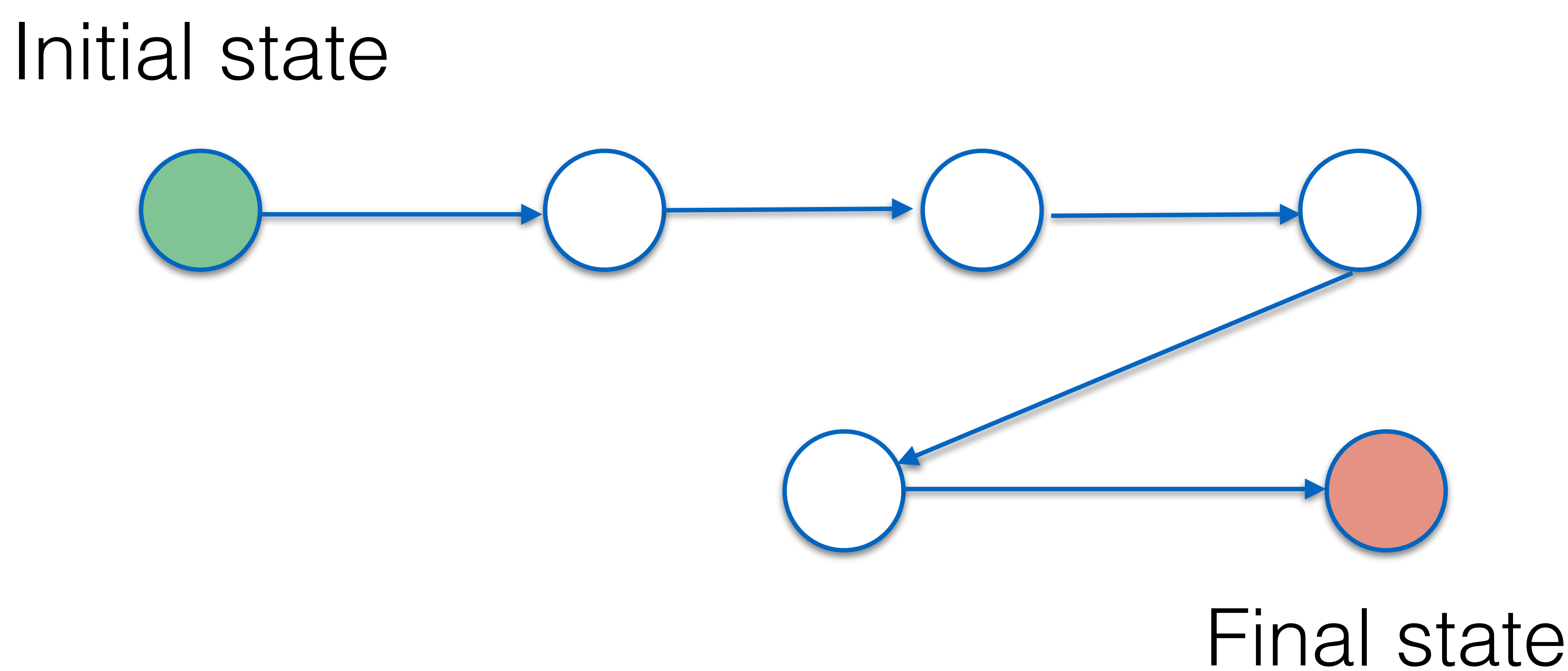
- The type (i.e., structure) of abstract machine **states**
- A **step relation** tells us how one state proceeds to next
  - Often this will be a **function** (rather than a relation, in which case semantics may be nondeterministic)
- How to *inject* a program into an **initial** state
- What **final** states look like



Abstract Machine Semantics

To define an abstract machine, we must specify:

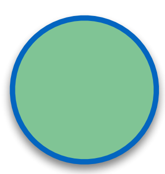
- The type (i.e., structure) of abstract machine **states**
- A **step relation** tells us how one state proceeds to next
  - Often this will be a **function** (rather than a relation, in which case semantics may be nondeterministic)
- How to *inject* a program into an **initial** state
- What **final** states look like



“Running” the machine

Is simply the transitive closure of the step function (or, if step function is a relation, iteration to a fixed point of a state graph)

Initial state



“Running” the machine

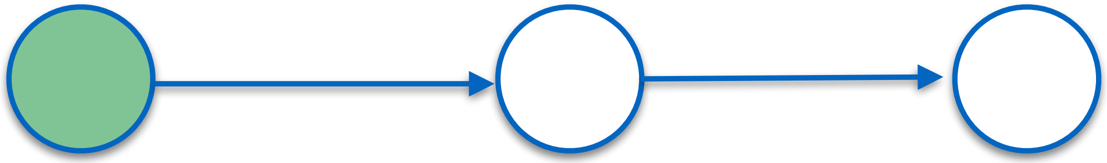
Is simply the transitive closure of the step function (or, if step function is a relation, iteration to a fixed point of a state graph)

