[redacted]

[redacted]

Matt Might

University of Utah

matt.might.net

Continuation-passing style

But first...

problem, boole?

Loose comparisons with ==

	TRUE	FALSE	*1*	*0*	*-1*	*"1"*	*"0"*	*"-1"*	NULL	*array()*	*"php"*	*""*
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

```
'9223372036854775807' == '9223372036854775808'
```

Final project

<prog> ::= <defs> <exp>

<defs> ::= <def> <defs>

```
<def> ::= (define (<var> <formals>) <exp>)

<formals> ::= <var> <formals>
            |
```

```
<exp> ::= <integer>
```

```
<exp> ::= <integer>
        |   <var>
```

```
<exp> ::= <integer>
        |   <var>
        |   #t | #f
```

```
<exp> ::= <integer>
        |   <var>
        |   #t | #f
        |   (lambda (<formals>) <exp>)
```

```
<exp> ::= <integer>
        |   <var>
        |   #t | #f
        |   (lambda (<formals>) <exp>)
        |   (let ([<var> <exp>] ...) <exp>)
```

```
<exp> ::= <integer>
        |  <var>
        |  #t | #f
        |  (lambda (<formals>) <exp>)
        |  (let ([<var> <exp>] ...) <exp>)
        |  (if <exp> <exp> <exp>)
```

```
<exp> ::= <integer>
        |  <var>
        |  #t | #f
        |  (lambda (<formals>) <exp>)
        |  (let ([<var> <exp>] ...) <exp>)
        |  (if <exp> <exp> <exp>)
        |  (set! <var> <exp>)
```

```
<exp> ::= <integer>
        |  <var>
        |  #t | #f
        |  (lambda (<formals>) <exp>)
        |  (let ([<var> <exp>] ...) <exp>)
        |  (if <exp> <exp> <exp>)
        |  (set! <var> <exp>)
        |  (begin <exp> ...)
```

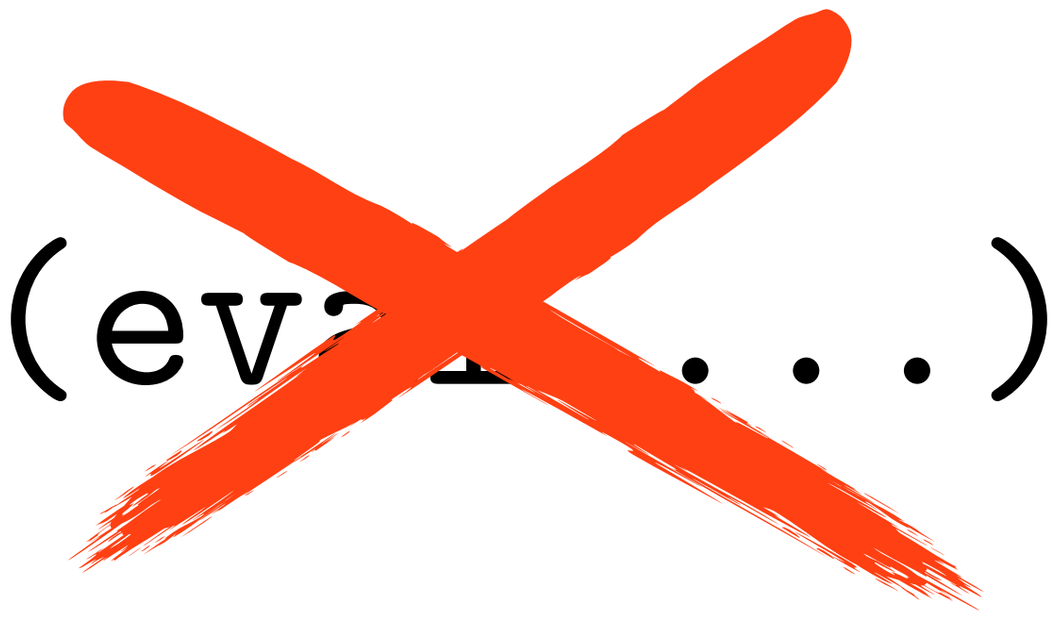
```
<exp> ::= <integer>
        |  <var>
        |  #t | #f
        |  (lambda (<formals>) <exp>)
        |  (let ([<var> <exp>] ...) <exp>)
        |  (if <exp> <exp> <exp>)
        |  (set! <var> <exp>)
        |  (begin <exp> ...)
        |  (<exp> <exp> ...)
```

```
<exp> ::= <integer>
        |  <var>
        |  #t | #f
        |  (lambda (<formals>) <exp>)
        |  (let ([<var> <exp>] ...) <exp>)
        |  (if <exp> <exp> <exp>)
        |  (set! <var> <exp>)
        |  (begin <exp> ...)
        |  (<exp> <exp> ...)
        |  (<prim> <exp> ...)
```

```
<exp> ::= <integer>
        |  <var>
        |  #t | #f
        |  (lambda (<formals>) <exp>)
        |  (let ([<var> <exp>] ...) <exp>)
        |  (if <exp> <exp> <exp>)
        |  (set! <var> <exp>)
        |  (begin <exp> ...)
        |  (<exp> <exp> ...)
        |  (<prim> <exp> ...)
        |  (call/cc <exp>)
```

```
<prim> ::= +   |   -   |   *   |   =
        |   void
```

