Automata Theory

CS411-2015F-14

Counter Machines & Recursive Functions

David Galles

Department of Computer Science
University of San Francisco

14-0: Counter Machines

- Give a Non-Deterministic Finite Automata a *counter*
  - Increment the counter
  - Decrement the counter
  - Check to see if the counter is zero

Counter Machines

- A Counter Machine $M = (K, \Sigma, \Delta, s, F)$
  - K is a set of states
  - Σ is the input alphabet
  - $s \in K$ is the start state
  - $F \subset K$ are Final states
  - $\Delta \subseteq ((K \times (\Sigma \cup \epsilon) \times \{zero, \neg zero\}) \times (K \times \{-1, 0, +1\}))$
- Accept if you reach the end of the string, end in an accept state, and have an empty counter.

14-2: Counter Machines

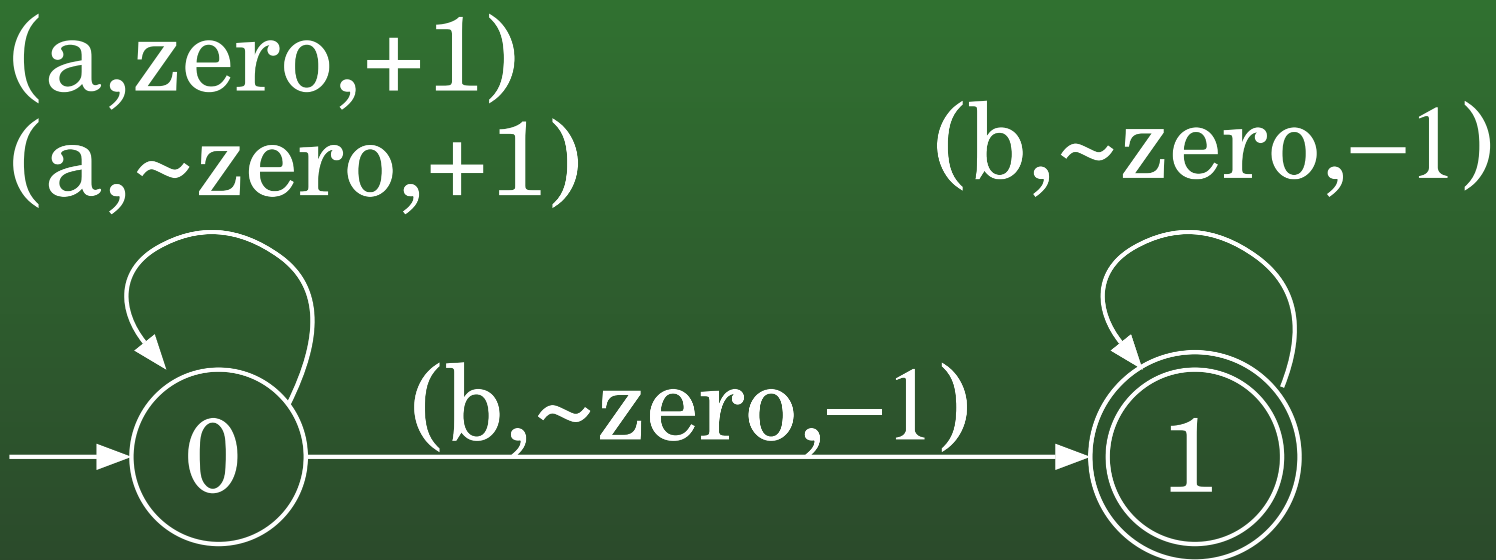
- Give a Non-Deterministic Finite Automata a *counter*
  - Increment the counter
  - Decrement the counter
  - Check to see if the counter is zero
- Do we have more power than a standard NFA?

14-3: Counter Machines

- Give a counter machine for the language $a^n b^n$

14-4: Counter Machines

- Give a counter machine for the language $a^n b^n$

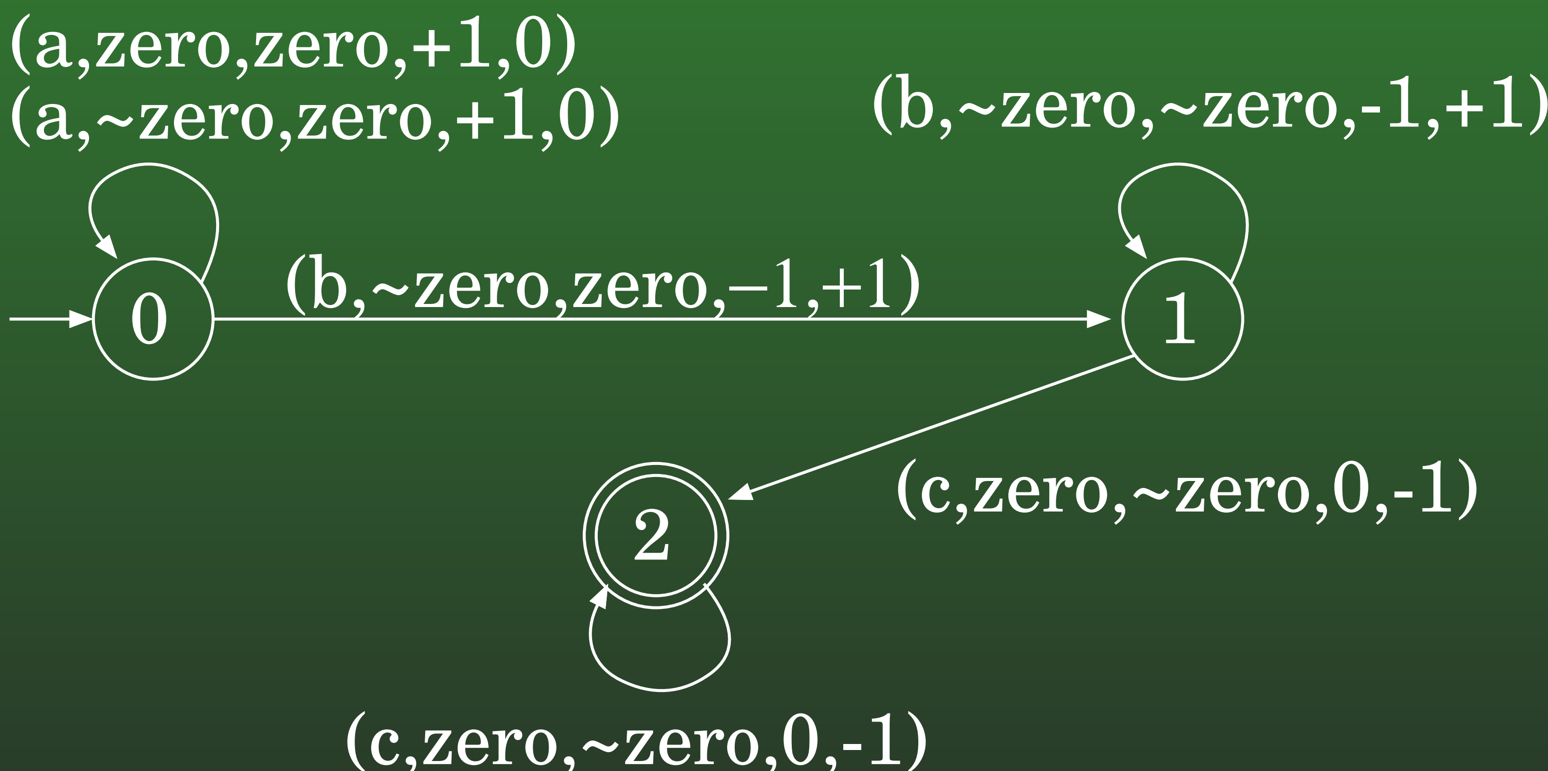


14-5: Counter Machines

- Give a 2-counter machine for the language $a^n b^n c^n$
  - Straightforward extension – examine (and change) two counters instead of one.

