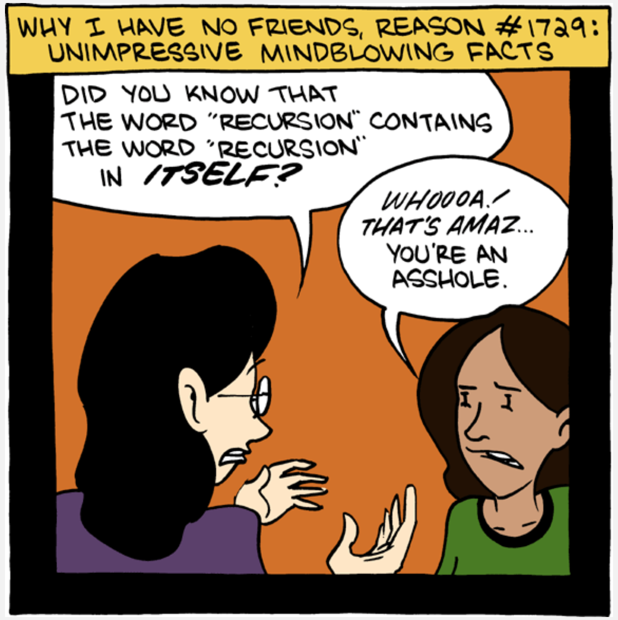
CS420:
Turing Machines and Recursion

Mon, November 16, 2020



HW 8 Questions?

HW announcements

- HW5 grades released

- Reminder: Cite your sources and collaborators!
  - In README
  - Will be penalized in future assignments
  - May have to present in class to demonstrate understanding

Past HW Review

- Using non-determinism properly:
  - “Non-deterministically split the (input) string” ???
  - “Non-deterministically split the (input) string into all possible pairs”

- Being careful with looping in TMS:
  - Let *M1* and *M2* recognize *L1* and *L2*, respectively
  - Let *S* = TM recognizing union of *L1* and *L2*
  - *S* = On input *x*:
    - Run *M1* on *x*, accept if accept, else
    - Run *M2* on *x*, accept if accept, else reject
  - If *M1* loops and *M2* accepts *x*, *S* wrongly loops when it should accept

Programmers Use Recursion


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

Turing Machines and Recursion

- We’ve been saying: “A Turing machine is just a program.”

- Q: Is a recursive program still a Turing machine?

- A: Yes!
  - But it’s not explicit.
  - In fact, it’s a little complicated.
  - Need to prove it:
    - The Recursion Theorem

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, 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. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Where’s the recursion???

The Recursion Theorem

- You can write a TM description like this:
  - Prove A_{TM} is undecidable by contradiction, 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 a valid (but non-existent) TM that does the opposite of itself!

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

How can 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.

- An example, in English:

"function"

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

"argument"

- This "program" knows about "itself"

- A program can know about "itself", without recursion!

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

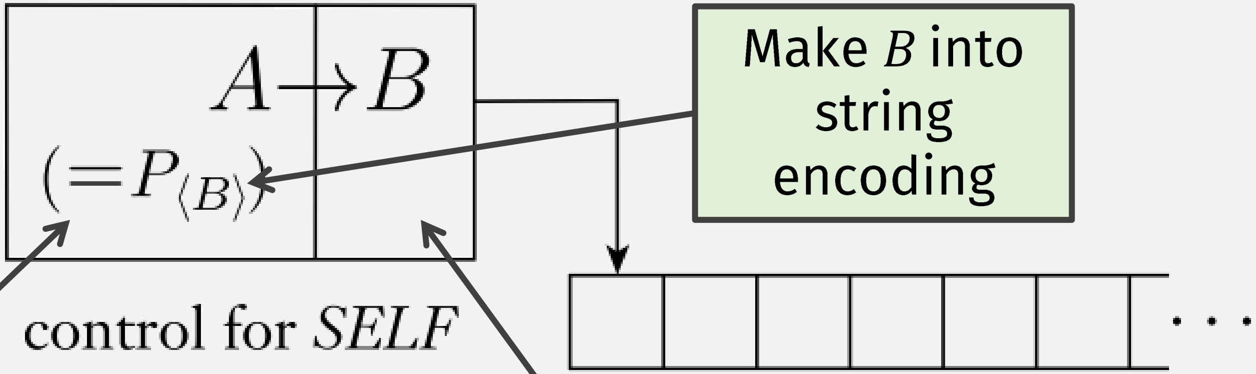
Print out two copies of the following, the second on in quotes:
“Print out two copies of the following, the second on in quotes:”