Initial state

“Running” the machine

Is simply the transitive closure of the step function (or, if step function is a relation, iteration to a fixed point of a state graph)

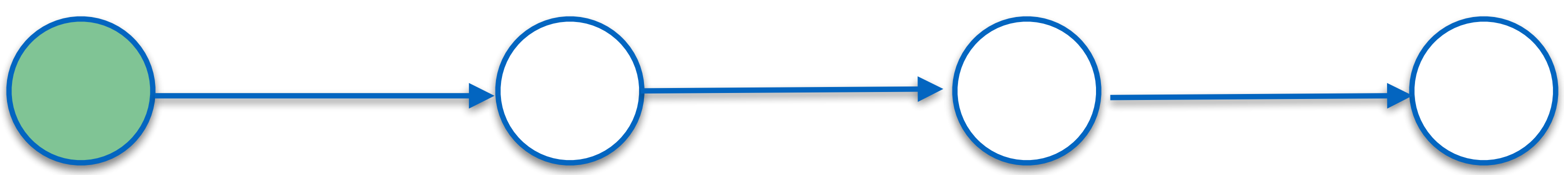
Initial state



“Running” the machine

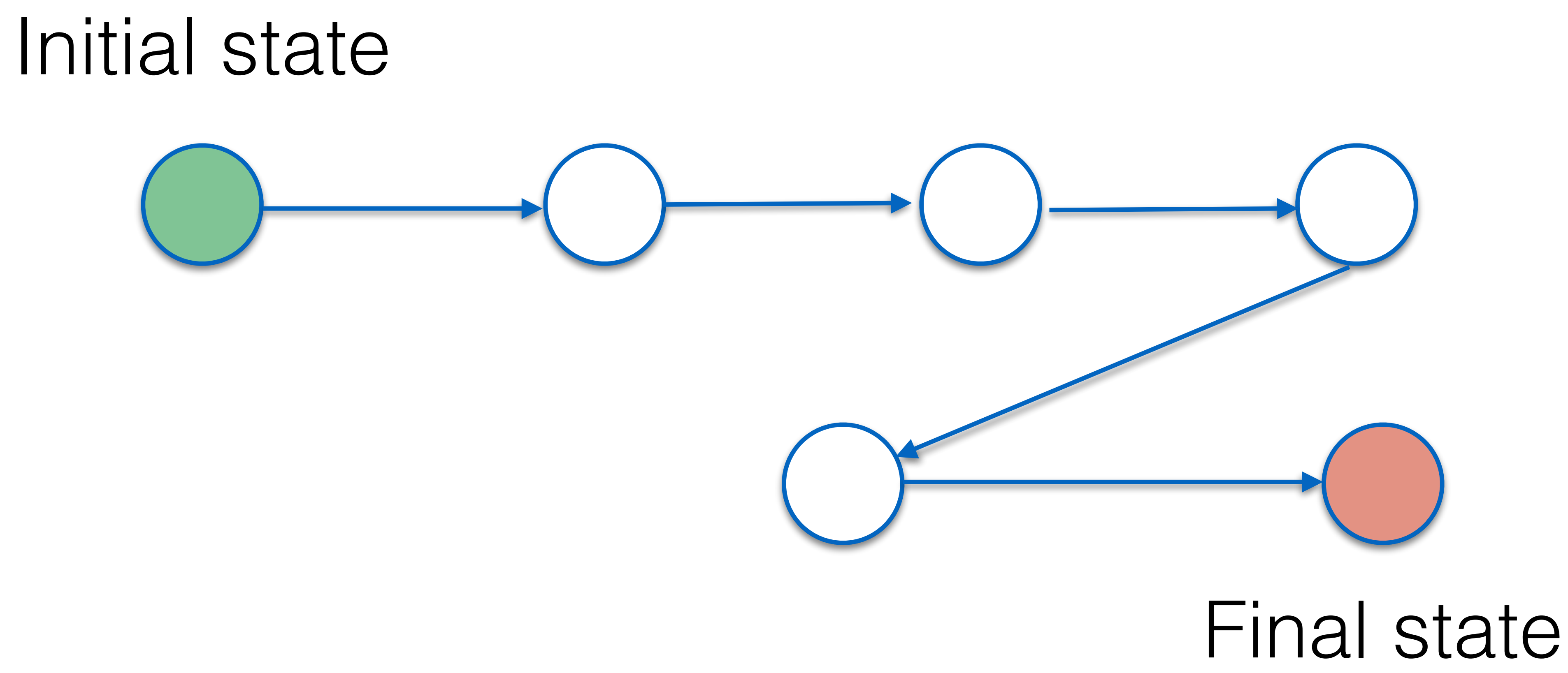
Is simply the transitive closure of the step function (or, if step function is a relation, iteration to a fixed point of a state graph)