14-6: Counter Machines

- Give a 2-counter machine for the language $a^n b^n c^n$

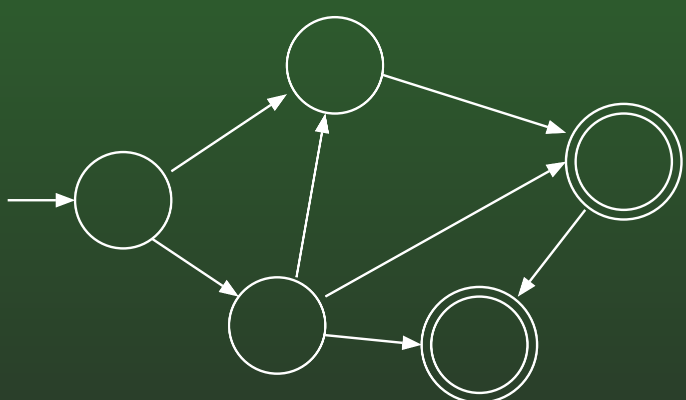


14-7: Counter Machines

- Our counter machines only accept if the counter is zero
  - Does this give us any *more* power than a counter machine that accepts whenever the end of the string is reached in an accept state?
  - That is, given a counter machine M that accepts only strings that both drive the machine to an accept state, and leave the counter empty, can we create a counter machine M' that accepts all strings that drive the machine to an accept state (regardless of the contents of the counter) so that $L[M] = L[M']$?

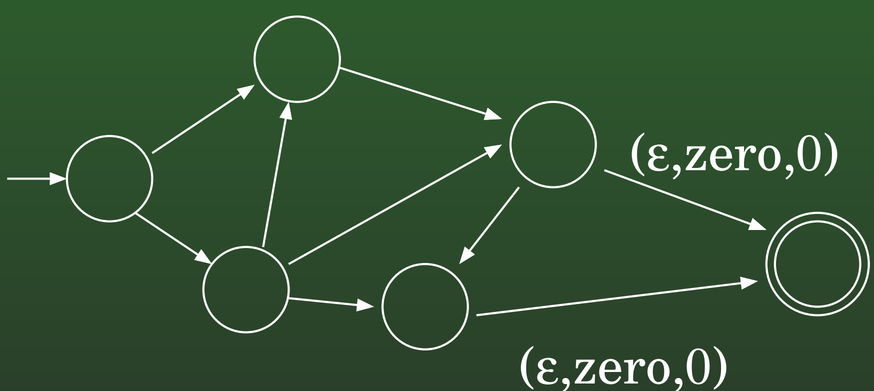
14-8: Counter Machines

- Our counter machines only accept if the counter is zero
  - Does this give us any *more* power than a counter machine that accepts whenever the end of the string is reached in an accept state?



14-9: Counter Machines

- Our counter machines only accept if the counter is zero
  - Does this give us any *more* power than a counter machine that accepts whenever the end of the string is reached in an accept state?

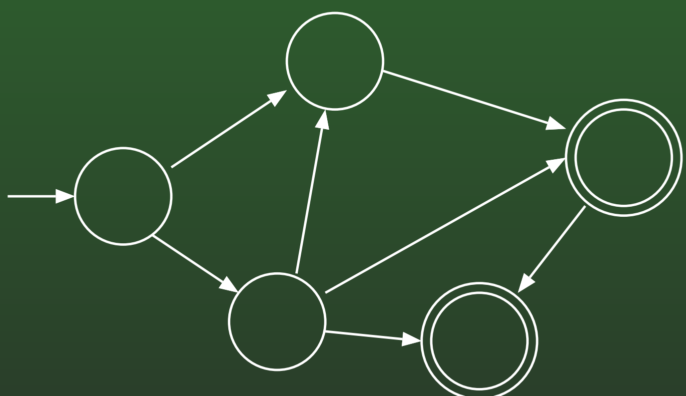


Counter Machines

- Our counter machines only accept if the counter is zero
  - Does this give us any *less* power than a counter machine that accepts whenever the end of the string is reached in an accept state?
  - That is, given a counter machine M that accepts all strings that drive the machine to an accept state (regardless of contents of counter), can we create a counter machine M' that accepts only strings that both drive the machine to an accept state *and* leave the counter empty, such that $L[M] = L[M']$?

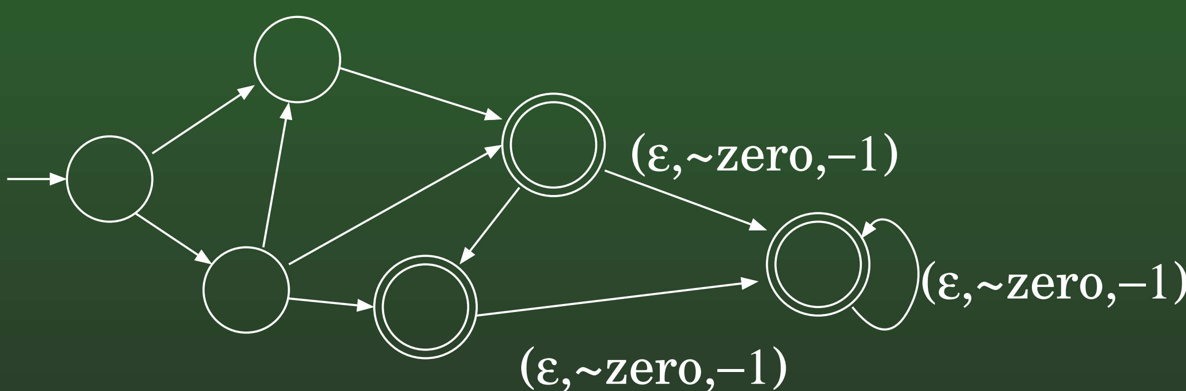
14-11: Counter Machines

- Our counter machines only accept if the counter is zero
  - Does this give us any *less* power than a counter machine that accepts whenever the end of the string is reached in an accept state?



14-12: Counter Machines

- Our counter machines only accept if the counter is zero
  - Does this give us any *less* power than a counter machine that accepts whenever the end of the string is reached in an accept state?



14-13: Counter Machines

- Give a Non-Deterministic Finite Automata *two* counters
- We can use two counters to simulate a stack
  - How?
  - *HINT:* We will simulate a stack that has two symbols, 0 and 1
  - *HINT2:* Think binary

14-14: Counter Machines

- We can use two counters to simulate a stack
  - One counter will store the contents of the stack
  - Other counter will be used as “scratch space”
- Stack will be represented as a binary number, with the top of the stack being the least significant bit
  - How can we push a 0?
  - How can we push a 1?