(eval ...)



Due: April 26 AoE

Next week

- Tuesday lecture: Online
- Thursday lecture: Flux!

Continuation-Passing Style

CPS

What is CPS?

A *style* of programming

An intermediate form

Why CPS?

Web programming

Simplifies interpreters

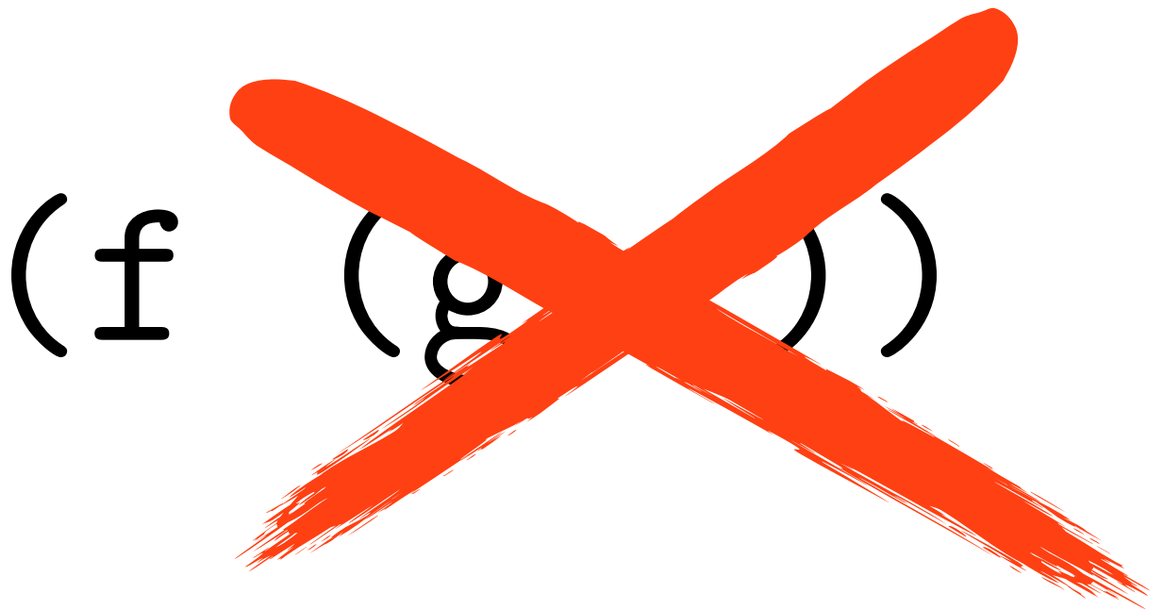
Eliminates `call/cc`

Two rules

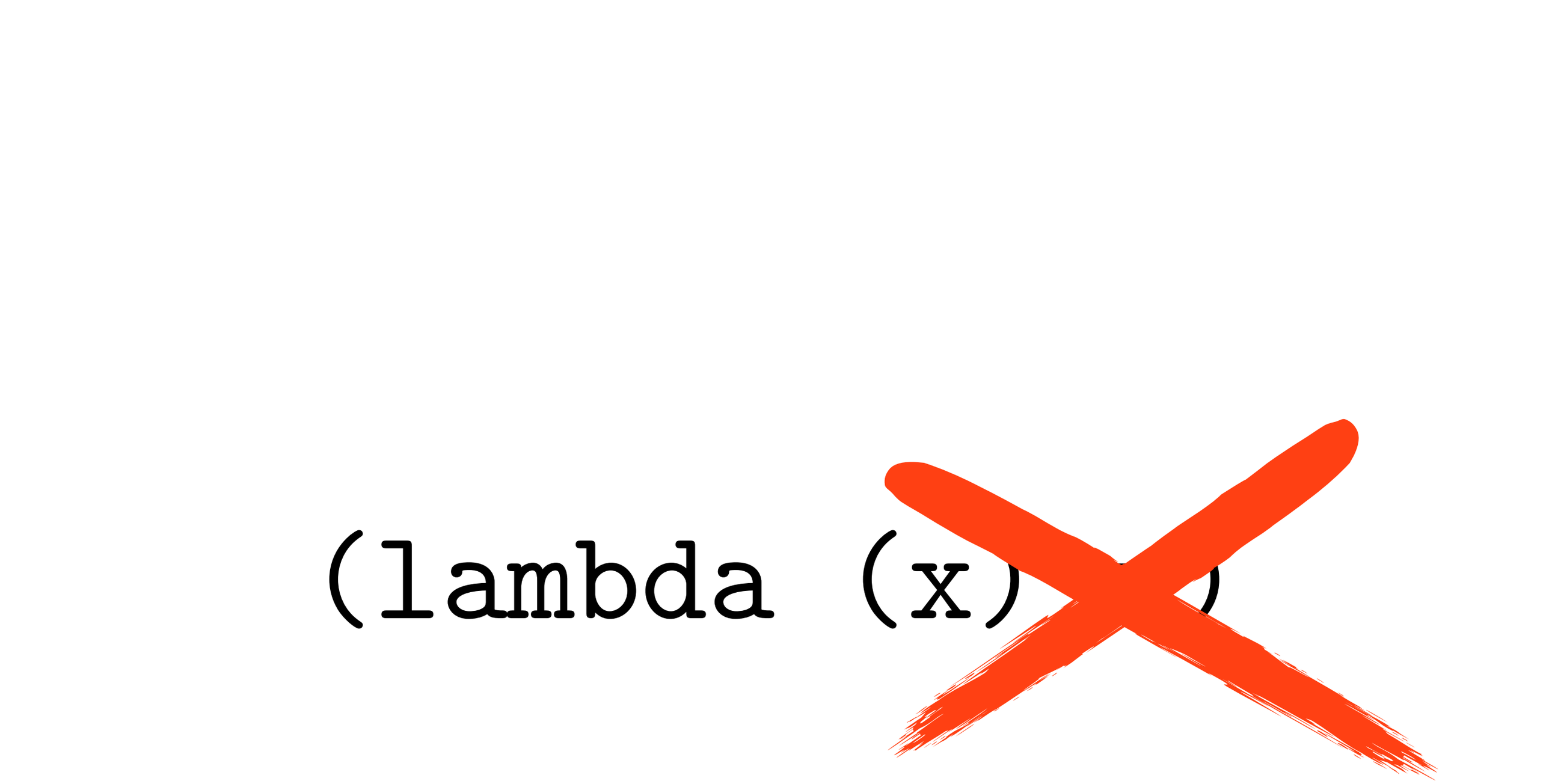
Functions never return

Arguments are atomic

```
(f (g x))
```


(f (g))

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


(lambda (x))

?

Pass callbacks

Pass continuations

```
(λ (x) x)
```

```
(λ (x cc) (cc x))
```

```
(λ (x return)
(return x))
```

Example

```
(define (f n)
  (if (= n 0)
      1
      (* n (f (- n 1)))))
```

Example

```
(define (f n return)
  (if (= n 0)
      1
      (* n (f (- n 1)))))
```

Example

```
(define (f n return)
  (if (= n 0)
      (return 1)
      (* n (f (- n 1)))))
```

Example

```
(define (f n return)
  (if (= n 0)
      (return 1)
      (f (- n 1) (lambda (m)
       (return (* n m))))))
```

Example

```
(define (f n return)
 (=$ n 0 (lambda (zero?)
  (if zero?
     (return 1)
     (-$ n 1 (lambda (sub)
      (f sub (lambda (mul)
       (*$ n mul return)))))))))
```

Example

```
(define (f a n return)
 (=$ n 0 (lambda (zero?)
(if zero?
 (return a)
 (*$ a n (lambda (an)
  (-$ n 1 (lambda (n1)
  (f an n1 return)))))))))
```