“function”

“argument”

```
((λ(x) (printf "(~a\n ~v)\n" x x))
 "(λ(x) (printf \"(~a\\n ~v)\\n\" x x)")
```

Self-Reproducing Turing Machine



Make B into string encoding

"argument"

"function"

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 $\langle P_w \rangle$."

TMs pass args by putting it on tape

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

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

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(x) (printf "(~a\n ~v)\n" x x))
 "(λ(x) (printf \"(~a\\n ~v)\\n\" x x)")
```

- Our program contains "itself" even though it has no recursion!
- What if we want to do something other than printing "itself"?

Other nonrecursive programs using "itself"

- Program that prints "itself":

```
((λ(x) (printf "(~a\n ~v)\n" x x))
 "(λ(x) (printf \"(~a\\n ~v)\\n\" x x)")
```

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

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

- Program that passes "itself" to another function:

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

- Still no "recursion" in sight!

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,

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

- In English:
  - If you want TM R that includes "obtain own description" …
  - … instead create TM T with an explicit "itself" argument …
  - … then you can construct R from T

Recursion Theorem, A Concrete Example

- If you want:

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

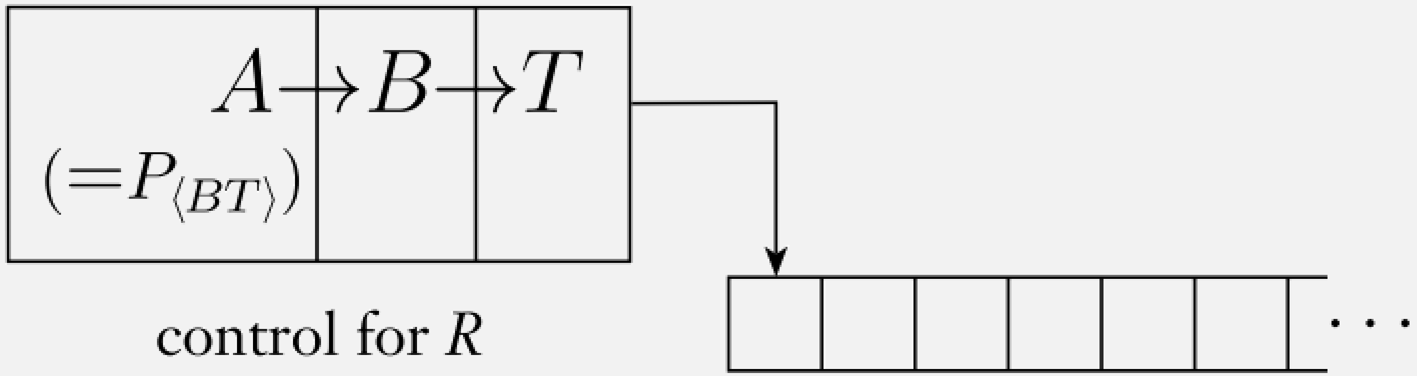
- Instead create:

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

But how to convert?

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:

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

- Function that needs "itself"

Pass to

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

Fixed Points

- A value x is a <u>fixed point</u> 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))))))
```

- `mk-recursive-fn` alternate name: Y combinator!

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, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, 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. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Where's the recursion???

Check-in Quiz 11/16

On gradescope

End of Class Survey 11/16

See course website