Counter Machines

- How can we push a 0?
  - Multiply the counter by 2
- How can we push a 1?
  - Multiply the counter by 2, and add 1

Counter Machines

- How can we multiply a counter by 2, if all we can do is increment
  - Remember, we have a “scratch counter”

14-17: Counter Machines

- How can we multiply a counter by 2, if all we can do is increment
  - Set the “Scratch Counter” to 0
  - While counter is not zero:
    - Decrement the counter
    - Increment the “Scratch Counter” twice

Counter Machines

- To Push a 0:
  - While Counter1 $\neq$ 0
    - Increment Counter2
    - Increment Counter2
    - Decrement Counter1
  - Swap Counter1 and Counter2

14-19: Counter Machines

- To Push a 1:
  - While Counter1 $\neq 0$
    - Increment Counter2
    - Increment Counter2
    - Decrement Counter1
  - Increment Counter2
  - Swap Counter1 and Counter2

14-20: Counter Machines

- To Pop:
  - While Counter1 $\neq 0$
    - Decrement Counter1
    - If Counter1 = 0, popped result is 1
    - Decrement Counter1
    - If Counter1 = 0, popped result is 0
    - Increment Counter2
  - Swap Counter1 and Counter2

14-21: Counter Machines

- How do we check if the simulated stack is empty?
  - We need to use 1 (not zero) to represent an empty stack (why?)
  - Stack is empty if (counter1$-1 = 0$)

14-22: Counter Machines

- Example

Stack Counter	Scratch Counter
1	0

- Stack counter starts out as 1 (represents empty stack)
- Scratch counter starts out as 0

14-23: Counter Machines

- Example

Stack Counter	Scratch Counter
1	0

- Push 0

Counter Machines

- Example

Stack Counter	Scratch Counter
0	1

- Decrement Stack Counter, increment scratch counter

14-25: Counter Machines

- Example

Stack Counter	Scratch Counter
0	10

- Decrement Stack Counter, increment scratch counter (twice)

Counter Machines

- Example

Stack Counter	Scratch Counter
0	10

- Swap Scratch Counter and Stack Counter

While Scratch Counter $\neq$ Stack Counter
    Decrement Scratch Counter
    Increment Stack Counter

Counter Machines

- Example

Stack Counter	Scratch Counter
10	0

- Swap Scratch Counter and Stack Counter

While Scratch Counter $\neq$ Stack Counter
    Decrement Scratch Counter
    Increment Stack Counter

14-28: Counter Machines

- Example

Stack Counter	Scratch Counter
10	0

- Push 1

14-29: Counter Machines

- Example

Stack Counter	Scratch Counter
1	1

- Decrement Stack Counter, increment scratch counter

14-30: Counter Machines

- Example

Stack Counter	Scratch Counter
1	10

- Decrement Stack Counter, increment scratch counter (twice)

14-31: Counter Machines

- Example

Stack Counter	Scratch Counter
0	11

- Decrement Stack Counter, increment scratch counter

14-32: Counter Machines

- Example

Stack Counter	Scratch Counter
0	100

- Decrement Stack Counter, increment scratch counter (twice)

Counter Machines

- Example

Stack Counter	Scratch Counter
0	101

- Add one to scratch counter (since pushing 1, not 0)

14-34: Counter Machines

- Example

Stack Counter	Scratch Counter
0	101

- Swap Scratch Counter and Stack Counter

While Scratch Counter $\neq$ Stack Counter
    Decrement Scratch Counter
    Increment Stack Counter

Counter Machines

- Example

Stack Counter	Scratch Counter
101	0

- Swap Scratch Counter and Stack Counter

While Scratch Counter $\neq$ Stack Counter
    Decrement Scratch Counter
    Increment Stack Counter

Counter Machines

- Example

Stack Counter	Scratch Counter
101	0

- Pop

Counter Machines

- Example

Stack Counter	Scratch Counter
100	0

- Decrement Stack counter

14-38: Counter Machines

- Example

Stack Counter	Scratch Counter
11	0

- Decrement Stack counter (twice)

Counter Machines

- Example

Stack Counter	Scratch Counter
11	1

- Increment Scratch counter

14-40: Counter Machines

- Example

Stack Counter	Scratch Counter
10	1

- Decrement Stack counter

14-41: Counter Machines

- Example

Stack Counter	Scratch Counter
1	1

- Decrement Stack counter (twice)

Counter Machines

- Example

Stack Counter	Scratch Counter
1	10

- Increment Scratch counter

14-43: Counter Machines

- Example

Stack Counter	Scratch Counter
0	10

- Decrement Stack counter

14-44: Counter Machines

- Example

Stack Counter	Scratch Counter
0	10

- Can't Decrement Stack counter a second time (empty), so popped value is 1

14-45: Counter Machines

- Example

Stack Counter	Scratch Counter
0	10

- Swap Scratch Counter and Stack Counter

While Scratch Counter $\neq$ Stack Counter
    Decrement Scratch Counter
    Increment Stack Counter

14-46: Counter Machines

- Example

Stack Counter	Scratch Counter
10	0

- Swap Scratch Counter and Stack Counter

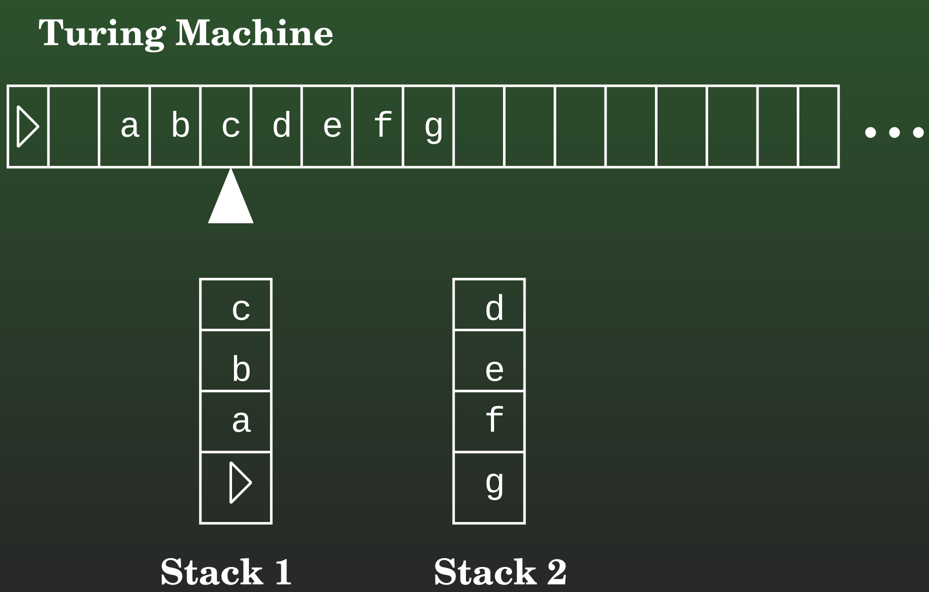
While Scratch Counter $\neq$ Stack Counter
    Decrement Scratch Counter
    Increment Stack Counter

Counter Machines

- Two counters can simulate a stack
- Four counters can simulate two stacks
- What can we do with two stacks?

