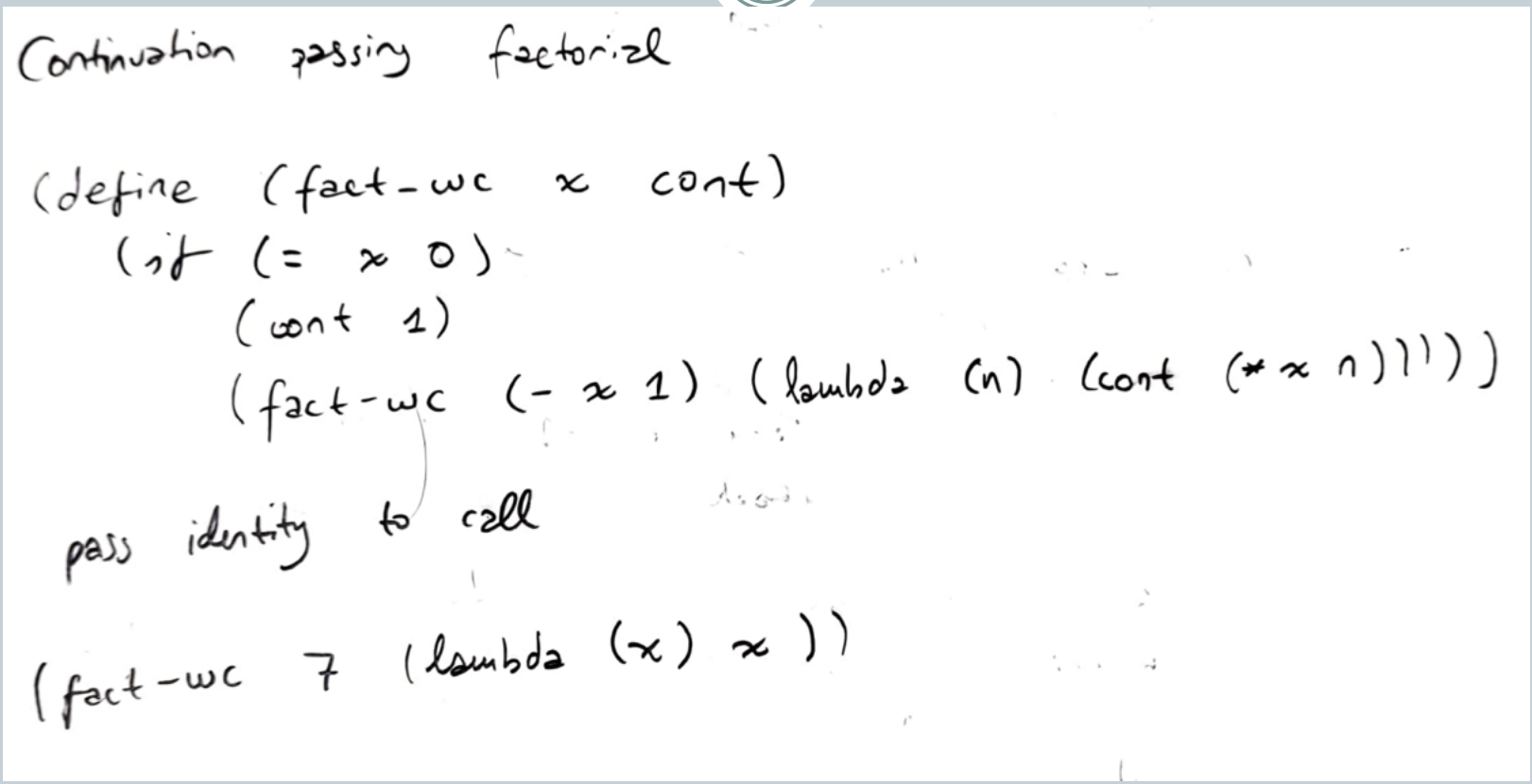
CPS – Implementation

T. METIN SEZGIN

Review



Continuation passing factorial

```
(define (fact-wc x cont)
  (if (= x 0)
      (cont 1)
      (fact-wc (- x 1) (lambda (n) (cont (* x n))))))
```

pass identity to call

```
(fact-wc 7 (lambda (x) x))
```

Cem Akarsubasi

Implementation

- What are the main changes?
  - The new value-of
  - Introduction of continuations
- How do we represent continuations?
  - Procedural
  - Data-structure
- How do we apply continuations?
  - Procedural
  - Data-structure

