Matt Might
University of Utah
matt.might.net

Continuations

[redacted]

[redacted]

Matt Might

University of Utah

matt.might.net

Continuation-passing style

```
x = 30

if TuringMachine():
    del x
else:
    x = 10

print(x) # error?
```

Why continuations?

- Back-tracking search
- Model exceptions
- Cooperative threads
- Preemptive threads
- Generators
- Coroutine systems
- Time-travel

What are continuations?

Continuations

A continuation is like a saved game.

Continuations

They're like time travel.

Continuations

They're like `go-when` instead of `goto`.

Continuations

The current continuation is “the rest of the computation.”

Continuations

The program stack encodes the current continuation.

Continuations

The state of a thread is a continuation.

Continuations

A continuation is a procedure
that never returns to its caller.

Continuations

Exceptions are a special case of continuations.

Continuations

A continuation is a first-class encoding of control.

Heap

Registers

Stack

Heap

Registers

Stack

Registers

Stack

Heap

Registers

Stack

Now, some magic

go-when **and** right-now

```
; An infinite loop:
(let ((the-beginning (right-now)))
  (display "Hello, world!")
  (newline)
  (go-when the-beginning))
```

The `amb` “function”

SAT-solving

```
; The following evaluates to (#f #f #t)
(sat-solve (and (implies a (not b))
                (not a)
                c)
           (list a b c))
```

Cooperative threads

```
(spawn (make-thread-thunk 'a))
(spawn (make-thread-thunk 'b))
(spawn (make-thread-thunk 'c))

(start-threads)
```

```
(define counter 10)

(define (make-thread-thunk name)
  (letrec ((loop (lambda ()
                   (if (< counter 0)
                       (quit))
                   (display "in thread ")
                   (display name)
                   (display "; counter = ")
                   (display counter)
                   (newline)
                   (set! counter (- counter 1))
                   (yield)
                   (loop))))
    loop))
```

Generators

```
(for v in (tree-iterator '(3 . ((4 . 5) . 6 )))
  (display v)
  (newline))
```

Generators

```
(define (tree-iterator tree)
  (lambda (yield)

    (define (walk tree)
      (if (not (pair? tree))
          (yield tree)
          (begin
            (walk (car tree))
            (walk (cdr tree)))))

    (walk tree)))
```

call-with-current-continuation

call/cc

```
(call/cc (lambda (cc) …))
```

current-continuation

```
(define (current-continuation)
  (call/cc (lambda (cc) (cc cc))))
```

Design pattern

```
(let ((cc (current-continuation)))
  (cond
   ((procedure? cc) ...)
   ((future-val? cc) ...)
   (else (error “contract broken!”))))
```

Escape pattern

```
(lambda (...)
 (call/cc (lambda (return)
   ...)))
```

Escape pattern

```
(call/cc (lambda (break)
  (for ...)))
```

call-with-escape-continuation

call/ec

Details

Continuation-Passing Style

CPS

What is CPS?

A *style* of programming

An intermediate form

Why CPS?

