if $n>0$, $x>0$ replace
$s'nx$ by $s(n-1)n(x-1)$

Claim on a given input $S=nx$ this algorithm terminates w/ $A(n,x)$

pf: (HW)

Register machines

- We introduce a model of computation using register machines.
- go back to working on an alphabet $A=\{\sigma_1,\dots\sigma_k\}$ w/ k-many symbols

Def'n a register machine on A is an infinite list of registers $R1, R2, R3, \dots$, each capable of storing an arbitrary word $w\in A^*$.

- RMs perform computations by responding to instructions of the following types.

Instruction	Meaning
1. ADD_j R_n	add σ_j to the right of word stored in R_n
2. DEL R_n	delete leftmost symbol of word in R_n

3. CLR R_n — erase word in R_n

4. $R_m \leftarrow R_n$ — copy word in R_n over word in R_m (w/o deleting word in R_n)

5. GOTO ℓ — go to ℓth instruction

6. IF FIRST$_j$ R_n, GOTO ℓ — if first symbol in word in R_n is σ_j, go to ℓth instruction, o/w go next instruction

7. STOP — halt computation

Def'n: a program P is a finite numbered list of instructions $I_1, I_2, \ldots, I_n$ s.t.:

- if I_m is a GOTO ℓ or IF FIRST$_j$, GOTO ℓ then $1 \leq \ell \leq n$
- I_n = STOP and no other instructions are STOP

- an input is a finite sequence of A-words $(w_1, \ldots, w_m)$ occupying first m registers
- on such an input program proceeds sequentially (after I_k, perform I_{k+1}) except for GOTOs
- computation halts on STOP
- if at STOP have $v_1, v_2, \ldots$ in $R1, R2, \ldots$ then $v_1, v_2, \ldots$ is output.

- a program P may or may not halt on a given input
- if halts: output is a finite list of words
- even if P doesn't halt, since instructions in P only refer to finitely many registers, registers R_q always empty for q larger than some cutoff point N.

Ex (a program that never halts)

I_1: ADD, R1
I_2: GOTO 1
I_3: STOP

(notice: satisfies conditions for being a program)

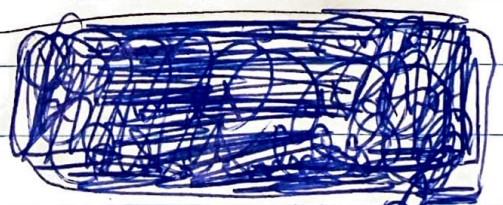
- on input $(w_1, \ldots, w_m)$ program adds σ_1's repeatedly at right of first word:

$$(w_1, \ldots, w_m) \to (w_1\sigma_1, \ldots, w_m) \to (w_1\sigma_1\sigma_1, \ldots, w_m) \to \ldots$$

- Sps we have a program P and $n \geq 1$
- P determines a partial function

$$\varphi_P^n : (A^*)^n \to A^*$$

↑ arity of φ.



defined by: given $\vec{x} = (x_1, \ldots, x_n) \in (A^*)^n$ have $\varphi(\vec{x})\downarrow$ iff P on input $\vec{x}$ halts

and in this case
if $v_1, v_2, \ldots$ is output of P on $\vec{x}$
then $\varphi_P^n(\vec{x}) = v_1$,
(i.e. word in R1 is output of Rmchn)

Defn a partial function $f: (A^*)^n \to A^*$ is **RM-computable** if there is a finite alphabet $B \supseteq A$ and an RM program P s.t. $\varphi_P^n \restriction (A^*)^n = f$.

(note: almost always just have $B = A$, but sometimes convenient to have access to extra symbols)

Theorem Every (partial) recursive f on A is RM-computable (... and can take $B = A$ in proof.)

Pf by induction on the construction of recursive functions.