Initial state



“Running” the machine

Is simply the transitive closure of the step function (or, if step function is a relation, iteration to a fixed point of a state graph)



Abstract Machines in Racket

```
(define (expr? e)

(define (state? s) 'todo)
```

```
(define/contract (step s)
  (-> state? state?)
  'todo)

(define/contract (inject s)
  (-> state? state?)
  'todo)
```

C Control-expression

Term-rewriting / textual reduction
Context and redex for deterministic eval

CE Control & Env machine

Big-step, explicit closure creation

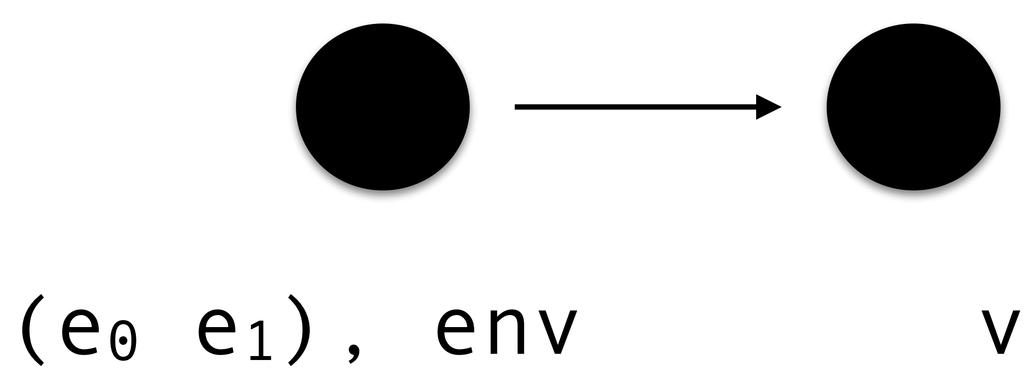
CES Store-passing machine

Passes addr->value map in evaluation order

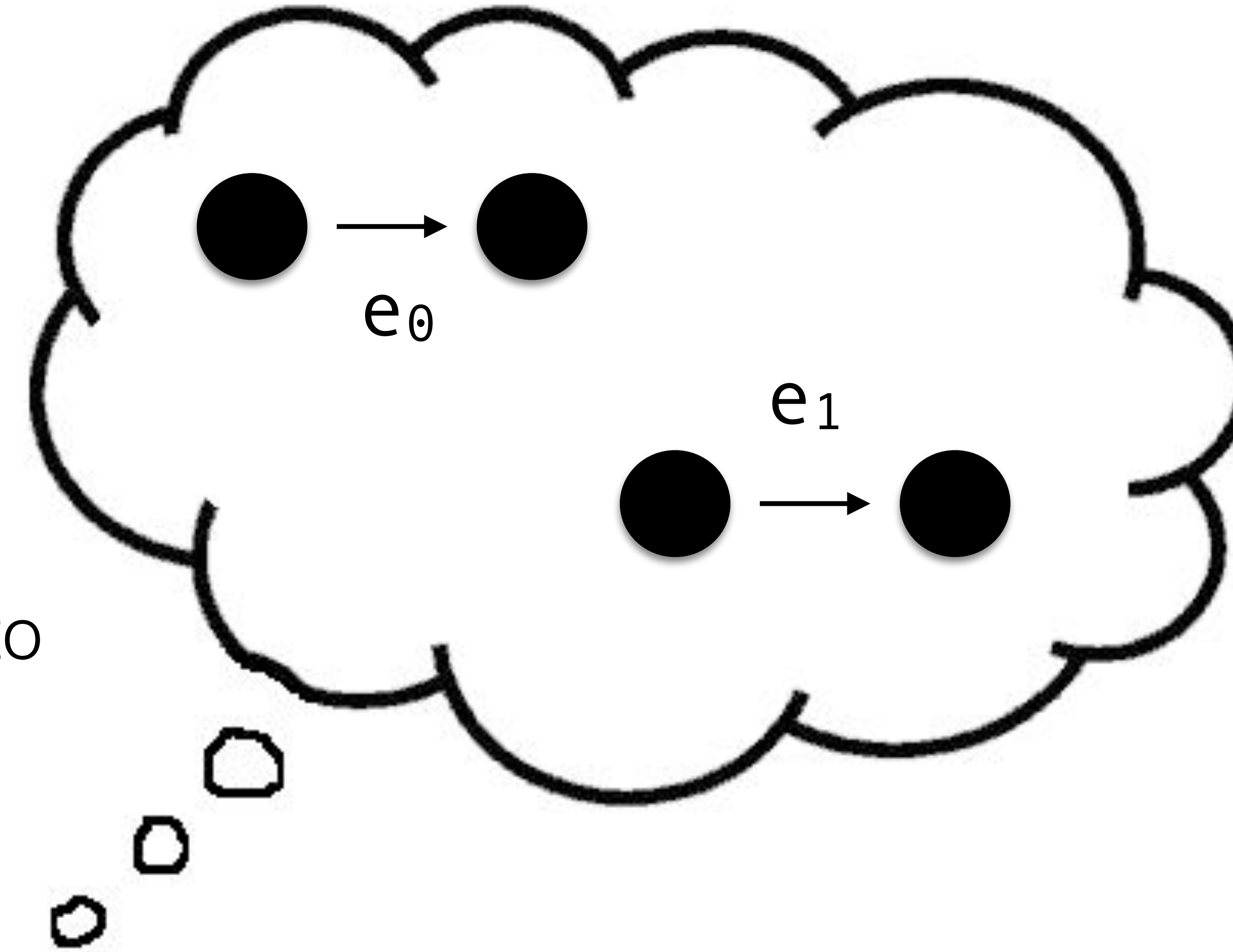
CEK Stack-passing machine

Passes a list of stack frames, small-step

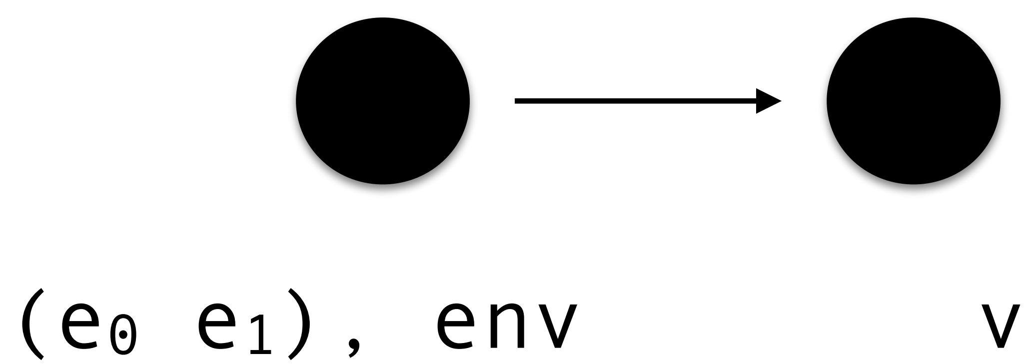
Previously…



Previously…



To define the semantics, we need to know what e_0 and e_1 evaluate to



```
(define (interp e env)
  (match e
         [(? symbol? x)
          (hash-ref env x)]

         [`(λ (,x) ,e0)
          `(clo (λ (,x) ,e0) ,env)]

         [`(,e0 ,e1)
          (define v0 (interp e0 env))
          (define v1 (interp e1 env))
          (match v0
            [`(clo (λ (,x) ,e2) ,env)
             (interp e2 (hash-set env x v1))])]))
```

In our previous interpreter, the *interpreter itself* is recursive. `interp` uses Racket's stack!

```
(interp ‘(prim + (prim * 3 2) 5) env)
 > (interp (prim * 3 2) env)
   > (interp 3)
   > 3
   > (interp 2)
   < 2
 < 6
 > (interp 5 env)
 < 5
< 11
```

Nested calls (to interp) in the interpreter then form nested call stacks in Racket

