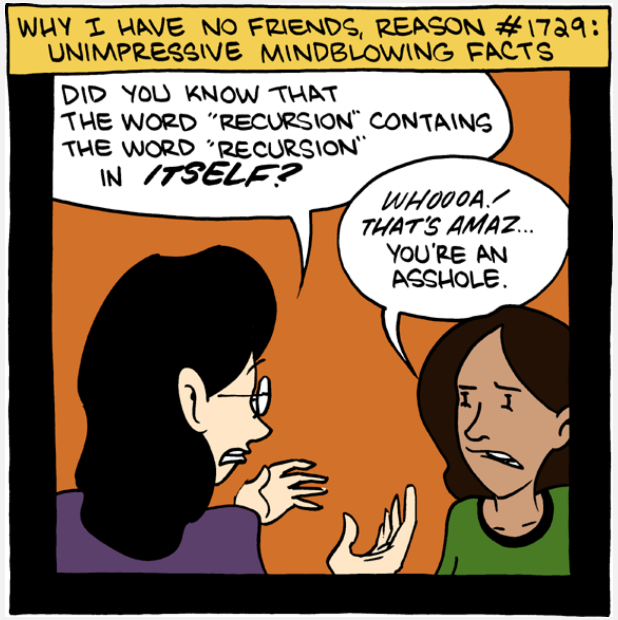
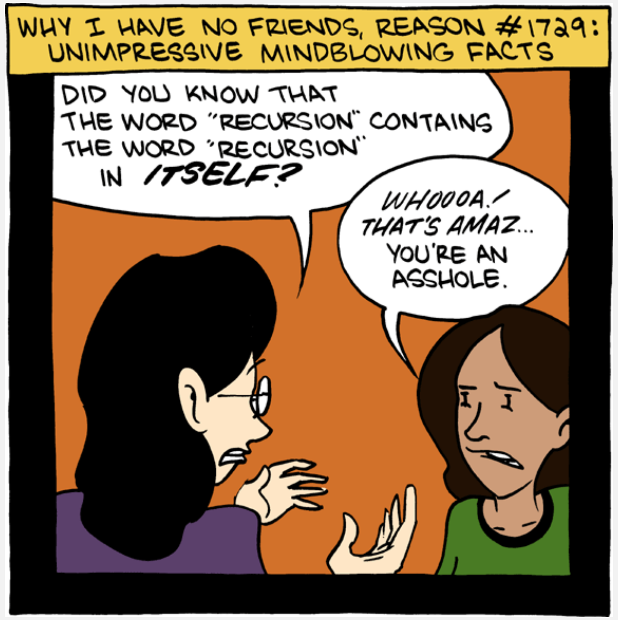
CS420, Ch 6.1:
Turing Machines and Recursion

Mon, April 12, 2021



Announcements

- HW8 past due
- HW9 due Sun 4/18 11:59pm EST
- **Last Unit**: Time Complexity
  - P, NP, NP-Completeness …
  - Starting Wed 4/14
- **Reminder**: no class next Monday 4/19
  - Patriot’s Day



Recursion in Programming

```
(define (factorial n)
  (if (zero? n)
      1
      (* n (factorial (sub1 n)))))
```

In most programming languages,
you can call a function recursively,
<u>even before it's completely defined</u>!

Turing Machines and Recursion

- We've been saying: "**A Turing machine models programs.**"

- **Q**: Is a recursive program modeled by a Turing machine?

- **A**: Yes!
  - But it's not explicit.
  - In fact, it's a little complicated.
  - Need to prove it …

- **Today**: The Recursion Theorem

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Where's the recursion in this definition???

The Recursion Theorem

$A_{\mathsf{TM}} = \{\langle M, w \rangle |\ M \text{ is a TM and } M \text{ accepts } w\}$

- You can write a TM description like this:

Example Usage

<u>Prove</u> A_{TM} is undecidable, by <u>contradiction</u>:

assume that Turing machine H decides A_{TM}

$B =$ "On input w:

1. Obtain, via the recursion theorem, own description $\langle B \rangle$.
2. Run H on input $\langle B, w \rangle$.
3. Do the opposite of what H says. That is, *accept* if H rejects and *reject* if H accepts."

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*
M_3	*reject*	*reject*	*reject*	*reject*		*reject*
M_4	*accept*	*accept*	*reject*	*reject*		*accept*
$\vdots$			$\vdots$		$\ddots$	
D	*reject*	*reject*	*accept*	*accept*		?

This is the non-existent "*D*" machine, the TM that **does the opposite of itself**, defined using recursion!

How can a TM “obtain it’s own description?”

How does a TM even know about “itself” before it’s completely defined?

A (Simpler) Coding Exercise

Your task:

- Write a program that, without using recursion, prints itself.
  - Such a program obviously must have knowledge about "itself"

- An example, in English:

function "parameter"

"function"

Print out two copies of the following, the second one in quotes:
"Print out two copies of the following, the second one in quotes:"

"argument" (the function, encoded as string)

- A program that does this knows about "itself",
  - but it does not explicitly use recursion!

