Evaluation

`1`

exp= `1`
env= `{}`

Done!

Evaluation

`+(1, 2)`

exp= `+(1, 2)`
env= `{}`

exp= `1`
env= `{}`

Done?

Evaluation

`+(1, 2)`

exp= `+(1, 2)`
env= `{}`

exp= `1`
env= `{}`

exp= `2`
env= `{}`

How do we know when we’re done?

How do we know what’s left to do?

Evaluation with To-Do List

`1`

exp= `1`
env= `{}`
todo= `[done]`

- Keep a to-do list, passed to evaluator

Evaluation with To-Do List

```
1
```

```
exp= 1
env= {}
todo= [done]
```

```
val= 1
todo= [done]
```

- When we get a value, go into to-do-checking mode

Evaluation with To-Do List

```
1
```

```
exp= 1
env= {}
todo= [done]
```

```
val= 1
todo= [done]
```

Done!

Evaluation with To-Do List

```
+(1, 2)
```

```
exp= +(1, 2)
env= {}
todo= [done]
```

```
exp= 1
env= {}
todo= [addexp 2 in {} then [done]]
```

- When evaluating sub-expressions, extend the to-do list
- **`addexp`** is an abbreviation for:

remember the result, evaluate another expression,
then add the two results

Evaluation with To-Do List

```
+(1, 2)
```

```
exp= +(1, 2)
env= {}
todo= [done]
```

```
exp= 1
env= {}
todo= [addexp 2 in {} then [done]]
```

```
val= 1
todo= [addexp 2 in {} then [done]]
```

Evaluation with To-Do List

```
val= 1
todo= [addexp 2 in {} then [done]]
```

```
exp= 2
env= {}
todo= [addval 1 then [done]]
```

- To do **addexp**, we start evaluating the remembered expression in the remembered environment
- Extend to-do list to remember the value we already have, and remember to do an addition later
- **addval** is an abbreviation for:

add the result with a remembered result

Evaluation with To-Do List

```
val= 1
todo= [addexp 2 in {} then [done]]
```

```
exp= 2
env= {}
todo= [addval 1 then [done]]
```

```
val= 2
todo= [addval 1 then [done]]
```

```
val= 3
todo= [done]
```

Done!

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
exp= +(1, +(2, 3))
env= {}
todo= [done]
```

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
exp= +(1, +(2, 3))
env= {}
todo= [done]
```

```
exp= 1
env= {}
todo= [addexp +(2, 3) in {} then [done]]
```

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
exp= 1
env= {}
todo= [addexp +(2, 3) in {} then [done]]

val= 1
todo= [addexp +(2, 3) in {} then [done]]
```

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
val= 1
todo= [addexp +(2, 3) in {} then [done]]

exp= +(2, 3)
env= {}
todo= [addval 1 then [done]]
```

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
exp= +(2, 3)
env= {}
todo= [addval 1 then [done]]

exp= 2
env= {}
todo= [addexp 3 in {} then [addval 1 then [done]]]
```

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
exp= 2
env= {}
todo= [addexp 3 in {} then [addval 1 then [done]]]

val= 2
todo= [addexp 3 in {} then [addval 1 then [done]]]
```

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
val= 2
todo= [addexp 3 in {} then [addval 1 then [done]]]
```

```
exp= 3
env= {}
todo= [addval 2 then [addval 1 then [done]]]
```

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
exp= 3
env= {}
todo= [addval 2 then [addval 1 then [done]]]
```

```
val= 3
todo= [addval 2 then [addval 1 then [done]]]
```

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
val= 3
todo= [addval 2 then [addval 1 then [done]]]
```

```
val= 5
todo= [addval 1 then [done]]
```

Evaluation with To-Do List

```
+(1, +(2, 3))
```

```
val= 5
todo= [addval 1 then [done]]
```

```
val= 6
todo= [done]
```

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

```
exp= let f = proc(y)y in (f 10)
env= {}
todo= [done]
```

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

```
exp= let f = proc(y)y in (f 10)
env= {}
todo= [done]

exp= proc(y)y
env= {}
todo= [let f in (f 10) {} then [done]]
```

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

```
exp= proc(y)y
env= {}
todo= [let f in (f 10) {} then [done]]

val= <y,y,{}>
todo= [let f in (f 10) {} then [done]]
```

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

```
val= <y,y,{}>
todo= [let f in (f 10) {} then [done]]

exp= (f 10)
env= {f=<y,y,{}>,{}}
todo= [done]
```

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

exp= (f 10)
env= {f=<y,y,{}>,{}}
todo= [done]

exp= f
env= {f=<y,y,{}>,{}}
todo= [apparg 10 in {f=<y,y,{}>,{}} then [done]]

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

exp= f
env= {f=<y,y,{}>,{}}
todo= [apparg 10 in {f=<y,y,{}>,{}} then [done]]

val= <y,y,{}>
todo= [apparg 10 in {f=<y,y,{}>,{}} then [done]]

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

val= <y,y,{}>
todo= [apparg 10 in {f=<y,y,{}>,{}} then [done]]

exp= 10
env= {f=<y,y,{}>,{}}
todo= [app <y,y,{}> then [done]]

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

exp= 10
env= {f=<y,y,{}>,{}}
todo= [app <y,y,{}> then [done]]

val= 10
todo= [app <y,y,{}> then [done]]

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

