# Lambda Calculus and Computation

## 6.037 Structure and Interpretation of Computer Programs

Chelsea Voss

csvoss@mit.edu
Massachusetts Institute of Technology
With material from Michael Philips and Nelson Elhage

February 1, 2018

# Limits to Computation

David Hilbert's *Entscheidungsproblem* (1928)

# Limits to Computation

David Hilbert's *Entscheidungsproblem* (1928): can calculating machines give a yes or no answer to all mathematical questions?

# Limits to Computation

David Hilbert's *Entscheidungsproblem* (1928): can calculating machines give a yes or no answer to all mathematical questions?

Figure: Alonzo Church (1903-1995), lambda calculus

Figure: Alan Turing (1912-1954), Turing machines

# Limits to Computation

David Hilbert's *Entscheidungsproblem* (1928): can calculating machines give a yes or no answer to all mathematical questions?

Figure: Alonzo Church (1903-1995), lambda calculus

Figure: Alan Turing (1912-1954), Turing machines

**Theorem** (Church, Turing, 1936): These models of computation can't solve every problem.

# Limits to Computation

David Hilbert's *Entscheidungsproblem* (1928): can calculating machines give a yes or no answer to all mathematical questions?

Figure: Alonzo Church (1903-1995), lambda calculus

Figure: Alan Turing (1912-1954), Turing machines

**Theorem** (Church, Turing, 1936): These models of computation can't solve every problem. Proof: next!

# Equivalence of Computation Methods

First part of the proof: **Church-Turing thesis.**

# Equivalence of Computation Methods

First part of the proof: **Church-Turing thesis.**
*Any intuitive notion for a "computer" that you can come up with will be no more powerful than a Turing machine or than lambda calculus. That is, most models of computation are equivalent.*

# Equivalence of Computation Methods

First part of the proof: **Church-Turing thesis.**
*Any intuitive notion for a "computer" that you can come up with will be no more powerful than a Turing machine or than lambda calculus. That is, most models of computation are equivalent.*
*Turing-complete* means capable of simulating Turing machines.

# Equivalence of Computation Methods

First part of the proof: **Church-Turing thesis.**
*Any intuitive notion for a "computer" that you can come up with will be no more powerful than a Turing machine or than lambda calculus. That is, most models of computation are equivalent.*
*Turing-complete* means capable of simulating Turing machines.
Lambda calculus is Turing-complete (proof: later), and Turing machines can simulate lambda calculus.

# Equivalence of Computation Methods

First part of the proof: **Church-Turing thesis.**
*Any intuitive notion for a "computer" that you can come up with will be no more powerful than a Turing machine or than lambda calculus. That is, most models of computation are equivalent.*
*Turing-complete* means capable of simulating Turing machines.
Lambda calculus is Turing-complete (proof: later), and Turing machines can simulate lambda calculus.
Some others:

- *Turing machines* are Turing-complete

# Equivalence of Computation Methods

First part of the proof: **Church-Turing thesis.**

*Any intuitive notion for a "computer" that you can come up with will be no more powerful than a Turing machine or than lambda calculus. That is, most models of computation are equivalent.*

*Turing-complete* means capable of simulating Turing machines.

Lambda calculus is Turing-complete (proof: later), and Turing machines can simulate lambda calculus.

Some others:

- *Turing machines* are Turing-complete
- *Scheme* is Turing-complete

# Equivalence of Computation Methods

First part of the proof: **Church-Turing thesis.**
*Any intuitive notion for a "computer" that you can come up with will be no more powerful than a Turing machine or than lambda calculus. That is, most models of computation are equivalent.*
*Turing-complete* means capable of simulating Turing machines.
Lambda calculus is Turing-complete (proof: later), and Turing machines can simulate lambda calculus.
Some others:

- *Turing machines* are Turing-complete
- *Scheme* is Turing-complete
- *Minecraft* is Turing-complete

# Equivalence of Computation Methods

First part of the proof: **Church-Turing thesis.**
*Any intuitive notion for a "computer" that you can come up with will be no more powerful than a Turing machine or than lambda calculus. That is, most models of computation are equivalent.*
*Turing-complete* means capable of simulating Turing machines.
Lambda calculus is Turing-complete (proof: later), and Turing machines can simulate lambda calculus.
Some others:

- *Turing machines* are Turing-complete
- *Scheme* is Turing-complete
- *Minecraft* is Turing-complete
- *Conway's Game of Life* is Turing-complete

## Equivalence of Computation Methods

First part of the proof: **Church-Turing thesis.**
*Any intuitive notion for a "computer" that you can come up with will be no more powerful than a Turing machine or than lambda calculus. That is, most models of computation are equivalent.*
*Turing-complete* means capable of simulating Turing machines.
Lambda calculus is Turing-complete (proof: later), and Turing machines can simulate lambda calculus.
Some others:

- *Turing machines* are Turing-complete
- *Scheme* is Turing-complete
- *Minecraft* is Turing-complete
- *Conway's Game of Life* is Turing-complete
- *Wolfram's Rule 110 cellular automaton* is Turing-complete

# Does not compute?

- Are there problems which our notion of computing cannot solve?