Web programming

Simplifies interpreters

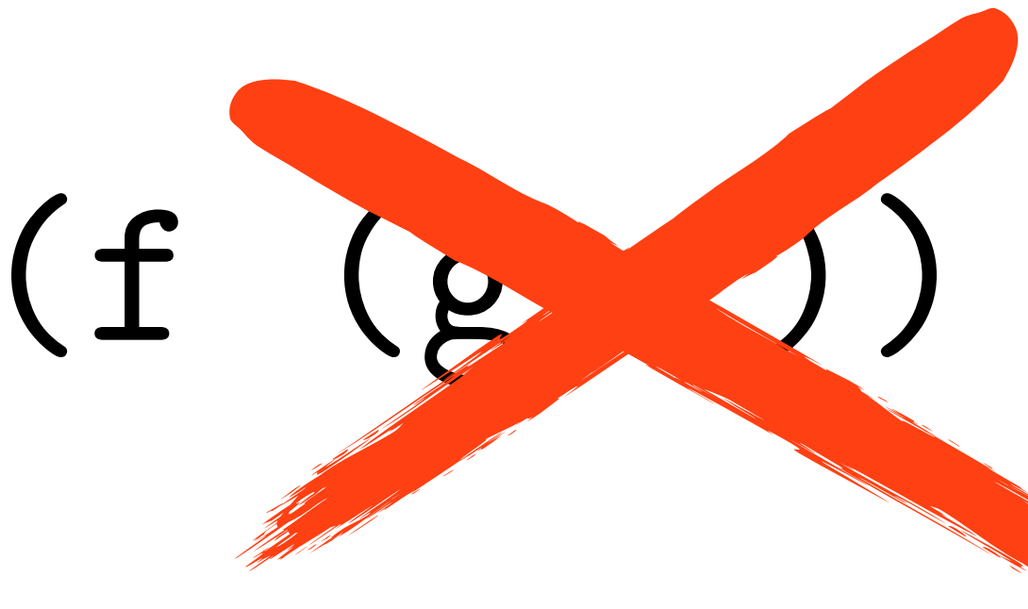
Eliminates `call/cc`

Two rules

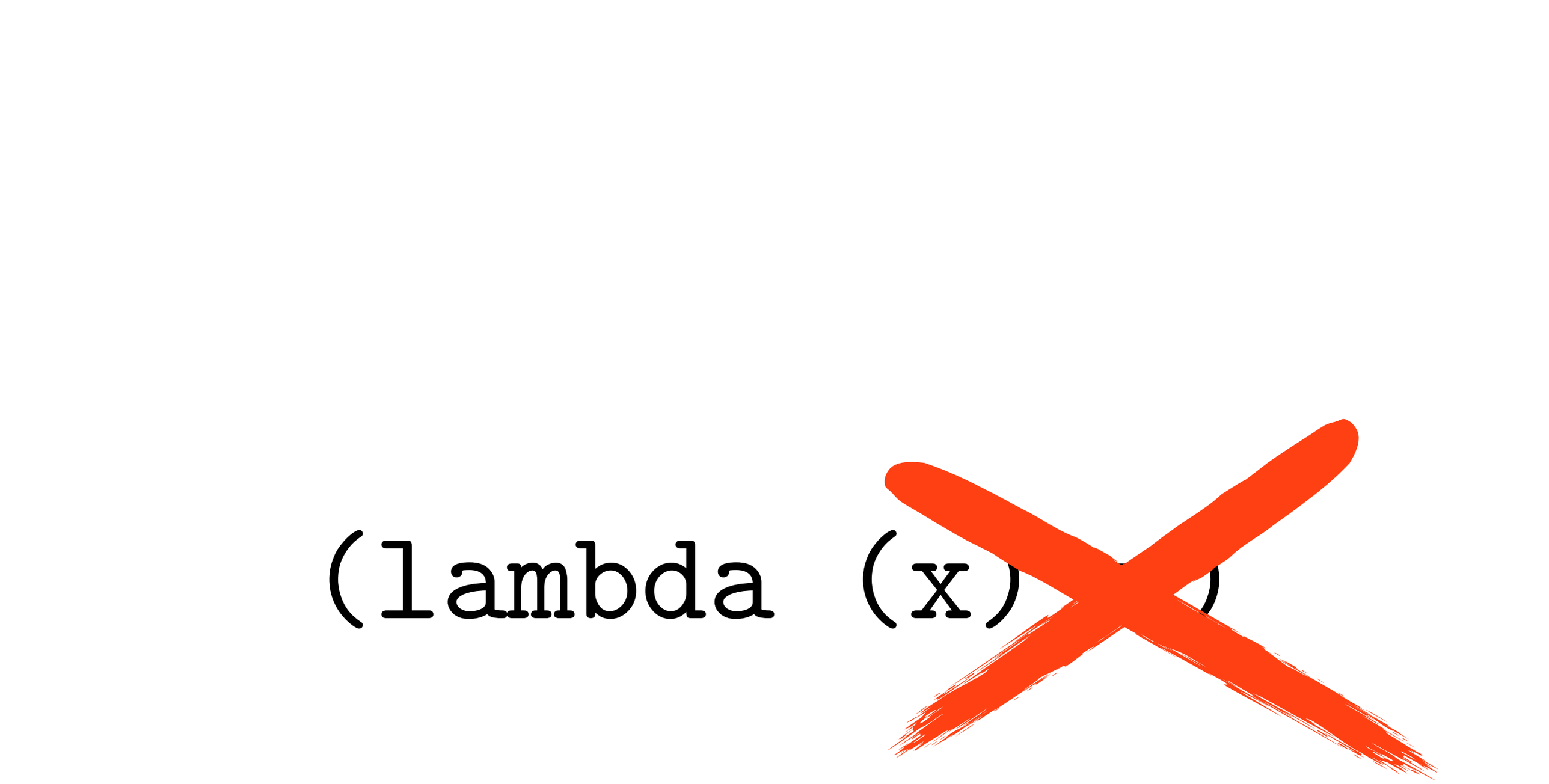
Functions never return

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?