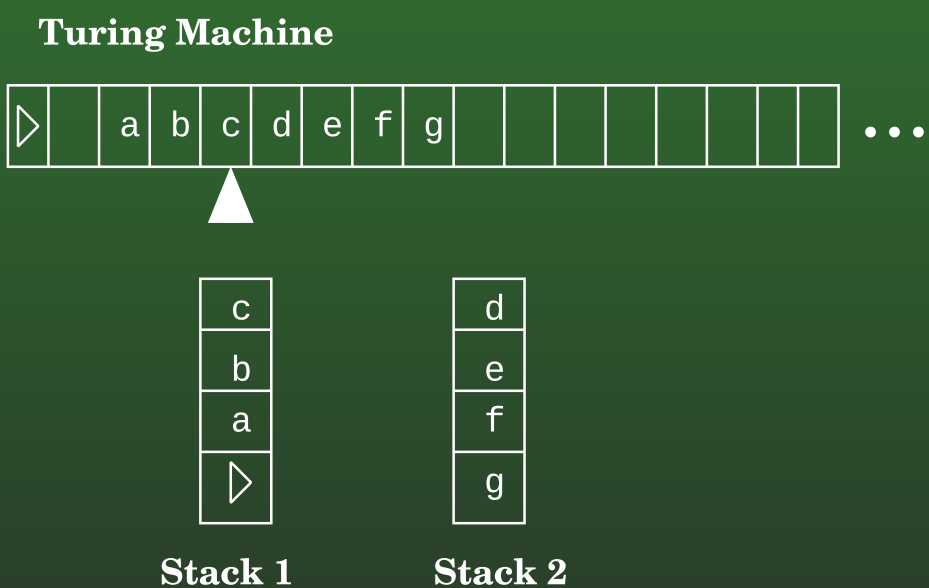
14-48: Counter Machines

- Two stacks can simulate a Turing Machine:
  - Stack 1: Everything to the left of the read/write head
  - Stack 2: Everything to the right of the read/write head
- Tape head points to top of Stack 1



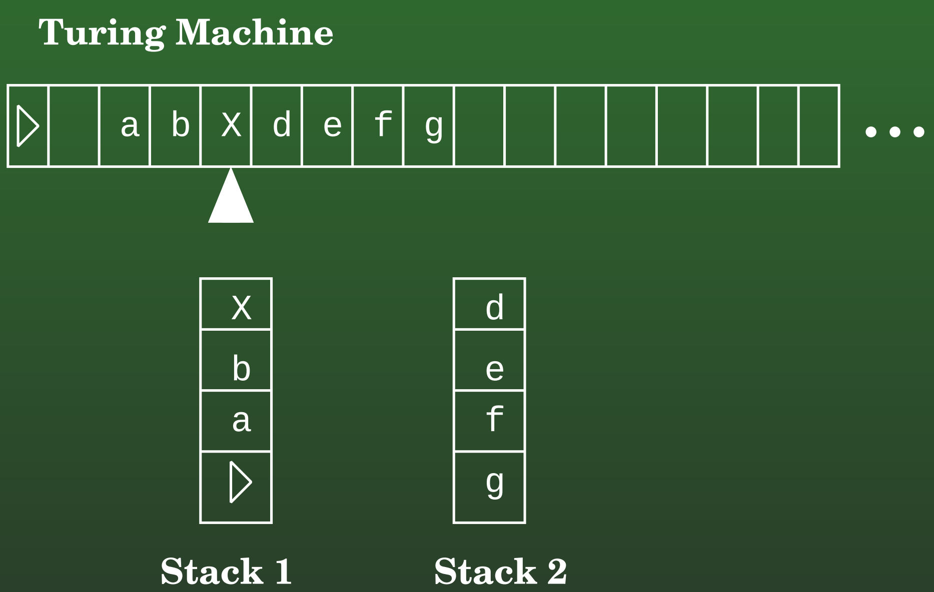
Counter Machines

- To write a new symbol at the Tape Head
  - Pop old value off the top of Stack 1
  - Push new value on the top of Stack 1



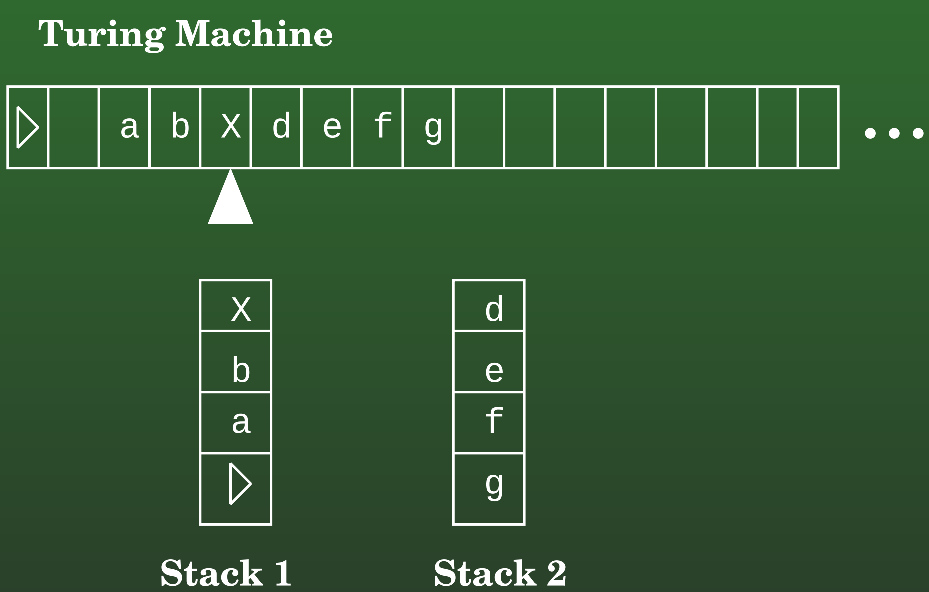
14-50: Counter Machines

- To write a new symbol at the Tape Head
  - Pop old value off the top of Stack 1
  - Push new value on the top of Stack 1



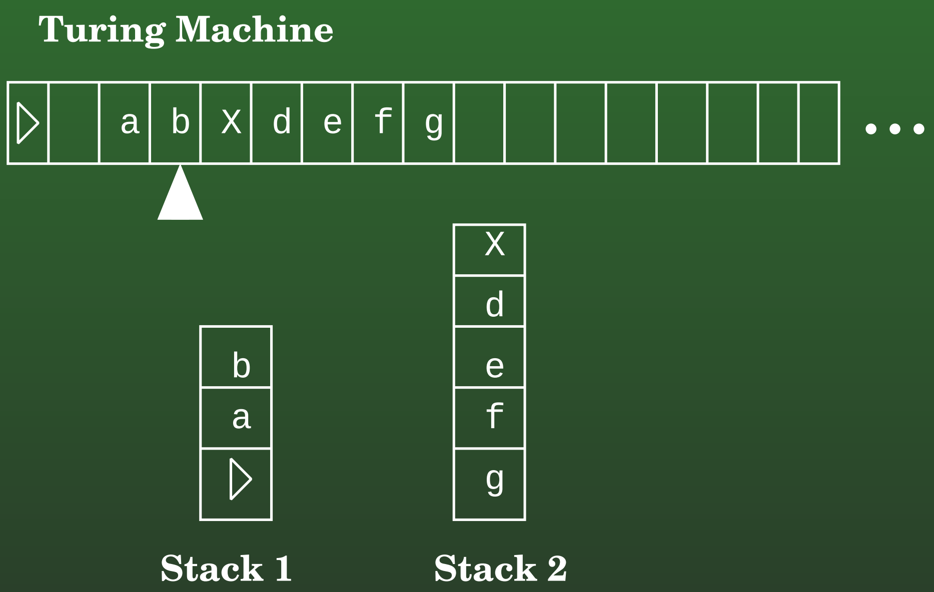
Counter Machines

- To move the tape head to the left
  - Pop symbol off Stack 1
  - Push it on Stack 2