We will add `call/cc` to the language

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

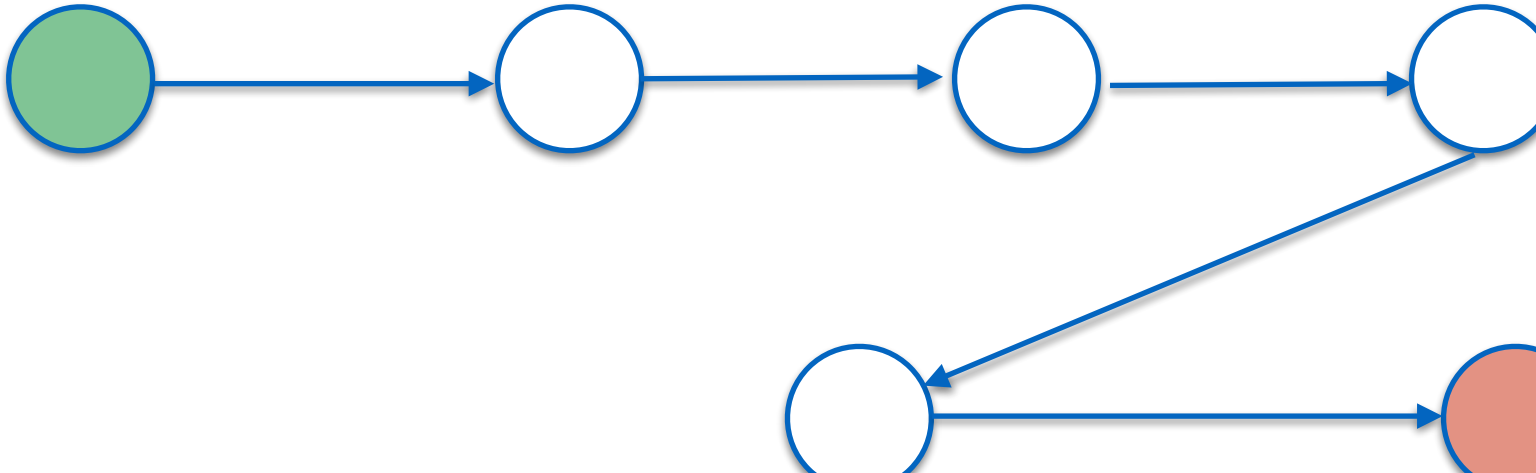
Our interpreter will **explicitly** represent a stack

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

CEK machine

Control, **E**nvironment, **K**ontinuation

- **States** are (c, env, k) where…
  - c (control) is an expression
  - env (environment) is a map from variables to values
  - k is a continuation (representation of the stack)
    - We will define k structurally in the next slide
- We define the step function in the next few slides
- The initial state for a program e is…
  - (e, [], Done) where e is the program, [] is the empty env, and Done is a special “Done” continuation
- A state is **final** when it has no next step and the continuation is “Done”



The lambda calculus already has a bit of lurking complexity in it: evaluating the function **and** argument position require an *unbounded* amount of work…

If all arguments were **atomic**, defining the semantics gets simpler (we know arguments can be immediately evaluated)—this is called ANF (administrative normal form) and we will cover it shortly.

```
((lambda (x) x) (lambda (y) y))
```

- Both arguments atomic (can be evaluated immediately)

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z))
```

- First argument is *not* atomic

```
((((lambda (x) x) (lambda (y) y)) (lambda (z) z))
 (((lambda (a) a) (lambda (b) b)) (lambda (c) c)))
```

- Neither argument atomic

Continuations for CEK

When the CEK machine encounters a callsite, it pushes a frame to the stack (continuation) to remember to go back and evaluate the argument.

When evaluation of the function position is complete, the machine switches to evaluating the arguments, but needs to remember to then apply the (computed) closure

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z)), env = {}, k = Done
```

Start to evaluate fn position

Remember to eval arg.

```
((lambda (x) x) (lambda (y) y)), env = {}, k = Ar<(lambda (z) z), {}, Done>
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z)), env = {}, k = Done
```

Start to evaluate fn position

Remember to eval arg.

```
((lambda (x) x) (lambda (y) y)), env = {}, k = Ar<(lambda (z) z),{},Done>
```

Continue to evaluate fn yet again

Remember to eval *this one's* argument

```
(lambda (x) x), env = {}, k = Ar<(lambda (y) y), Ar<(lambda (z) z),{},Done>>
```

Here we are done, this evals to a closure clo<(lambda (x) x),[]>

```
(lambda (y) y), env = {}, k = Fn<clo<(lambda (x) x),[]), Ar<(lambda (z) z),{},Done>>
```

So we **swap** from evaluating the function (of the purple thing) to evaluating its **argument** (the red thing), **saving** the closure (to remember to apply later)

Also, we remember that later on we need to do Ar<(lambda (z) z), Done>

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z)), env = {}, k = Done
```

Start to evaluate fn position

Remember to eval arg.