# Does not compute?

- Are there problems which our notion of computing cannot solve?
- Reworded: are there *functions* that cannot be computed?

# Does not compute?

- Are there problems which our notion of computing cannot solve?
- Reworded: are there *functions* that cannot be computed?
- Consider functions which map integers to integers.

## Does not compute?

- Are there problems which our notion of computing cannot solve?
- Reworded: are there *functions* that cannot be computed?

- Consider functions which map integers to integers.
- Can write out a function f as the infinite list of integers $f(0)$, $f(1)$, $f(2)$...

## Does not compute?

- Are there problems which our notion of computing cannot solve?
- Reworded: are there *functions* that cannot be computed?

- Consider functions which map integers to integers.
- Can write out a function f as the infinite list of integers $f(0)$, $f(1)$, $f(2)$...
- Any program text can be written as a single number, joining together this list

# Does not compute?

- Suppose, for contradiction, you've made a program to compute each possible function

## Does not compute?

- Suppose, for contradiction, you've made a program to compute each possible function
- Put them in a big table, one function per row, one input per column

# Does not compute?

- Suppose, for contradiction, you've made a program to compute each possible function
- Put them in a big table, one function per row, one input per column
- Diagonalize!

# Does not compute?

- Suppose, for contradiction, you've made a program to compute each possible function
- Put them in a big table, one function per row, one input per column
- Diagonalize!
- We get a contradiction: here's a function that's not in your list.

# Does not compute?

- Suppose, for contradiction, you've made a program to compute each possible function
- Put them in a big table, one function per row, one input per column
- Diagonalize!
- We get a contradiction: here's a function that's not in your list.

**Theorem (Church, Turing)**: *These models of computation can't solve every problem.*

# How many uncomputable problems?

- Countably infinite: $\aleph_0$

## How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - The number of integers

# How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - The number of integers
  - The number of binary strings

## How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - The number of integers
  - The number of binary strings
  - The number of programs

# How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - The number of integers
  - The number of binary strings
  - The number of programs
- Uncountably infinite: $2^{\aleph_0}$

# How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - The number of integers
  - The number of binary strings
  - The number of programs
- Uncountably infinite: $2^{\aleph_0}$
  - The number of functions mapping from integer to integer

# How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - The number of integers
  - The number of binary strings
  - The number of programs
- Uncountably infinite: $2^{\aleph_0}$
  - The number of functions mapping from integer to integer
  - The number of sets of binary strings

# How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - The number of integers
  - The number of binary strings
  - The number of programs
- Uncountably infinite: $2^{\aleph_0}$
  - The number of functions mapping from integer to integer
  - The number of sets of binary strings
  - The number of problem specifications

# Does not compute: Halting Problem

*Okay, but can you give me an example?*

# Does not compute: Halting Problem

*Okay, but can you give me an example?*

- We've seen our programs create infinite lists and infinite loops

# Does not compute: Halting Problem

*Okay, but can you give me an example?*

- We've seen our programs create infinite lists and infinite loops
- Can we write a program to check if an expression will return a value?

# Does not compute: Halting Problem

*Okay, but can you give me an example?*

- We've seen our programs create infinite lists and infinite loops
- Can we write a program to check if an expression will return a value?

```
(define (halt? p)
    ; ...
)
```

# Aside: what does this do?

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

# Aside: what does this do?

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

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

# Aside: what does this do?

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

= ((lambda (x) (x x))
   (lambda (x) (x x)))

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

# Aside: what does this do?

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

= ((lambda (x) (x x))
   (lambda (x) (x x)))

= ((lambda (x) (x x))
   (lambda (x) (x x)))

= ...
```

# Does not compute: Halting Problem

Contradiction!

# Does not compute: Halting Problem

Contradiction!

```
(define (troll)
  (if (halt? troll)
      ; if halts? says we halt, infinite-loop
      ((lambda (x) (x x)) (lambda (x) (x x)))
      ; if halts? says we dont, return a value
      \#f))
```

# Does not compute: Halting Problem

Contradiction!

```
(define (troll)
  (if (halt? troll)
      ; if halts? says we halt, infinite-loop
      ((lambda (x) (x x)) (lambda (x) (x x)))
      ; if halts? says we dont, return a value
      \#f))

    (halt? troll)
```

# Does not compute: Halting Problem

Contradiction!