1. Erase function $E(x) = 0$ is computable by following program P:
   1. CLR R1
   2. STOP
   (no matter input $(x_1, \ldots, x_n)$ P erases first word and stops — output is $(0, x_2, \ldots, x_n, 0, 0, \ldots)$
   (so: 1-ary function φ_P^1 computed by P is E)

successor

2. $S_j(x) = x\sigma$ is computable by:
   1. ADD j R1
   2. STOP

projections

3. $P_j^n(x_1, \ldots, x_n) = x_j$ is comp'l by:
   1. R1 ← R_j
   2. STOP

4. (composition) $f: (A^*)^n \to A^*$ $\vec{x} = (x_1, \ldots, x_n)$

- Sps $f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$ and we have programs $Q, P_1, \ldots, P_m$ computing $g, h_1, \ldots, h_m$.
- we find program that computes f
- For simplicity assume $n=2$, so $f(\vec{x}) = g(h_1(\vec{x}), h_2(\vec{x}))$, Q computes g, P_1, P_2 compute h_1, h_2

- pick $q >$ all register indices mentioned in Q, P_1, P_2 (so these programs don't interact w/ registers R_i for $i > q$)
- Following program computes f:

$R(q+1) \leftarrow R1$
$\vdots$
$R(q+n) \leftarrow Rn$
} "store the x_i's"

$R1 \leftarrow R(q+1)$
$\vdots$
$Rn \leftarrow R(q+n)$
} "copy x_i's to front registers" (unnecessary @ first stage, but including for uniformity)

CLR R(n+1)
⋮
CLR Rq
} remove all other info except x_i's in registers seen by Q, P_1, P_2 (again, unnecessary @ first stage)

" " explained below

" P_1 "
$R(q+n+1)$
} compute $h_1(\vec{x})$ and store in $R(q+n+1)$

"copy x_i's to front"
"clear all other registers up to R_q"

" P_2 "
$R(q+n+2) \Leftarrow R1$
} compute $h_2(\vec{x})$ and store in $R(q+n+2)$

$R1 \Leftarrow R(q+n+1)$
$R2 \Leftarrow R(q+n+2)$
} copy $h_1(\vec{x})$, $h_2(\vec{x})$ to front registers

"clear all other registers up to R_q"

"Q" compute $g(h_1(\vec{x}), h_2(\vec{x}))$

↪ Here "P_1", "P_2", "Q" indicate we have suitably modified original programs removing STOP commands (except in Q) and re-indexing commands appropriately. ✓

5. (primitive recursion)

- Sps $f(0, \vec{x}) = g(\vec{x})$
and $f(y\sigma_j, \vec{x}) = h_j(f(y, \vec{x}), y, \vec{x})$
and we have programs $Q, P_1, \ldots, P_K$ computing $g, h_1, \ldots, h_K$

- Fix q large enough so programs don't interact w/ registers R_i for $i > q$.
- We informally describe a program computing f.

- Sps $y, x_1, \ldots, x_n$ is input where $y = \sigma_{i_1} \cdots \sigma_{i_m}$

"store $y, \vec{x}$ in $R(q+1), \ldots, R(q+n+1)$"
"compute $g(\vec{x})$ by copying $\vec{x}$ to front registers, clearing other registers up to R_q and running Q"
"store $g(\vec{x})$ in $R(q+n+2)$"

- then do following:

"IF $R(q+1) = 0$, copy $g(\vec{x})$ to $R1$ and STOP"

(really: IF $\text{FIRST}_{q+1} = 0$ GOTO ℓ where I_ℓ is instruction copying $R(q+n+2)$ to $R1$ followed by GOTO a STOP)

"Else, if $R(q+1)$ starts w/ σ_j, delete σ_j, copy $R(q+n+2)$ to $R1$, copy $R(q+n+3)$ to $R2$, copy x_i's to ~~[illegible]~~ $R3, R4, \ldots, R(n+2)$"