```
((lambda (x) x) (lambda (y) y)), env = {}, k = Ar<(lambda (z) z),{},Done>
```

Continue to evaluate fn yet again

Remember to eval *this one's* argument

```
(lambda (x) x), env = {}, k = Ar<(lambda (y) y), Ar<(lambda (z) z),{},Done>>
```

Here we are done, this evals to a closure `clo<(lambda (x) x),[]>`

```
(lambda (y) y), env = {}, k = Fn<clo<(lambda (x) x),[]), Ar<(lambda (z) z),{},Done>>
```

So we **swap** from evaluating the function (of the purple thing) to evaluating its **argument** (the red thing), **saving** the closure (to remember to apply later)

This is **also** a value, so we build a closure again, but this time we **apply** the saved closure

We do this by swapping into the stored environment [] extended with a binding for **x**

And then stepping to the **body** of the stored closure…

```
x, env = {x |-> clo<(lambda (x) x),[])}, Ar<(lambda (z) z),{},Done>>
```

Also, now that we're in the body, we **pop** the stack to the previous frame

We do this by swapping into the stored environment [] extended with a binding for **x**

And then stepping to the **body** of the stored closure…

```
x, env = {x |-> clo<(lambda (x) x),[])}, Ar<(lambda (z) z),{},Done>>
```

Also, now that we're in the body, we **pop** the stack to the previous frame

x is a value, so we look it up in the current env, it is a closure. We swap to evaluating the body of the argument (i.e., the orange thing), substituting z (its argument) with the value of x (the closure we just looked up)

```
z, env = {z |-> clo<(lambda (x) x),[])}, Done>
```

Note that the stack shrinks back to just Done

We're now in an accepting state: there's only a value being popped to Done. If we coded up our semantics to give us an answer, this is the point at which we'd have it.

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z)), env = {}, k = Done
```

Let's think about this another way: watching the program and stack change over time (representing stack visually)

Portion of program being evaluated	Stack
`(((lambda (x) x) (lambda (y) y)) (lambda (z) z))`	Done

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z)), env = {}, k = Done
```

```
((lambda (x) x) (lambda (y) y)), env = {}, k = Ar<(lambda (z) z), Done>
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z))
```

```
Ar<(lambda (z) z)
, … >
```

```
Done
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z)), env = {}, k = Done
```

```
((lambda (x) x) (lambda (y) y)), env = {}, k = Ar<(lambda (z) z), Done>
```

```
(lambda (x) x), env = {}, k = Ar<(lambda (y) y), Ar<(lambda (z) z), Done>>
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z))
```

```
Ar<(lambda (y) y)
, … >

Ar<(lambda (z) z)
, … >

Done
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z)), env = {}, k = Done
```

```
((lambda (x) x) (lambda (y) y)), env = {}, k = Ar<(lambda (z) z), Done>
```

```
(lambda (x) x), env = {}, k = Ar<(lambda (y) y), Ar<(lambda (z) z), Done>>
```

```
(lambda (y) y), env = {}, k = Fn<clo<(lambda (x) x),[]), Ar<(lambda (z) z), Done>>
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z))
```

```
Fn<clo(lambda (x)
x), [], …>

Ar<(lambda (z) z)
, … >

Done
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z)), env = {}, k = Done
```

```
((lambda (x) x) (lambda (y) y)), env = {}, k = Ar<(lambda (z) z), Done>
```

```
(lambda (x) x), env = {}, k = Ar<(lambda (y) y), Ar<(lambda (z) z), Done>>
```

```
(lambda (y) y), env = {}, k = Fn<clo<(lambda (x) x),[]), Ar<(lambda (z) z), Done>>
```

```
x, env = {x |-> clo<(lambda (x) x),[])}, Ar<(lambda (z) z),{},Done>>
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z))
```

```
Ar<(lambda (z) z)
, … >

Done
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z)), env = {}, k = Done
```

```
((lambda (x) x) (lambda (y) y)), env = {}, k = Ar<(lambda (z) z), Done>
```

```
(lambda (x) x), env = {}, k = Ar<(lambda (y) y), Ar<(lambda (z) z), Done>>
```

```
(lambda (y) y), env = {}, k = Fn<clo<(lambda (x) x),[]), Ar<(lambda (z) z), Done>>
```

```
x, env = {x |-> clo<(lambda (x) x),[])}, Ar<(lambda (z) z),{},Done>>
```

```
z, env = {z |-> clo<(lambda (x) x),[])}, Done>
```

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z))
```

```
Done
```

k ::= **halt** | **ar**(e, env, k)
| **fn**(v, k)

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= **halt** | **ar**(e, env, k)
| **fn**(v, k)