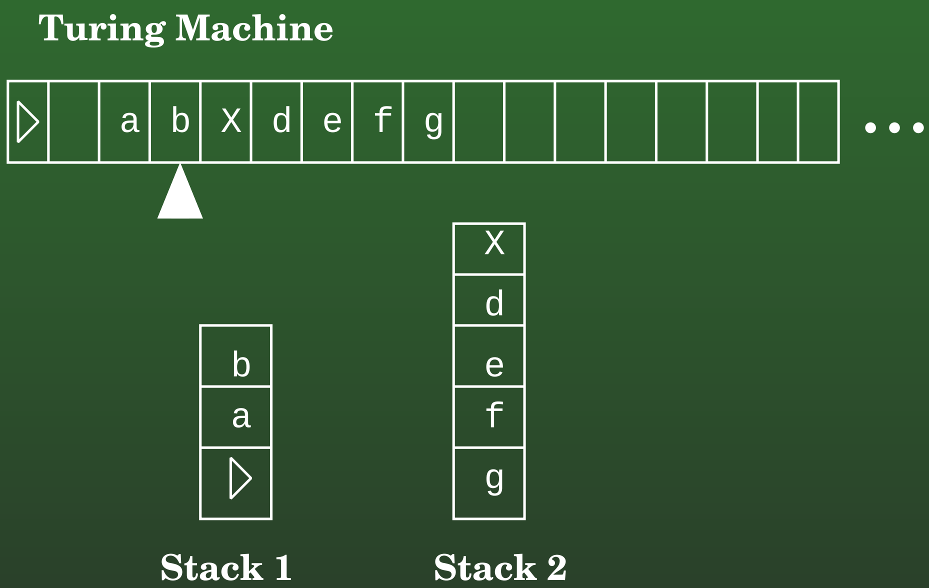
Counter Machines

- To move the tape head to the left
  - Pop symbol off Stack 1
  - Push it on Stack 2



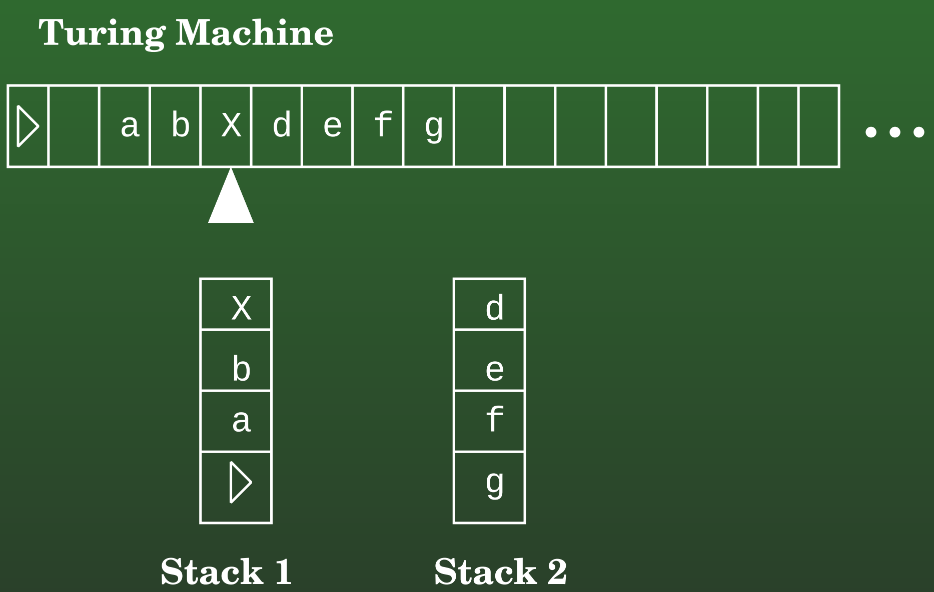
Counter Machines

- To move the tape head to the right
  - Pop symbol off Stack 2
  - Push it on Stack 1



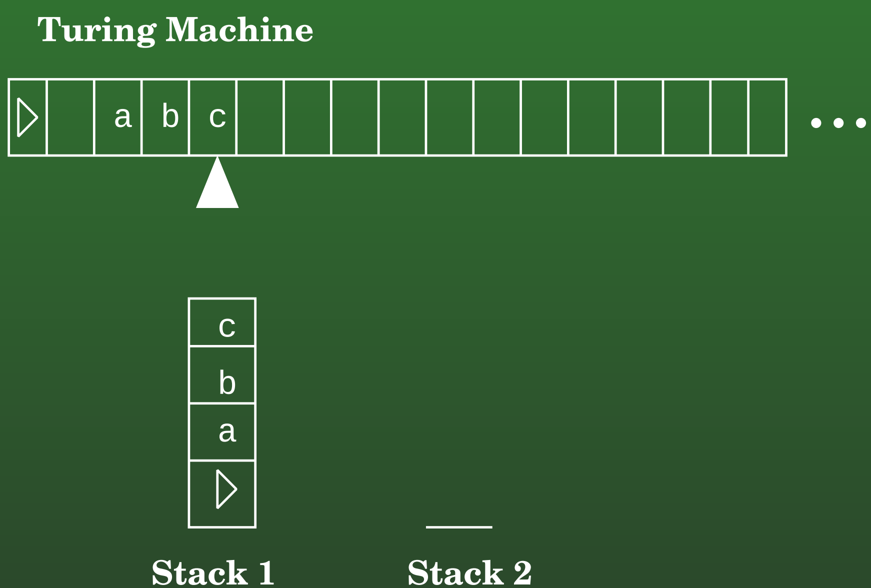
14-54: Counter Machines

- To move the tape head to the right
  - Pop symbol off Stack 2
  - Push it on Stack 1



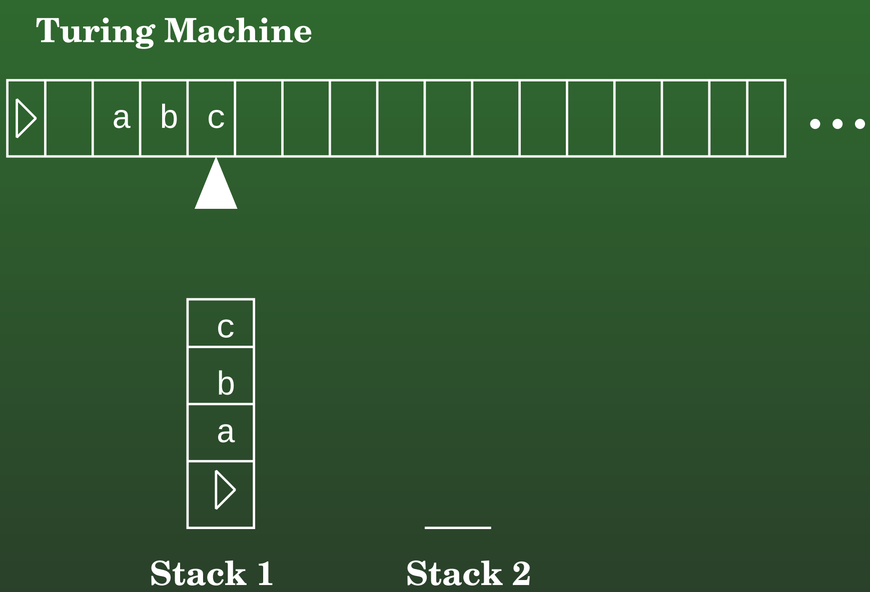
14-55: Counter Machines

- To move the tape head to the right, if Stack 2 is empty ...