```
val= 10
todo= [app <y,y,{}> then [done]]
```

```
exp= y
env= {y=10,{}}
todo= [done]
```

Evaluation with To-Do List

```
let f = proc(y)y
 in (f 10)
```

```
exp= y
env= {y=10,{}}
todo= [done]
```

```
val= 10
todo= [done]
```

To-Do Lists

- To-do list is called the ***continuation***
- It makes the Scheme context in our interpreter explicit

Interpreter now consists of two main functions:

- `eval-expression : expr env cont -> expval`

```
exp= 1
env= {}
todo= [done]
```

- `apply-cont : value cont -> expval`

```
val= 1
todo= [done]
```

Continuation Datatype

```
(define-datatype
  continuation continuation?
  (done-cont)
  (app-arg-cont (rand expression?)
                (env environment?)
                (cont continuation?))
  (app-cont (rator value?)
            (cont continuation?))
  ...)
```

Continuation Datatype

```
[done]
=
(done-cont)

[addval 1 then [done]]
=
(prim-cont (add-prim) 1 (done-cont))

[addexp y in {y=10} then [done]]
=
(prim-other-cont (add-prim)
   (var-exp ’y)
   (extend-env ’(y) ’(10) (empty-env))
   (done-cont))
```

Continuation Datatype

```
[let f in (f 10) {} then [done]]
=
(let-cont ’f (app-exp (var-exp ’f)
                      (list-exp 10))
          (empty-env)
          (done-cont))
```

Interpreter

```
(define eval-program
  (lambda (pgm)
    (cases program pgm
      (a-program (body)
        (eval-expression body
                         (init-env)
                         (done-cont))))))
```

Interpreter

```
(define (eval-expression exp env cont)
  (cases expression exp
    (lit-exp (datum)
      (apply-cont cont datum))
    (var-exp (id)
      (apply-cont cont (apply-env env id)))
    (proc-exp (id body-exp)
      (apply-cont cont
                  (closure id body-exp env)))
    ...)))

(define (apply-cont cont val)
  (cases continuation cont
    (done-cont () val)
    ...))
```

Interpreter: Let

```
... ; in eval-expression:
(let-exp (id exp body-exp)
 (eval-expression
  exp env
  (let-cont id body-exp env cont)))
...
...  ; in apply-cont:
(let-cont (id body env cont)
  (eval-expression
   body (extend-env (list id) (list val)
                    env)
       cont))
...
```

Interpreter: Primitives

```
... ; in eval-expression:
(primapp-exp (prim rand1 rand2)
  (eval-expression
   rand1 env
   (prim-other-cont prim rand2 env cont)))
...
...  ; in apply-cont:
(prim-other-cont (prim arg2 env cont)
  (eval-expression
   arg2 env
   (prim-cont prim val cont)))
(prim-cont (prim arg1-val cont)
  (apply-cont cont
    (apply-primitive prim arg1-val val)))
...
```

Interpreter: Application

```
... ; in eval-expression:
(app-exp (rator rand)
 (eval-expression rator env
    (app-arg-cont rand env cont)))
...
...  ; in apply-cont:
(app-arg-cont (rand env cont)
  (eval-expression rand env
          (app-cont val cont)))
(app-cont (f cont)
  (apply-proc f val cont))
...
```

Interpreter: If

```
... ; in eval-expression:
(if-exp (test then else)
  (eval-expression test env
     (if-cont then else env cont)))
...
...  ; in apply-cont:
(if-cont (then else env cont)
  (eval-expression
   (if (zero? val) else then)
   env cont))
...
```

Continuations

- Every call to **`eval-expression`** or **`apply-cont`** is a tail call
- Tail calls could be replaced by **`goto`**
- Our interepreter does not rely on Scheme's "stack" at all!

Continuations as Values

What if a program could see its continuation?