Procedural representation

Cont = *ExpVal* → *FinalAnswer*

end-cont : () → *Cont*

```
(define end-cont
  (lambda ()
    (lambda (val)
      (begin
        (eopl:printf "End of computation.~%")
        val))))
```

zero1-cont : *Cont* → *Cont*

```
(define zero1-cont
  (lambda (cont)
    (lambda (val)
      (apply-cont cont
        (bool-val
          (zero? (expval->num val)))))))
```

let-exp-cont : *Var* × *Exp* × *Env* × *Cont* → *Cont*

```
(define let-exp-cont
  (lambda (var body env cont)
    (lambda (val)
      (value-of/k body (extend-env var val env) cont))))
```

if-test-cont : *Exp* × *Exp* × *Env* × *Cont* → *Cont*

```
(define if-test-cont
  (lambda (exp2 exp3 env cont)
    (lambda (val)
      (if (expval->bool val)
        (value-of/k exp2 env cont)
        (value-of/k exp3 env cont)))))
```

apply-cont : *Cont* × *ExpVal* → *FinalAnswer*

```
(define apply-cont
  (lambda (cont v)
    (cont v)))
```

Applying the continuations

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf
      "End of computation.~%")
    val)

(apply-cont (diff1-cont exp2 env cont) val1)
= (value-of/k exp2 env (diff2-cont val1 cont))

(apply-cont (diff2-cont val1 cont) val2)
= (let ((num1 (expval->num val1))
        (num2 (expval->num val2)))
    (apply-cont cont (num-val (- num1 num2))))

(apply-cont (rator-cont rand env cont) val1)
= (value-of/k rand env (rand-cont val1 cont))
```

```
(apply-cont (rand-cont val1 cont) val2)
= (let ((proc1 (expval->proc val1)))
    (apply-procedure/k proc1 val2 cont))

(apply-cont (zero1-cont cont) val)
= (apply-cont cont (bool-val (zero? (expval->num val))))

(apply-cont (if-test-cont exp2 exp3 env cont) val)
= (if (expval->bool val)
    (value-of/k exp2 env cont)
    (value-of/k exp3 env cont))

(apply-cont (let-exp-cont var body env cont) val1)
= (value-of/k body (extend-env var val1 env) cont)
```

Data-structure representation

```
(define-datatype continuation continuation?
  (end-cont)
  (zero1-cont
    (cont continuation?))
  (let-exp-cont
    (var identifier?)
    (body expression?)
    (env environment?)
    (cont continuation?))
  (if-test-cont
    (exp2 expression?)
    (exp3 expression?)
    (env environment?)
    (cont continuation?)))
```

apply-cont : *Cont* × *ExpVal* → *FinalAnswer*

```
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (end-cont ()
        (begin
          (eopl:printf "End of computation.~%")
          val))
      (zero1-cont (saved-cont)
        (apply-cont saved-cont
          (bool-val
            (zero? (expval->num val)))))
      (let-exp-cont (var body saved-env saved-cont)
        (value-of/k body
          (extend-env var val saved-env) saved-cont))
      (if-test-cont (exp2 exp3 saved-env saved-cont)
        (if (expval->bool val)
          (value-of/k exp2 saved-env saved-cont)
          (value-of/k exp3 saved-env saved-cont))))))
```

value-of/k : *Exp* × *Env* × *Cont* → *FinalAnswer*

```
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num) (apply-cont cont (num-val num)))
      (var-exp (var) (apply-cont cont (apply-env env var)))
      (proc-exp (var body)
        (apply-cont cont
          (proc-val
            (procedure var body env))))
      (letrec-exp (p-name b-var p-body letrec-body)
        (value-of/k letrec-body
          (extend-env-rec p-name b-var p-body env)
          cont))
      (zero?-exp (exp1)
        (value-of/k exp1 env
          (zero1-cont cont)))
      (if-exp (exp1 exp2 exp3)
        (value-of/k exp1 env
          (if-test-cont exp2 exp3 env cont)))
      (let-exp (var exp1 body)
        (value-of/k exp1 env
          (let-exp-cont var body env cont)))
      (diff-exp (exp1 exp2)
        (value-of/k exp1 env
          (diff1-cont exp2 env cont)))
      (call-exp (rator rand)
        (value-of/k rator env
          (rator-cont rand env cont))))))
```

value-of-program : *Program* → *FinalAnswer*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of/k exp1 (init-env) (end-cont))))))
```

apply-procedure/k : *Proc* × *ExpVal* × *Cont* → *FinalAnswer*

```
(define apply-procedure/k
  (lambda (proc1 val cont)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of/k body
          (extend-env var val saved-env)
          cont)))))
```