```
(define (troll)
  (if (halt? troll)
      ; if halts? says we halt, infinite-loop
      ((lambda (x) (x x)) (lambda (x) (x x)))
      ; if halts? says we dont, return a value
      \#f))

    (halt? troll)
```

Halting Problem is undecidable for Turing Machines and thus all programming languages. (Turing, 1936)

# Does not compute: Halting Problem

Contradiction!

```
(define (troll)
  (if (halt? troll)
      ; if halts? says we halt, infinite-loop
      ((lambda (x) (x x)) (lambda (x) (x x)))
      ; if halts? says we dont, return a value
      \#f))

    (halt? troll)
```

Halting Problem is undecidable for Turing Machines and thus all programming languages. (Turing, 1936)
Want to learn more computability theory? See 18.400J/6.045J or 18.404J/6.840J (Sipser).

# The Source of Power

What's the minimal set of Scheme syntax that you need to achieve Turing-completeness?

# The Source of Power

What's the minimal set of Scheme syntax that you need to achieve Turing-completeness?

- `define`
- `set!`
- numbers
- strings
- `if`
- recursion
- `cons`
- booleans
- `lambda`

# Cons cells?

```
(define (cons a b)
 (lambda (c)
 (c a b)))
```

# Cons cells?

```
(define (cons a b)
 (lambda (c)
 (c a b)))

(define (car p)
 (p (lambda (a b) a)))
```

# Cons cells?

```
(define (cons a b)
 (lambda (c)
 (c a b)))

(define (car p)
 (p (lambda (a b) a)))

(define (cdr p)
 (p (lambda (a b) b)))
```

# Booleans?

```
(define true
 (lambda (a)
 (lambda (b)
 a)))
```

# Booleans?

```
(define true
 (lambda (a)
 (lambda (b)
 a)))

(define false
 (lambda (a)
 (lambda (b)
 b)))
```

# Booleans?

```
(define true
 (lambda (a)
 (lambda (b)
 a)))

(define false
 (lambda (a)
 (lambda (b)
 b)))

(define if (lambda (test then else)
 ((test then) else))
```

# Booleans?

```
(define true
 (lambda (a)
 (lambda (b)
 a)))

(define false
 (lambda (a)
 (lambda (b)
 b)))

(define if (lambda (test then else)
 ((test then) else))
```

Also try: `and`, `or`, `not`

# Numbers?

Number N: A procedure which takes in a successor function $s$ and a zero $z$, and returns the successor applied to the zero N times.

- For example, 3 is represented as $(s\ (s\ (s\ z)))$, given $s$ and $z$
- This technique: *Church numerals*

## Numbers?

```
(define (church-0
 (lambda (s)
 (lambda (z)
 z)))
```

# Numbers?

```
(define (church-0
 (lambda (s)
 (lambda (z)
 z)))
```

```
(define (church-1
 (lambda (s)
 (lambda (z)
 (s z))))
```

## Numbers?

```
(define (church-0
 (lambda (s)
 (lambda (z)
 z)))

(define (church-1
 (lambda (s)
 (lambda (z)
 (s z))))

(define (church-2
 (lambda (s)
 (lambda (z)
 (s (s z)))))
```

# Numbers?

```
(define (church-inc n)
 (lambda (s)
 (lambda (z)
 (s ((n s) z))))))
```

# Numbers?

```
(define (church-inc n)
 (lambda (s)
 (lambda (z)
 (s ((n s) z))))))

(define (church-add a b)
 (lambda (s)
 (lambda (z)
 ((a s) ((b s) z))))
```

# Numbers?

```
(define (church-inc n)
 (lambda (s)
 (lambda (z)
 (s ((n s) z))))))

(define (church-add a b)
 (lambda (s)
 (lambda (z)
 ((a s) ((b s) z))))

(define (also-church-add a b)
 ((a church-inc) b))))
```

# Numbers?

```
(define (church-inc n)
 (lambda (s)
 (lambda (z)
 (s ((n s) z))))))

(define (church-add a b)
 (lambda (s)
 (lambda (z)
 ((a s) ((b s) z))))

(define (also-church-add a b)
 ((a church-inc) b))))
```

For fun: Write `decrement`, write `multiply`.

## Let, define?

**Use lambdas.**

# Let, define?

**Use lambdas.**

```
(define x 4)
(...stuff)
```

# Let, define?

**Use lambdas.**

```
(define x 4)
(...stuff)
```

becomes...

```
((lambda (x)
  (...stuff)
) 4)
```

## Let, define?

A problem arises!

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

## Let, define?

A problem arises!

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

Why? `(lambda (fact) ...)  (...definition of fact...)` fails! `fact` is not yet defined when called in its function body.

# Let, define?

A problem arises!

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

Why? `(lambda (fact) ...)  (...definition of fact...)`
fails! `fact` is not yet defined when called in its function body.
If we can't name "fact" how do we use it in the recursive call?

# Factorial again

Run it with a copy of itself.