```
letcc k
 in +(1, continue k 3)
```

- **`letcc`**: puts the current continuation into a variable
- **`continue`**: sends a value to a continuation, forgets the current continuation

Continuations as Values

```
letcc k
 in +(1, continue k 3)
```

```
exp= letcc k in +(1, continue k 3)
env= {}
todo= [done]
```

Continuations as Values

```
letcc k
 in +(1, continue k 3)
```

```
exp= letcc k in +(1, continue k 3)
env= {}
todo= [done]
```

```
exp= +(1, continue k 3)
env= {k=[done],{}}
todo= [done]
```

Continuations as Values

```
letcc k
 in +(1, continue k 3)
```

```
exp= +(1, continue k 3)
env= {k=[done],{}}
todo= [done]

exp= 1
env= {k=[done],{}}
todo= [addexp continue k 3 {k=[done],{}} then [done]]
```

Continuations as Values

```
letcc k
 in +(1, continue k 3)
```

```
exp= 1
env= {k=[done],{}}
todo= [addexp continue k 3 {k=[done],{}} then [done]]

val= 1
todo= [addexp continue k 3 {k=[done],{}} then[done]]
```

Continuations as Values

```
letcc k
 in +(1, continue k 3)
```

```
val= 1
todo= [addexp continue k 3 {k=[done],{}} then[done]]

exp= continue k 3
env= {k=[done],{}}
todo= [addval 1 then [done]]
```

Continuations as Values

```
letcc k
 in +(1, continue k 3)
```

```
exp= continue k 3
env= {k=[done],{}}
todo= [addval 1 then [done]]

val= 3
todo= [done]
```

Done!

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
exp= +(4, letcc k in +(1, continue k 3))
env= {}
todo= [done]
```

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
exp= +(4, letcc k in +(1, continue k 3))
env= {}
todo= [done]

exp= 4
env= {}
todo= [addexp letcc k in +(1, continue k 3))
      {} then [done]]
```

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
exp= 4
env= {}
todo= [addexp letcc k in +(1, continue k 3))
      {} then [done]]

val= 4
todo= [addexp letcc k in +(1, continue k 3))
      {} then [done]]
```

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
val= 4
todo= [addexp letcc k in +(1, continue k 3))
      {} then [done]]

exp= letcc k in +(1, continue k 3)
env= {}
todo= [addval 4 then [done]]
```

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
exp= letcc k in +(1, continue k 3)
env= {}
todo= [addval 4 then [done]]
```

```
exp= +(1, continue k 3)
env= {k=[addval 4 then [done]],{}}
todo= [addval 4 then [done]]
```

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
exp= +(1, continue k 3)
env= {k=[addval 4 then [done]],{}}
todo= [addval 4 then [done]]
```

```
exp= 1
env= {k=[addval 4 then [done]],{}}
todo= [addexp continue k 3
      {k=[addval 4 then [done]],{}}
      then [addval 4 then [done]]]
```

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
exp= 1
env= {k=[addval 4 then [done]],{}}
todo= [addexp continue k 3
      {k=[addval 4 then [done]],{}}
      then [addval 4 then [done]]]
```

```
val= 1
todo= [addexp continue k 3
      {k=[addval 4 then [done]],{}}
      then [addval 4 then [done]]]
```

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
val= 1
todo= [addexp continue k 3
      {k=[addval 4 then [done]],{}}
      then [addval 4 then [done]]]
```

```
exp= continue k 3
env= {k=[addval 4 then [done]],{}}
todo= [addval 1 then [addval 4 then [done]]]
```

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
exp= continue k 3
env= {k=[addval 4 then [done]],{}}
todo= [addval 1 then [addval 4 then [done]]]
```

```
val= 3
todo= [addval 4 then [done]]
```

Continuations as Values

```
+(4, letcc k
      in +(1, continue k 3))
```

```
val= 3
todo= [addval 4 then [done]]
```

```
val= 7
todo= [done]
```

Done!

Continuations as Values

```
let f = letcc k in k
 continue f f
```

```
exp= let f = letcc k in k continue f f
env= {}
todo= [done]
```

Continuations as Values

```
let f = letcc k in k
 continue f f
```

```
exp= let f = letcc k in k continue f f
env= {}
todo= [done]
```

```
exp= letcc k in k
env= {}
todo= [let f in continue f f {} [done]]
```

Continuations as Values

```
let f = letcc k in k
 continue f f
```

```
exp= letcc k in k
env= {}
todo= [let f in continue f f {} [done]]
```

```
exp= k
env= {k=[let f in continue f f {} [done]],{}}
todo= [let f in continue f f {} [done]]
```

Continuations as Values

```
let f = letcc k in k
 continue f f
```

```
exp= k
env= {k=[let f in continue f f {} [done]],{}}
todo= [let f in continue f f {} [done]]
```

```
val= [let f in continue f f {} [done]]
todo= [let f in continue f f {} [done]]
```

Continuations as Values

```
let f = letcc k in k
 continue f f
```

```
val= [let f in continue f f {} [done]]
todo= [let f in continue f f {} [done]]
```

```
exp= continue f f
env= {f=[let f in continue f f {} [done]],{}}
todo= [done]
```

Continuations as Values

```
let f = letcc k in k
 continue f f
```

```
exp= continue f f
env= {f=[let f in continue f f {} [done]],{}}
todo= [done]
```

```
val= [let f in continue f f {} [done]]
todo= [let f in continue f f {} [done]]
```

Infinite loop!