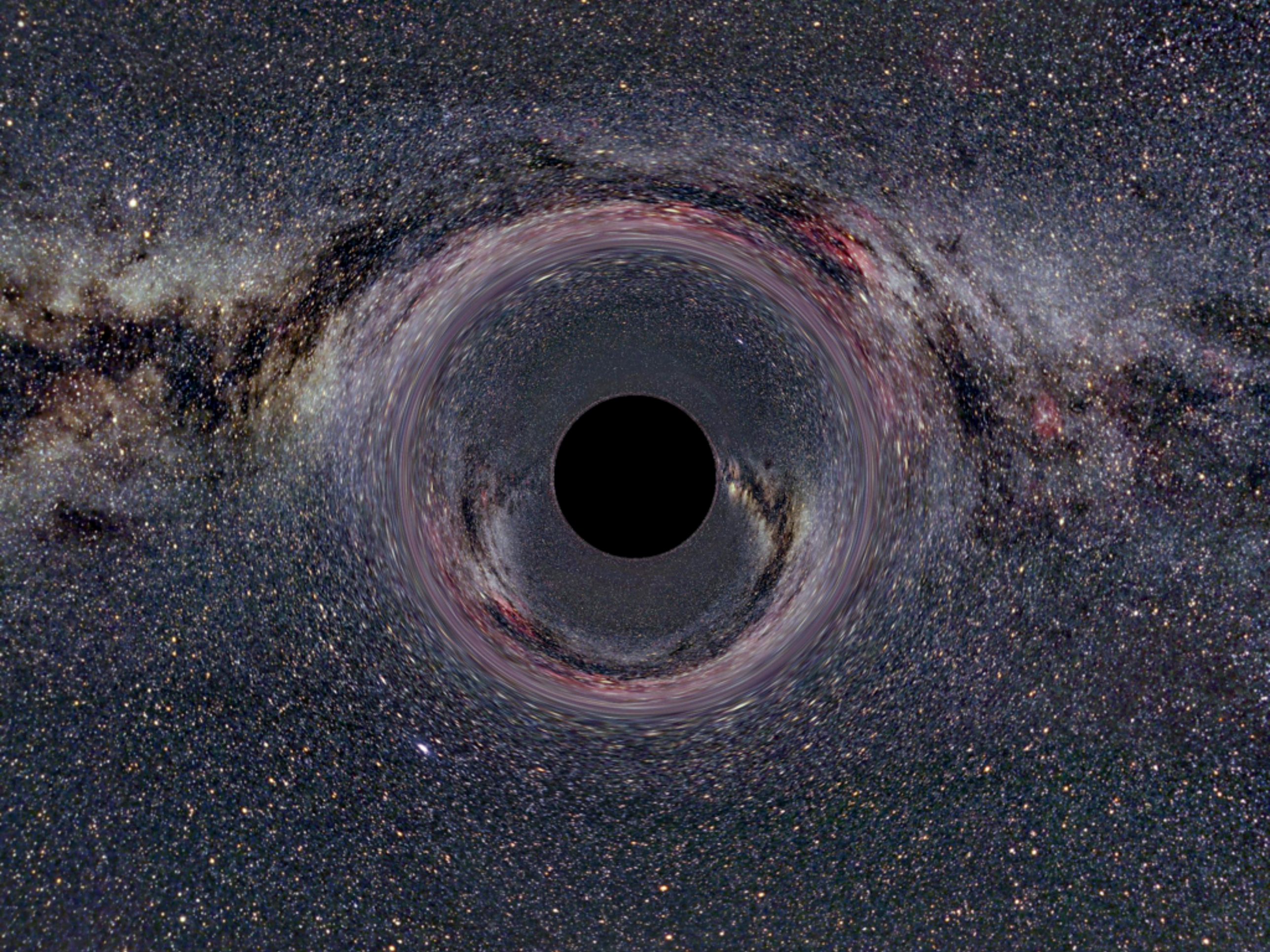
```
(λ (f cc)
  (f (λ (x cc*)
       (cc x))
     cc)))
```

Brainteaser

Braintaser

Braintaser

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



More on the blog...