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

$$\mathscr{E} ::= (\mathscr{E}\ e) \mid (v\ \mathscr{E}) \mid \square$$

Our interpreter will also include atoms

If we assume arguments to forms (e.g., `prim`) are atoms, we can define their semantics without additional continuations

```
;; atomically-evaluable-expressions
(define (atom? a)
  (match a
    [`(lambda (,x) ,e) #t]
    [(? number? n) #t]
    [(? symbol? x) #t]
    [_ #f]))

(define (expr? e)
  (match e
    [`(lambda (,(? symbol? x)) ,(? expr? e)) #t]
    [(? number? n) #t]
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    [`(prim ,prim ,(? atom? x) ,(? atom? y)) #t]
    [(? symbol? x) #t]
    [`(call/cc ,(? expr? e)) #t]
    [_ #f]))

(define environment? hash?)
```

```
(define (value? v)
  (match v
    [(? number? n) #t] ;; numeric constants
    [`(clo ,(? expr? e) ,(? environment? env)) #t]
    ;; very important: now continuations can be values as well!
    ;; means we have to be able to *apply* them
    [(? continuation? k) #t]
    [_ #f]))


(define (state? s)
  (match s
    [`(,(? expr? c) ,(? environment? env) ,(? continuation? k)) #t]
    [_ #f]))

(define (continuation? k)
  (match k
    [`(ar ,(? expr? e) ,(? environment? env) ,(? continuation? k)) #t]
    [`(fn ,(? value? v) ,(? continuation? k)) #t]
    ['done #t]
    [_ #f]))
```

$$((e_0\ e_1), env, k) \rightarrow (e_0, env, \mathbf{ar}(e_1, env, k))$$

$$(x, env, \mathbf{ar}(e_1, env_1, k_1)) \rightarrow (e_1, env_1, \mathbf{fn}(env(x), k_1))$$

$$((\lambda\ (x)\ e), env, \mathbf{ar}(e_1, env_1, k_1)) \rightarrow (e_1, env_1, \mathbf{fn}(((\lambda\ (x)\ e), env), k_1))$$

$$(x, env, \mathbf{fn}(((\lambda\ (x_1)\ e_1), env_1), k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$((\lambda\ (x)\ e), env, \mathbf{fn}(((\lambda\ (x_1)\ e_1), env_1), k_1)) \rightarrow (e_1, env_1[x_1 \mapsto ((\lambda\ (x)\ e), env)], k_1)$$

call/cc semantics

((call/cc (λ (x) e_0)), env, k) → (e_0, env[x ↦ k], k)

((λ (x) e_0), env, **fn**(k_0, k_1)) → ((λ (x) e_0), env, k_0)

(x, env, **fn**(k_0, k_1)) → (x, env, k_0)

$$e ::= \ldots \mid \texttt{(let ([x } e_0\texttt{]) } e_1\texttt{)}$$

$$k ::= \ldots \mid \textbf{let}(x, e, env, k)$$

$$(x, env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$(\texttt{(}\lambda\texttt{ (x) e)}, env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto (\texttt{(}\lambda\texttt{ (x) e)}, env)], k_1)$$

e ::= ...

| (let ([x e0]) e1)

$$(\texttt{x}, env, \textbf{fn}((\texttt{(λ (x}_1\texttt{) e}_1\texttt{)}, env_1), k_1)) \rightarrow (\texttt{e}_1, env_1[x_1 \mapsto env(\texttt{x})], k_1)$$

$$(\texttt{(λ (x) e)}, env, \textbf{fn}((\texttt{(λ (x}_1\texttt{) e}_1\texttt{)}, env_1), k_1)) \rightarrow (\texttt{e}_1, env_1[x_1 \mapsto (\texttt{(λ (x) e)}, env)], k_1)$$

k ::= ... | **let**(x, e, env, k)

These are nearly identical because a let form is just an immediate application of a lambda!

$$(\texttt{x}, env, \textbf{let}(x_1, \texttt{e}_1, env_1, k_1)) \rightarrow (\texttt{e}_1, env_1[x_1 \mapsto env(\texttt{x})], k_1)$$

$$(\texttt{(λ (x) e)}, env, \textbf{let}(x_1, \texttt{e}_1, env_1, k_1)) \rightarrow (\texttt{e}_1, env_1[x_1 \mapsto (\texttt{(λ (x) e)}, env)], k_1)$$

```
;; create an initial state
(define/contract (inject e)
  (-> expr? state?)
  `(,e ,(hash) done))
```

```
;; evaluate an atomic value (mixes in aspects of ANF)
(define (eval-atomic a env)
  (match a
    [`(lambda (,x) ,e) `(clo ,a ,env)]
    [(? number? n) n]
    [(? symbol? x) (hash-ref env x)]))
```

```
(define/contract (step s)
  (-> state? state?)
  (define (op->fn op) (match op ['+ +] ['- -] ['* *] ['/ /]))
  …
```