Example

```
(define (fib n return)
 (<=$ n 0 (lambda (zero?)
  (if zero?
      (return n)
      (-$ n 1 (lambda (n1)
       (-$ n 2 (lambda (n2)
        (fib n1 (lambda (f1)
         (fib n2 (lambda (f2)
          (+$ f1 f2 return)))))))))))))
```

Naive transform

$$
\begin{aligned}
expr &::= (\lambda\ (var)\ expr) \\
&\mid\ var \\
&\mid\ (expr\ expr)
\end{aligned}
$$

$$
\begin{aligned}
aexp &::= (\lambda\ (var_1\ \ldots\ var_n)\ cexp) \\
&\ \ |\ \ var \\
\\
cexp &::= (aexp_0\ \ldots\ aexp_n)
\end{aligned}
$$

```
(define (T expr cont)
```

```
(define (T expr cont)
```

A term that invokes `cont` on the result of `expr`.

```
(define (T expr cont) `(,cont ,expr))
```

```
(define (T expr cont)
  (match expr
```

```
(define (T expr cont)
  (match expr
    [`(λ . ,_)      `(,cont ,   expr )]
```

```
(define (T expr cont)
  (match expr
    [`(λ . ,_)      `(,cont ,(M expr))]
```

```
(define (M expr)
```

```
(define (M expr)
```

An equivalent expression that obeys rules of CPS.

```
(define (M expr)
  (match expr
```

```
(define (M expr)
  (match expr
    [`(λ (,var) ,expr)
```

```
(define (M expr)
  (match expr
    [`(λ (,var) ,expr)
       ; =>
       (define $k (gensym '$k))
```

```
(define (M expr)
  (match expr
    [`(λ (,var) ,expr)
       ; =>
      (define $k (gensym '$k))
     `(λ (,var ,$k) ,(T expr $k))]
```

```
(define (M expr)
  (match expr
    [`(λ (,var) ,expr)
       ; =>
      (define $k (gensym '$k))
     `(λ (,var ,$k) ,(T expr $k))]

    [(? symbol?)  #;=>  expr]))
```

```
(define (T expr cont)
  (match expr
    [`(λ . ,_)      `(,cont ,(M expr))]
```

```
(define (T expr cont)
  (match expr
    [`(λ . ,_)      `(,cont ,(M expr))]
    [ (? symbol?)   `(,cont ,(M expr))]
```

```
(define (T expr cont)
  (match expr
    [`(λ . ,_)      `(,cont ,(M expr))]
    [ (? symbol?)   `(,cont ,(M expr))]
    [`(,f ,e)
```

```
(define (T expr cont)
  (match expr
    [`(λ . ,_)      `(,cont ,(M expr))]
    [ (? symbol?)   `(,cont ,(M expr))]
    [`(,f ,e)
; =>
(define $f (gensym '$f))
(define $e (gensym '$e))
```

```
(define (T expr cont)
  (match expr
    [`(λ . ,_)      `(,cont ,(M expr))]
    [ (? symbol?)   `(,cont ,(M expr))]
    [`(,f ,e)
; =>
(define $f (gensym '$f))
(define $e (gensym '$e))
(T f `(λ (,$f)
```

```
(define (T expr cont)
  (match expr
    [`(λ . ,_)      `(,cont ,(M expr))]
    [ (? symbol?)   `(,cont ,(M expr))]
    [`(,f ,e)
; =>
(define $f (gensym '$f))
(define $e (gensym '$e))
(T f `(λ (,$f)
         ,(T e `(λ (,$e)
```

```
(define (T expr cont)
  (match expr
    [`(λ . ,_)      `(,cont ,(M expr))]
    [ (? symbol?)   `(,cont ,(M expr))]
    [`(,f ,e)
; =>
(define $f (gensym '$f))
(define $e (gensym '$e))
(T f `(λ (,$f)
         ,(T e `(λ (,$e)
                   (,$f ,$e ,cont)))))]))
```

```
(M '(λ (x) x))
```

```
(λ (x $k9) ($k9 x))
```

```
(T '(g a) 'halt)
```

```
((λ ($f9596)
  ((λ ($e9597)
    ($f9596 $e9597 halt)) a)) g)
```

```
(g a halt)
```

call/cc?

```
(λ (f cc)
  (f (λ (x cc*)
       (cc x))
     cc)))
```

Net effect?

CESK

CES

Braintaser

```
(call/cc call/cc)
```

See blog for more!