Interlude: Lambda

- λ = anonymous function value, e.g. `(λ (x) x)`
  - **C++**: `[](int x){ return x; }`
  - **Java**: `(x) -> { return x; }`
  - **Python**: `lambda x : x`
  - **JS**: `(x) => { return x; }`

My Self-Reproducing Program

Print out two copies of the following, the second one in quotes:
“Print out two copies of the following, the second one in quotes:”

“function” “parameter” “argument”

```
((λ (the-following) (print2x-2ndquoted the-following))
 "(λ (the-following) (print2x-2ndquoted the-following))")
```

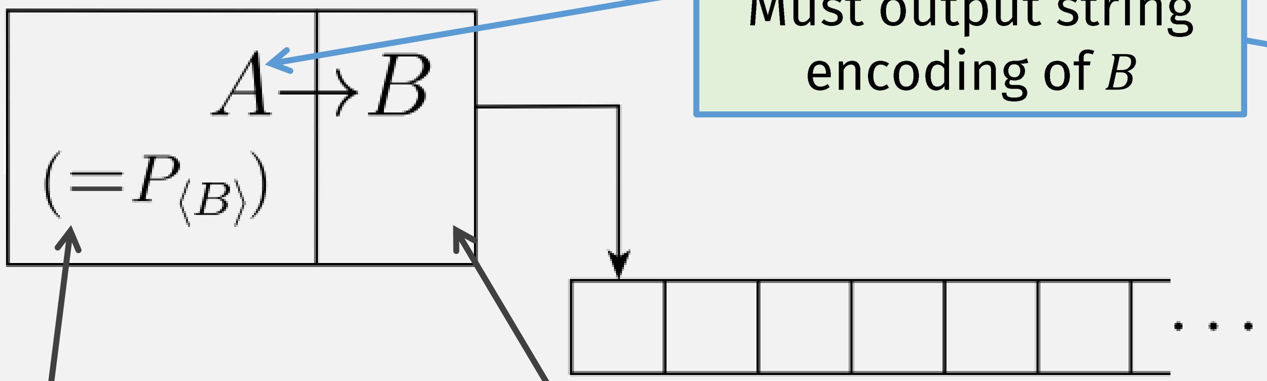
```
(define (print2x-2ndquoted str)
  (printf "(~a\n ~v)\n" str str))
```

(could have inlined this)

First copy

Second copy (quoted)

Self-Reproducing Turing Machine



Must output string encoding of B

q creates $P_{<B>}$

The following TM Q computes $q(w)$.

$Q =$ "On input string w:
1. Construct the following Turing machine P_w.
   $P_w =$ "On any input:
   1. Erase input.
   2. Write w on the tape.
   3. Halt."
2. Output $\langle P_w \rangle$."

$P_{<B>}$ is TM that writes <B> on tape
(TMs pass args by writing it on tape)

"argument"
(the function, encoded as string)

"function"

$B =$ "On input $\langle M \rangle$, where M is a portion of a TM:
Second copy (quoted) — Compute $q(\langle M \rangle)$.
First copy — Combine the result with $\langle M \rangle$ to make a complete TM.
3. Print the description of this TM and halt."

"parameter"

Print out two copies of the following, the second on in quotes:
"Print out two copies of the following, the second on in quotes:"

Recursive Program that Prints Itself

$SELF$ = "On any input:

1. Obtain, via the recursion theorem, own description $\langle SELF \rangle$.
2. Print $\langle SELF \rangle$."

This whole program is "itself"

Just this part is also "itself"

```
((λ (the-following) (print2x-2ndquoted the-following))
 "(λ (the-following) (print2x-2ndquoted the-following))")
```

- Our program contains "itself" even though it has no recursion!
- **We don't need explicit recursion to write recursive programs!**
- Can we write a program that does something other than print "itself"?

Other nonrecursive programs using "itself"

- Program that prints "itself":

```
((λ (the-following) (print2x-2ndquoted the-following))
 "(λ (the-following) (print2x-2ndquoted the-following))")
```

- Program that runs "itself" repeatedly (ie, it loops):

```
((λ (x) (x x))
 (λ (x) (x x)))
```

Call arg fn with itself as arg

Don't convert arg to string

- Loop, but call some other function f each time:

```
(λ (f)
  ((λ (x) (f (x x)))
   (λ (x) (f (x x)))))
```

➡

```
(λ (f)
  ((λ (x) (f (λ (v) ((x x) v))))
   (λ (x) (f (λ (v) ((x x) v))))))
```

Need this extra lambda bc we want to call **f** first before looping

- **None of these programs use explicit recursion!**

Y combinator

The Recursion Theorem, Formally

Recursion theorem Let T be a Turing machine that computes a function $t\colon \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. There is a Turing machine R that computes a function $r\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$r(w) = t\big(\langle R\rangle, w\big).$$