First, I define a helper function, (handle-return v k) that returns a value? to a continuation k

```
(define/contract (step s)
  (-> state? state?)
  (define (op->fn op) (match op ['+ +] ['- -] ['* *] ['/ /]))
  ;; How to handle the return of a value to a continuation
  (define (handle-return value k)
    (match k
      ;; switch to a fn frame
      [`(ar ,e-next ,env ,k-next)
       `(,e-next ,env (fn ,value ,k-next))]
      [`(fn ,function ,k-next)
       ;; handle an apply
       (match function
         [`(clo (lambda (,x) ,e-body) ,env+)
          `(,e-body ,(hash-set env+ x value) ,k-next)])]
      ['done (raise `(done ,value))]))
```

We will **raise** an exception when we’re done

(A bit of a hack for various reasons)

```
(define/contract (step s)
  (-> state? state?)
  … ;; op->fn and (handle-return v k)
  (match s
    ;; assumes that
    [`((call/cc (lambda (,x) ,e)) ,env ,k)
     `(,e ,(hash-set env x k) ,k)]
    [`(,e ,env (fn ,(? continuation? k0) ,k1))
     ;; switch to k0, continue to evaluate as normal.
     `(,e ,env ,k0)]
    [`(,(? number? n) ,env ,k) (handle-return n k)]
    ;; assume each argument is a variable
    [`((prim ,op ,a0 ,a1) ,env ,k)
     (define v0 (eval-atomic a0 env))
     (define v1 (eval-atomic a1 env))
     (handle-return ((op->fn op) v0 v1) k)]
    ;; push ar frame
    [`((,e0 ,e1) ,env ,k) `(,e0 ,env (ar ,e1 ,env ,k))]
    [`((lambda (,x) ,e) ,env ,k)
      (handle-return `(clo (lambda (,x) ,e) ,env) k)]
    [`(,(? symbol? x) ,env ,k) (handle-return (hash-ref env x) k)]))
```

CEK-machine evaluation

$(e_0, [], ()) \rightarrow \ldots$

$\rightarrow \ldots$

$\rightarrow \ldots$

$\rightarrow \ldots$

$\rightarrow$ (x, env, **halt**) $\rightarrow$ env(x)

$(e_0, [], ()) \to \ldots$

$\to \ldots$

$\to \ldots$

$\to \ldots$

$\to (x, env, \textbf{halt}) \to env(x)$

```
(define (done? d) (match d [`(done ,(? value? v)) #t] [_ #f]))

(define (trace-derivation e)
  (define (step* state)
    (with-handlers
        ([done? (match-lambda
                  [`(done ,v)
                   (displayln (format "~a (final result)" v))])])
      (define next-state (step state))
      (pretty-print state)
      (displayln "-->")
      (step* next-state)))
  (step* (inject e)))
```

```
> (trace-derivation '((lambda (x) (prim * x 3)) (call/cc (lambda (k) (k (prim + 2
3))))))
'(((lambda (x) (prim * x 3)) (call/cc (lambda (k) (k (prim + 2 3)))))
  #hash()
  done)
-->
'((lambda (x) (prim * x 3))
  #hash()
  (ar (call/cc (lambda (k) (k (prim + 2 3)))) #hash() done))
-->
'((call/cc (lambda (k) (k (prim + 2 3))))
  #hash()
  (fn (clo (lambda (x) (prim * x 3)) #hash()) done))
-->
'((k (prim + 2 3))
  #hash((k . (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
  (fn (clo (lambda (x) (prim * x 3)) #hash()) done))
-->
'(k
  #hash((k . (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
  (ar
   (prim + 2 3)
   #hash((k . (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
   (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
-->
'((prim + 2 3)
  #hash((k . (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
```

```
  (fn (clo (lambda (x) (prim * x 3)) #hash()) done))
-->
'((k (prim + 2 3))
  #hash((k . (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
  (fn (clo (lambda (x) (prim * x 3)) #hash()) done))
-->
'(k
  #hash((k . (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
  (ar
   (prim + 2 3)
   #hash((k . (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
   (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
-->
'((prim + 2 3)
  #hash((k . (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
  (fn
   (fn (clo (lambda (x) (prim * x 3)) #hash()) done)
   (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
-->
'((prim + 2 3)
  #hash((k . (fn (clo (lambda (x) (prim * x 3)) #hash()) done)))
  (fn (clo (lambda (x) (prim * x 3)) #hash()) done))
-->
15 (final result)
```