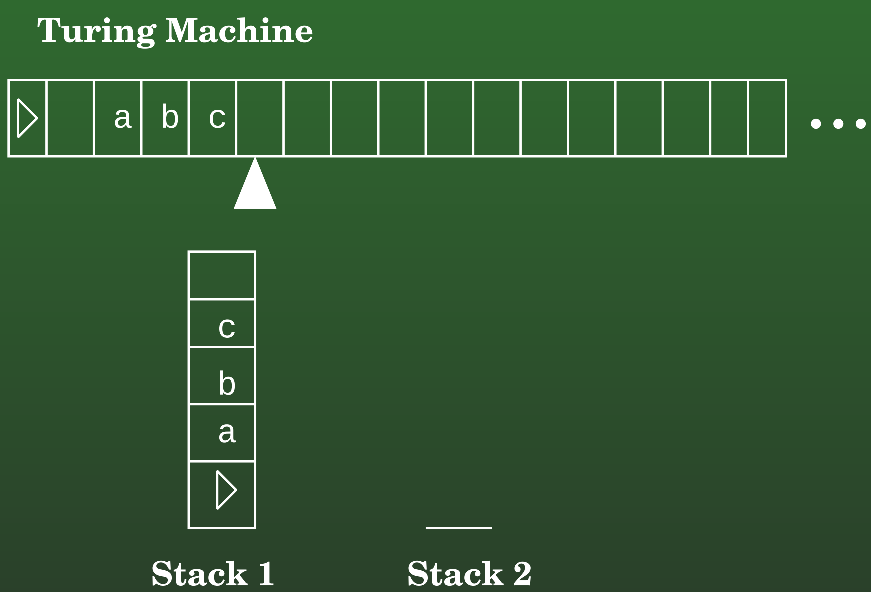
14-56: Counter Machines

- To move the tape head to the right, if Stack 2 is empty ...
  - Push a Blank Symbol on Stack 1



14-57: Counter Machines

- To move the tape head to the right, if Stack 2 is empty ...
  - Push a Blank Symbol on Stack 1



Counter Machines

- Four Counters $\Rightarrow$ Two Stacks $\Rightarrow$ Turing Machine
- If we can simulate a 4-counter machine with a 2-counter machine ...
- Two Counters $\Rightarrow$ Four Counters $\Rightarrow$ Two Stacks $\Rightarrow$ Turing Machine

2 Counter $\Rightarrow$ 4 Counter

- We can represent 4 counters using just one counter
- Counters have values i, j, k, l
- Master Counter: $2^i 3^j 5^k 7^l$
- When all counters have value 0, master counter has value 1

2 Counter $\Rightarrow$ 4 Counter

- Master Counter: $2^i 3^j 5^k 7^l$
  - To increment counter j, multiply Master Counter by 3
    - Decrement Master Counter
    - Increment Scratch Counter 3 times
    - Repeat until Master Counter = 0
    - Move Scratch Counter to Master Counter

14-61: 2 Counter $\Rightarrow$ 4 Counter

- Master Counter: $2^i 3^j 5^k 7^l$
  - To decrement counter j, divide Master Counter by 3
    - Decrement Master Counter 3 times
    - Increment Scratch Counter
    - Repeat until Master Counter = 0
    - Copy Scratch Counter to Master Counter

2 Counter $\Rightarrow$ 4 Counter

- Master Counter: $2^i 3^j 5^k 7^l$
  - To check of counter j is zero, see if MC mod 3 = 0
    - Decrement Master Counter 3 times (if we hit zero in the middle of this operation, MC mod 3 $\neq$ 0, if he hit zero at the end, MC mod 3 = 0)
    - Increment Scratch Counter 3 times
    - Repeat until Master Counter = 0
    - Use Scratch Counter to restore Master Counter

Counter Machines

- Machine with:
  - Finite State Control
  - Two counters
    - Increment, Decrement, check for zero
- Has full power of a Turing machine – can compute anything

14-64: Numerical Functions

- New model of computation: Recursive Functions
  - Very simple functions
  - Method of combining functions
- End up with equivalent power of Turing Machines

Numerical Functions

- Basic Functions:
  - Zero function: $zero_k(n_1, \ldots, n_k) = 0$
  - Identity function: $id_{k,j}(n_1, \ldots, n_k) = n_j$
  - Successor function: $succ(n) = n + 1$ for all $n \in \mathbf{N}$

Numerical Functions

- Zero Function:
  - $zero_3(3, 11, 22) = 0$
  - $zero_2(9, 13) = 0$
  - $zero_0() = 0$

Numerical Functions

- Zero Function:
  - Why have $k-$ary zero function, instead of just defining a the constant 0, or a single $0-$ary function?
    - Notational convenience
    - “Historical Reasons”

14-68: Numerical Functions

- Identity function
  - $id_{1,1}(4) = 4$
  - $id_{4,2}(3, 7, 9, 5) = 7$
  - $id_{5,5}(9, 11, 4, 5, 20) = 20$

14-69: Numerical Functions

- Successor Function
  - $succ(0) = 1$
  - $succ(1) = 2$
  - $succ(2) = 3$
  - $succ(57) = 58$

Numerical Functions

- Combining Functions:
  - Composition
    - $g : \mathbf{N}^k \mapsto \mathbf{N}$ any $k-$ary function
    - $h_1, \ldots h_k$ $l-$ary functions
    - Composition of g with $h_1, \ldots h_k$

$$f(n_1, \ldots n_l) = g(h_1(n_1, \ldots n_l), \ldots h_k(n_1, \ldots, n_l))$$

14-71: Numerical Functions

- Composition:
  - $plus2(x) = succ(succ(x))$
  - $plus3(x) = succ(succ(succ(x)))$

Numerical Functions

- Composition: Constant functions
  - $f() = 5 = succ(succ(succ(succ(succ(zero())))))$
  - $f(3, 2) = 2 = succ(succ(zero()))$

Numerical Functions

- Combining Functions:
  - Recursion
    - $k-$ary function g, $k+2$-ary function h
    - Function f defined recursively by g and h:

$$
\begin{aligned}
f(n_1, \ldots n_k, 0) &= g(n_1, \ldots, n_k) \\
f(n_1, \ldots, n_k, m+1) &= h(n_1, \ldots n_k, m, f(n_1, \ldots, n_k, m))
\end{aligned}
$$

Numerical Functions

- Recursive functions:

$$\begin{aligned} plus(m, 0) &= m \\ plus(m, n+1) &= succ(plus(m, n)) \end{aligned}$$

Numerical Functions

- Recursive functions:

$$\begin{aligned} plus(m, 0) &= m \\ plus(m, n+1) &= succ(plus(m, n)) \end{aligned}$$

  - $g(n) = id_{1,1}(n) = n$
  - $h(n_1, n_2, n_3) = succ(n_3)$

14-76: Numerical Functions

- Recursive functions:

$$
\begin{aligned}
mult(m, 0) &= \\
mult(m, n+1) &=
\end{aligned}
$$

Numerical Functions

- Recursive functions:

$$\begin{aligned} mult(m, 0) &= zero(m) \\ mult(m, n+1) &= plus(m, mult(m, n)) \end{aligned}$$