<u>In English</u>:

- If you want TM R that includes a step "obtain own description" …
- … instead create TM T with an extra "itself" argument …
- … then construct R from T

Recursion Theorem, A Concrete Example

- If you want:
  - Recursive fn

```
(define (factorial n) ;; R
  (if (zero? n)
      1
      (* n (factorial (sub1 n)))))
```

- Instead create:
  - Non-recursive fn

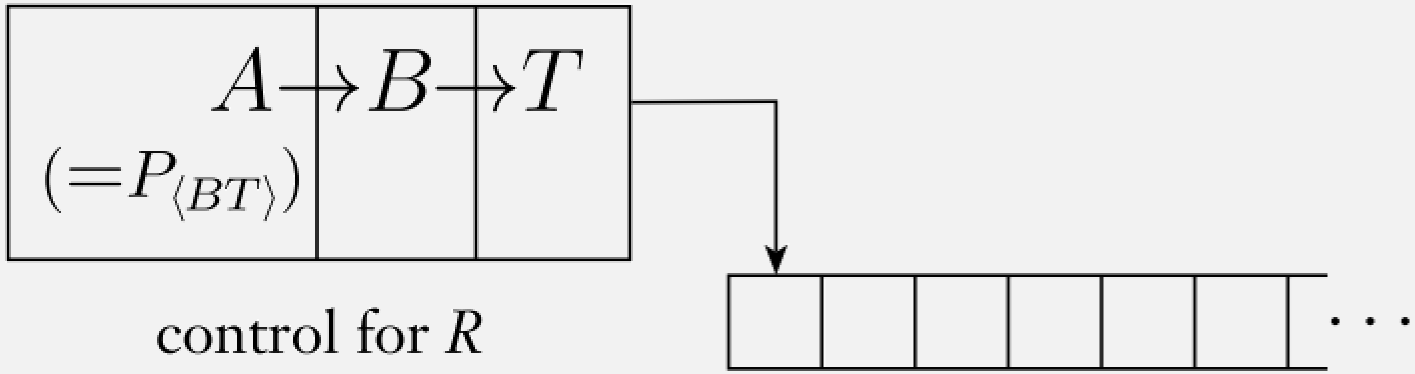
```
(define (factorial/itself ITSELF n) ;; T
  (if (zero? n)
      1
      (* n (ITSELF (sub1 n)))))
```

Recursion Theorem says you can convert

But how??

Recursion Theorem, Proof

- To convert a “T” to “R”:



1. Construct *A* = program constructing <*BT*>, and
2. Pass result to *B* (from before),
3. which passes “itself” to *T*

- Compare with *SELF*:

Print out two copies of the following, the second on in quotes: *B*
“Print out two copies of the following, the second on in quotes:” *A*

Recursion Theorem Proof: Coding Demo

- Program that passes "itself" to another function:

Y combinator

```
(λ (f)
  ((λ (x) (f (λ (v) ((x x) v))))
   (λ (x) (f (λ (v) ((x x) v))))))
```

Pass to

- Function that needs "itself"

```
(define (factorial/itself ITSELF n) ;; T
  (if (zero? n)
      1
      (* n (ITSELF (sub1 n)))))
```

Fixed Points

- A value x is a fixed point of a function f if $f(x) = x$

Recursion Theorem and Fixed Points

THEOREM 6.8

Let $t\colon \Sigma^* \longrightarrow \Sigma^*$ be a computable function. Then there is a Turing machine F for which $t(\langle F \rangle)$ describes a Turing machine equivalent to F. Here we'll assume that if a string isn't a proper Turing machine encoding, it describes a Turing machine that always rejects immediately.

In this theorem, t plays the role of the transformation, and F is the fixed point.

PROOF Let F be the following Turing machine.

F = "On input w:
1. Obtain, via the recursion theorem, own description $\langle F \rangle$.
2. Compute $t(\langle F \rangle)$ to obtain the description of a TM G.
3. Simulate G on w."

Clearly, $\langle F \rangle$ and $t(\langle F \rangle) = \langle G \rangle$ describe equivalent Turing machines because F simulates G.

- I.e., Recursion theorem says:
  - "every TM that computes on TMs has a fixed point"
  - As code: "every function on functions has a fixed point"

Y Combinator

- `mk-recursive-fn` = a "fixed point finder"

```
(define mk-recursive-fn
  (λ (f)
    ((λ (x) (f (λ (v) ((x x) v))))
     (λ (x) (f (λ (v) ((x x) v)))))))
```

Summary: Where "Recursion" Comes From

- TMs are powerful enough to:
  1. Construct other TMs
  2. Simulate other TMs

- That's enough to achieve recursion!

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Where's the recursion???

Check-in Quiz 4/12

On gradescope