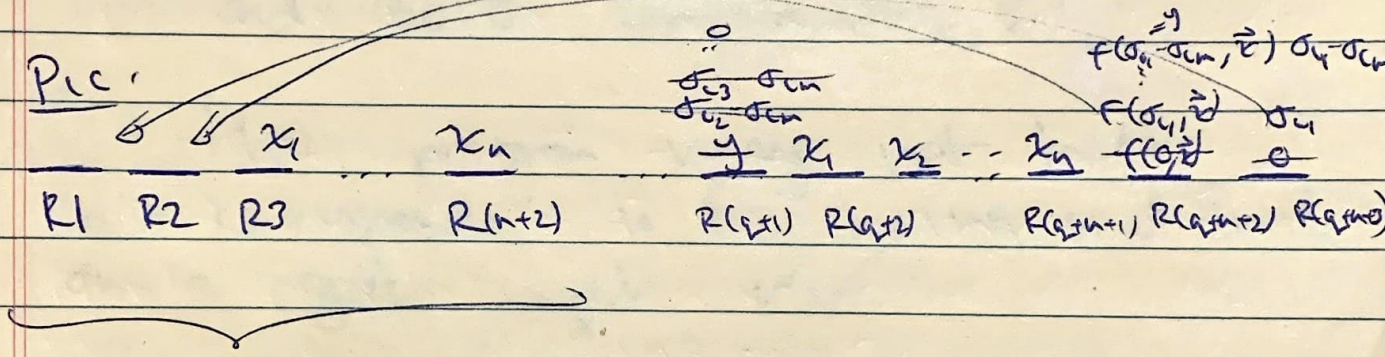
"Then compute $h_j(R1, R2, x_1, \ldots, x_n)$ (by running P_j) and store in $R(q+n+2)$"

"Add σ_j to right of word in $R(q+n+3)$"

Idea: - $R(q+n+2)$ successively holds the values $f(0,\vec{x}) = g(\vec{x})$, $f(\sigma_{i_1},\vec{x})$, $f(\sigma_{i_1}\sigma_{i_2},\vec{x})$, ...
$f(\sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_m},\vec{x}) = f(y,\vec{x})$

- $R(q+n+3)$ successively holds
$0, \sigma_{i_1}, \sigma_{i_1}\sigma_{i_2}, \ldots, \sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_m} = y$

Pic:



at a given stage k compute
$h_{i_k}(f(\sigma_{i_1}\cdots\sigma_{i_{k-1}},\sigma_{i_1}\cdots\sigma_{i_k},\vec{x})$

continue until $R(q+1)$ is empty
output is $f(y,\vec{x})$ ✓

(exercise: try to spell it formally)

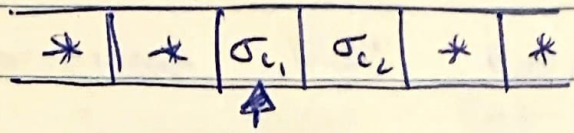
6. (minimization)
- Sps $f(\vec{x}) = \mu y [g(y, \vec{x}) = 0]$ and P is a program computing g
- Let q be large enough

- Following program computes f:
  "store $x_1, \ldots, x_n$ in $R(q+1), \ldots, R(q+n)$"
  "compute $g(R_q, \vec{x})$" (R_q is empty to start)
  using P
  "if $g(R_q, \vec{x}) = 0$, copy R_q to $R1$ and STOP"
  "else add σ_j to word in R_q and GOTO compute $(R_q, \vec{x})$"

↳ this program may not halt but corresponds to f being undefined on a given input ✓✓

Turing machines

- we describe another model of computation
- Sps $A = \{\sigma_1, \ldots, \sigma_k\}$ is our alphabet
- a Turing machine (TM) on A consists of a bi-infinite tape divided into squares



a TM

- each square can hold at most one symbol from A
- machine has a read/write head that scans current square, can print or erase a symbol on that square, and can move left or right to an adjacent square.

- a program for a TM is a finite ordered list of instructions

Instruction	Meaning
- PRINT$_j$ (for $1 \le j \le k$)	- print σ_j in current square
- DEL	- erase current square
- RIGHT	- move head one square right
- LEFT	- " " left
- GOTO l	- goto lth instruction in program
- IF$_j$, GOTO l	- if current square contains σ_j, goto lth instruction
- STOP	- halt computation