```
(define (fact inner-fact n)
    (if (= n 0)
        1
        (* n
           (inner-fact inner-fact (- n 1)))))
```

# Factorial again

Run it with a copy of itself.

```
(define (fact inner-fact n)
    (if (= n 0)
        1
        (* n
           (inner-fact inner-fact (- n 1)))))
```

Now, `(fact fact 4)` works!

# Now without define

(fact fact 4) becomes:

## Now without define

(fact fact 4) becomes:

```
((lambda (fact n)
    (if (= n 0)
        1
        (* n (fact fact (- n 1)))))
 (lambda (fact n)
    (if (= n 0)
        1
        (* n (fact fact (- n 1)))))
 4)
```

# Messy. Can we do better?

Let's define fact-inner as:

# Messy. Can we do better?

Let's define fact-inner as:

```
(lambda (fact)
    (lambda (n)
        (if (= n 0)
            1
            (* n (fact (- n 1))))))
```

# Messy. Can we do better?

Let's define fact-inner as:

```
(lambda (fact)
    (lambda (n)
        (if (= n 0)
            1
            (* n (fact (- n 1))))))
```

Huh - what's (fact-inner fact)?

# Messy. Can we do better?

Let's define fact-inner as:

```
(lambda (fact)
    (lambda (n)
        (if (= n 0)
            1
            (* n (fact (- n 1))))))
```

Huh - what's (fact-inner fact)? `(fact-inner fact) = fact`.

# Messy. Can we do better?

Let's define fact-inner as:

```
(lambda (fact)
    (lambda (n)
        (if (= n 0)
            1
            (* n (fact (- n 1))))))
```

Huh - what's (fact-inner fact)? `(fact-inner fact) = fact`.
**A fixed point!**

## Producing Fixed Points

Now let's define `Y` as:

```
(lambda (f)
    ((lambda (g) (f (g g)))
     (lambda (g) (f (g g)))))
```

We'll prove that `(Y f) = (f (Y f))`

## Producing Fixed Points

Now let's define `Y` as:

```
(lambda (f)
    ((lambda (g) (f (g g)))
     (lambda (g) (f (g g)))))
```

We'll prove that `(Y f) = (f (Y f))` – that we can use `Y` to create fixed points.

# Producing Fixed Points

From the problem before: we want `(fact-inner fact)`.

## Producing Fixed Points

From the problem before: we want `(fact-inner fact)`.

```
(define Y (lambda (f)
            ((lambda (g) (f (g g)))
             (lambda (g) (f (g g))))))
;; For convenience:
;;     H := (lambda (g) (f (g g)))

;; Is (fact-inner fact) = (Y fact-inner)?
;; (Y fact-inner)
;; = (H H)                  ; (with f = fact-inner)
;; = (fact-inner (g g))
;; = (fact-inner (H H))
;; = (fact-inner (Y fact-inner))
;; = (fact-inner fact)  ; Success!
```

# Producing Fixed Points

Now we can define `fact` as follows:

# Producing Fixed Points

Now we can define `fact` as follows:

```
(Y (lambda (fact-inner)
     (lambda (n)
          (if (= n 0)
              1
              (* n (fact-inner (- n 1)))))))
```

# Producing Fixed Points

Now we can define `fact` as follows:

```
(Y (lambda (fact-inner)
     (lambda (n)
          (if (= n 0)
              1
              (* n (fact-inner (- n 1)))))))
```

Can create `fact` without using `define`!

## Producing Fixed Points

Now we can define `fact` as follows:

```
(Y (lambda (fact-inner)
     (lambda (n)
         (if (= n 0)
             1
             (* n (fact-inner (- n 1)))))))
```

Can create `fact` without using `define`!
Can create all of Scheme using just `lambda`!

# Producing Fixed Points

Now we can define `fact` as follows:

```
(Y (lambda (fact-inner)
     (lambda (n)
          (if (= n 0)
              1
              (* n (fact-inner (- n 1)))))))
```

Can create `fact` without using `define`!
Can create all of Scheme using just `lambda`!
**Lambda calculus is Turing-complete!**

## Producing Fixed Points

Now we can define `fact` as follows:

```
(Y (lambda (fact-inner)
     (lambda (n)
         (if (= n 0)
             1
             (* n (fact-inner (- n 1)))))))
```

Can create `fact` without using `define`!
Can create all of Scheme using just `lambda`!
**Lambda calculus is Turing-complete!** Church-Turing thesis!

## Fun links

- https://xkcd.com/505/
- http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html
- https://youtu.be/1X21HQphy6I
- https://youtu.be/My8AsV7bA94
- https://youtu.be/xP5-iIeKXE8
- https://en.wikipedia.org/wiki/Rule_110