Numerical Functions

- Recursive functions:

$$\begin{aligned} mult(m, 0) &= zero(m) \\ mult(m, n+1) &= plus(m, mult(m, n)) \end{aligned}$$

  - $g(n) = zero(n)$
  - $h(n_1, n_2, n_3) = plus(n_1, n_3)$

Numerical Functions

- Recursive functions:

$$\begin{aligned} exp(m, 0) &= \\ exp(m, n+1) &= \end{aligned}$$

14-80: Numerical Functions

- Recursive functions:

$$\begin{aligned} exp(m, 0) &= suc(zero(m)) \\ exp(m, n+1) &= mult(m, exp(m, n)) \end{aligned}$$

  - $g(n) = succ(zero(n))$
  - $h(n_1, n_2, n_3) = mult(n_1, n_3)$

14-81: Numerical Functions

- Recursive functions:

$$
\begin{aligned}
fact(0) &= \\
fact(n+1) &=
\end{aligned}
$$

Numerical Functions

- Recursive functions:

$$
\begin{aligned}
fact(0) &= suc(zero()) \\
fact(n+1) &= mult(n+1, fact(n))
\end{aligned}
$$

  - $g(n) = succ(zero(n))$
  - $h(n_1, n_2) = mult(succ(n_1), n_2)$

14-83: Numerical Functions

- Recursive functions:

$$\begin{aligned} pred(0) &= 0 \\ pred(n+1) &= n \end{aligned}$$

  - $g(n) = zero(n)$
  - $h(n_1, n_2) = id_{12}(n_1, n_2) = n_1$

14-84: Numerical Functions

- Recursive functions:

$$\begin{aligned} sub(m, 0) &= m \\ sub(m, n+1) &= pred(sub(m, n)) \end{aligned}$$

What is $sub(3, 5)$? Why?

14-85: Numerical Functions

- Predicate functions
  - $iszero(n) = 1$ if $n = 0$, and 0 otherwise

$$
\begin{aligned}
iszero(0) &= 1 \\
iszero(m+1) &= 0
\end{aligned}
$$

14-86: Numerical Functions

- Predicate functions
  - $geq(m, n) = 1$ if $m \geq n$, and 0 otherwise

$$geq(m, n) =$$

Numerical Functions

- Predicate functions
  - $geq(m, n) = 1$ if $m \geq n$, and 0 otherwise

$$geq(m, n) = iszero(sub(n, m))$$

Numerical Functions

- Predicate functions
  - $lt(m, n) = 1$ if $m < n$, and 0 otherwise

Numerical Functions

- Predicate functions
  - $lt(m, n) = 1$ if $m < n$, and 0 otherwise

$$lt(m, n) = sub(1, geq(m, n))$$

14-90: Numerical Functions

- Predicate functions
  - $and(m, n) = 1$ if $m = 1$ and $n = 1$, and 0 otherwise

Numerical Functions

- Predicate functions
  - $and(m,n) = 1$ if $m = 1$ and $n = 1$, and 0 otherwise

$$and(m,n) = mult(m,n)$$

14-92: Numerical Functions

- Predicate functions
  - $or(m, n) = 1$ if $m = 1$ or $n = 1$, and 0 otherwise

14-93: Numerical Functions

- Predicate functions
  - $or(m, n) = 1$ if $m = 1$ or $n = 1$, and 0 otherwise

$$or(m, n) = sub(1, iszero(plus(m, n)))$$

Numerical Functions

- Defining functions by cases:

$$f(n_1, \ldots, n_k) = \begin{cases} g(n_1, \ldots, n_k) & \text{if } p(n_1, \ldots, n_k) \\ h(n_1, \ldots, n_k) & \text{otherwise} \end{cases}$$

Numerical Functions

- Defining functions by cases:

$$
\begin{aligned}
rem(0, n) &= 0 \\
rem(m+1, n) &= \begin{cases} 0 & \text{if } equal(rem(m, n), pred(n)) \\ rem(m, n) + 1 & \text{otherwise} \end{cases}
\end{aligned}
$$

(Using first parameter of function as recursion control)

Numerical Functions

- Defining functions by cases:

$$\begin{aligned} div(0, n) &= 0 \\ div(m+1, n) &= \begin{cases} div(m, n) + 1 & \text{if } equal(rem(m, n), pred(n)) \\ div(m, n) & \text{otherwise} \end{cases} \end{aligned}$$

(Using first parameter of function as recursion control)

14-97: Numerical Functions

- Defining functions by cases:

$$f(n_1, n_2, \ldots, n_k) = \begin{cases} g(n_1, n_2, \ldots, n_k) & \text{if } P(n_1, n_2, \ldots, n_k) \\ h(n_1, n_2, \ldots, n_k) & \text{otherwise} \end{cases}$$

How can we get “functions by cases” using the tools we already have?

Numerical Functions

- Defining functions by cases:

$$f(n_1, n_2, \ldots, n_k) = \begin{cases} g(n_1, n_2, \ldots, n_k) & \text{if } P(n_1, n_2, \ldots, n_k) \\ h(n_1, n_2, \ldots, n_k) & \text{otherwise} \end{cases}$$

$$\begin{aligned} f(n_1, n_2, \ldots, n_k) &= P(n_1, n_2, \ldots, n_k) * g(n_1, n_2, \ldots, n_k) \\ &\quad +((1 - P(n_1, n_2, \ldots, n_k)) * \\ &\quad\; h(n_1, n_2, \ldots, n_k)) \end{aligned}$$

Numerical Functions

- Are there any functions which we can compute, that *cannot* be computed with primitive recursive functions?

Numerical Functions

- Are there any functions which we can compute, that *cannot* be computed with primitive recursive functions?
  - Yes!
  - Use a diagonalization argument
- To make life easier, we will only consider functions that take a single argument (unary functions)

Numerical Functions

- Unary Primitive Recursive Functions can be enumerated
  - That is, we can define an order over all unary primitive recursive functions, $f_1(n), f_2(n), f_3(n), \ldots$
  - How can we order them?

14-102: **Numerical Functions**

- Enumerating Unary Primitive Recursive Functions
  - Each function is created by combining basic functions (succ, zero, select, etc) using composition and recursion
  - Can describe any function using a string
  - Order the strings in lexographic order (shortest to longest, using standard string compare for strings of the same length)

Numerical Functions

- Let the unary primitive recursive functions be: $f_0, f_1, f_2, f_3, \ldots$
- Define a new function $g(n) = f_n(n) + 1$
  - We can compute $g(n)$ by first finding the nth unary recursive function f_n, computing $f_n(n)$, and adding 1 to the result

14-104: Numerical Functions

- Let the unary primitive recursive functions be: $f_0, f_1, f_2, f_3, \ldots$
- Define a new function $g(n) = f_n(n) + 1$
  - We can compute $g(n)$ by first finding the nth unary recursive function f_n, computing $f_n(n)$, and adding 1 to the result
- $g(n)$ can be computed (we just showed how)
- $g(n)$ cannot be computed by a primitive recursive function! (why not?)

14-105: Numerical Functions

- $g(n)$ can be computed (we just showed how)
- $g(n)$ cannot be computed by a primitive recursive function! (why not?)
  - Not computed by the 0th primitive recursive function
  - Not computed by the 1st primitive recursive function
  - Not computed by the 2nd primitive recursive function
  - $\ldots$

14-106: Numerical Functions

- There are some well defined functions, which we can compute, which cannot be computed by primitive recursive functions.
- Can we add anything to primitive recursive functions to give them more power, so that any well defined function that can be computed can be computed with recursive functions?

Numerical Functions

- Minimization
  - If g is a $(k+1)$-ary function. The minimialization of g is the k-ary function f defined as:

$$f(n_1, \ldots, n_k) = \begin{cases} \text{The least } m \text{ such that} \\ \quad g(n_1, \ldots, n_k, m) = 1, \\ \quad \text{if such an } m \text{ exists} \\ 0 \text{ otherwise} \end{cases}$$

Minimization of g is denoted $\mu m[g(n_1, \ldots n_k, m) = 1]$

Numerical Functions

- Minimization Examples

$$div(x, y) = \mu z[(y * (z + 1)) - x > 0]$$

"$-$" is "positive subtraction" (that is, if $y > x$, then $x - y = 0$)

$$\begin{aligned} div(x, y) &= z \\ y * z &\leq x \\ y * (z + 1) &> x \end{aligned}$$

14-109: Numerical Functions

- Minimization Examples

$$log(m,n) = \mu p[power(m,p) \geq n]$$

"$\geq$" is the "greater-than-or-equal" predicate

Numerical Functions

- Calculating minimalization:

$m \leftarrow 0$;
while ($g(n_1, \ldots, n_k, m) \neq 1$)
    $m \leftarrow m + 1$
return m

Numerical Functions

- Calculating minimalization:

$m \leftarrow 0$;
while ($g(n_1, \ldots, n_k, m) \neq 1$)
    $m \leftarrow m + 1$
return m

... of course, this may never terminate, if there is no value of m such that $g(n_1, \ldots, n_k, m) = 1$

Numerical Functions

- A function $g(n_1, \ldots, n_k, m)$ is *minimalizable* if
  - For each $n_1, \ldots, n_k \in \mathbf{N}$, there exists some m such that $g(n_1, \ldots, n_k, m) = 1$

That is:

$m \leftarrow 0$;
while ($g(n_1, \ldots, n_k, m) \neq 1$)
    $m \leftarrow m + 1$
return m

always terminates, for all values $n_1, \ldots, n_k$

14-113: Numerical Functions

- μ-Recursive
  - A function is μ-recursive if it consists entirely of primitive-recursive functions, and minimalizations of minimalizable functions.
  - μ-recursive functions can calculate anything that can be decided by a Turing machine
    - (recall that “decide” means the TM halts on all inputs)

14-114: Numerical Functions

- μ-recursive functions can calculate anything that can be decided by a Turing machine
  - We can enumerate μ-recursive functions just like we enumerated primitive recursive functions $f_0, f_1, f_2, \ldots$
  - We can define the function $g(n) = f_n(n) + 1$
  - How can I assert that μ-recursive functions can compute anything that a Turing Machine can compute, when μ-recursive functions can't compute g?

Numerical Functions

- Method to compute $g(n)$ using a Turing machine:
  - Enumerate first $n+1$ functions f
    - $f_0, f_1, \ldots, f_n$
  - Compute $f_n(n)$
  - Output $f_n(n)+1$

14-116: Numerical Functions

- Method to compute $g(n)$ using a Turing machine:
  - Enumerate first $n+1$ functions f
    - $f_0, f_1, \ldots, f_n$
  - Compute $f_n(n)$
  - Output $f_n(n) + 1$
- Function f_n might not be minimalizable! If $f_n(n)$ is not minimalizable, then $f_n(n) = 0$, but we have no way of discovering this!

14-117: Recursive Languages

- μ-recursive functions can calculate anything that can be decided by a Turing machine.
- $\{L : L$ is decided by some TM $M\}$ is the recursive languages
- How can a function from the natural numbers to the natural numbers decide a language?

14-118: Recursive Languages

- How can a function from the natural numbers to the natural numbers decide a language?
  - Any string can be encoded as a number
    - ASCII-style encoding scheme to encode each symbol in string
    - Append codes of each symbol together to get a (really large) number

$\Sigma = \{a, \ldots, z\}$, $en(a) = 10$, $en(b) = 11$, $\ldots$,
$en(z) = 35$
$en(abbz) = 10111135$

14-119: Recursive Languages

- How can a function from the natural numbers to the natural numbers decide a language?
  - Any string can be encoded as a number
  - Predicate function can be used to determine membership
    $L[f] = \{w : f(en(w)) = 1\}$

μ-Recursive Functions & TMs

- Any μ-recursive function can be decided by a Turing machine
  - Each of the primitive-recursive functions can easily be simulated by a Turing machine
  - Any minimalizable function can be computed by a Turing machine that tries all values in order

$m \leftarrow 0$;
while $(g(n_1, \ldots, n_k, m) \neq 1)$
    $m \leftarrow m + 1$
return m

 # μ-Recursive Functions & TMs

- Any function that can be decided by a Turing machine can be computed with a μ-recursive function
  - We can encode a configuration as a number
  - We can encode a sequence of configurations with a (much larger) number

  config$_1$config$_2$config$_3 \ldots$.config$_n$
- Each configuration encodes tape contents, head location, and current state of the Turing Machine

14-122: μ-Recursive Functions & TMs

- We have a large number, which represents a series of configurations for a Turning Machine config$_1$config$_2$config$_3$ . . .config$_n$
- We can write a primitive-recursive predicate function $isvalid$ that examines this string of configurations, and determines if it is legal
  - if config$_i$config$_j$ appears in the sequence
  - Turing machine will move from config$_i$ to config$_j$ in a single step

14-123: μ-Recursive Functions & TMs

- $isvalid(n)$
  - Predicate function
  - True if n is a number which represents a valid sequence of configurations of a Turing Machine
  - Writing $isvalid$ for a particular Turing Machine is reasonably straightforward
    - Extract 1st and 2nd configurations from the number (using div and mod)
    - Make sure that the transition from 1st to 2nd configuration is valid
    - Recursively check the rest of the transitions

14-124: μ-Recursive Functions & TMs

- Given a number which represents a valid sequence of configurations for the Turning Machine M, if:
  - If the first configuration represents the initial state and the input n
  - The last configuration contains a halting state h
  - The tape contents of the last configuration represents the value y
- Then the Turing Machine M gives the output y for the input n

14-125: μ-Recursive Functions & TMs

- Given a number which represents a sequence of configurations, and an input n, we can:
  - Determine if the sequence of configurations is valid
  - Ensure that the first configuration encodes n
  - Ensure that the last configuration contains a halting state

14-126: μ-Recursive Functions & TMs

- Function $check\_compute(n, x)$
  - Takes as input a string of configurations n, and an initial configuration x
  - Returns 1 (true) if n is a valid series of computations that starts with x
    - $isvalid(n) = 1$
    - $first(n) = x$

14-127: μ-Recursive Functions & TMs

- Function $compute(x)$
  - Calculates and returns the string of valid configurations that starts with x and ends in a halting state

μ-Recursive Functions & TMs

- Function $compute(x)$
  - Calculates and returns the string of valid configurations that starts with x and ends in a halting state

$$compute(x) = \mu n[check\_compute(n, x)]$$

μ-Recursive Functions & TMs

- Function f_M, that calculates the same function as the Turing Machine M:

$$f_M(x) = last(compute(x))$$