CSE396 Spr26 Lecture Thu. Apr. 2: Computations, Simulations, and Problem-Solving

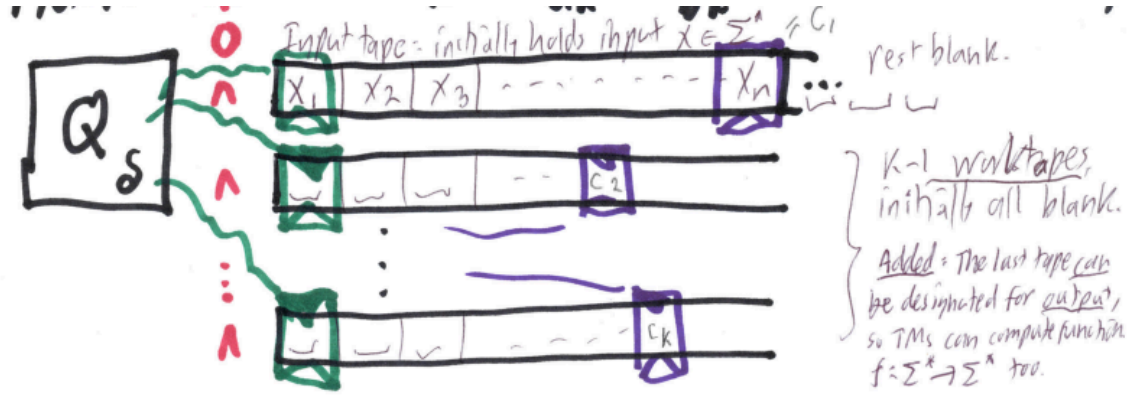
An **instantaneous description** (**ID**), also called a **configuration**, of a Turing machine M specifies:

1. The current internal state q of M.
2. The contents $\vec{w} = w_1, \ldots, w_k$ of the k tapes, such that all else on the tapes is blank.
3. The positions $\vec{h} = h_1, \ldots, h_k$ of the heads on those tapes.

We can write $I = \langle q, \vec{w}, \vec{h} \rangle$ to denote an ID.

Write $I \vdash_M J$ if there is an instruction in δ that when executed in ID I produces ID J. For $r \geq 2$, write $I \vdash_M^r K$ if there is an ID J such that $I \vdash_M J$ and $J \vdash_M^{r-1} K$. Also write $I \vdash_M^0 I$ for all I and $I \vdash_M^* J$ if $I \vdash_M^r J$ for some r. These notions apply to nondeterministic TMs as well as DTMs. (This reminisces the "derives" notation for grammars. The "turnstile" symbol is also used in logic to mean "proves"---and this is actually the more pregnant analogy.)

For a single-tape TM and input x, the initial ID can be written $I_0(x) = \langle s, x, 1 \rangle$ (if we number the cells from 1) or $I_0(x) = \langle s, \wedge x, 1 \rangle$ (if we use the convention of an initial $\wedge$ in cell 0 but still number x from 1 and start up scanning the first bit rather than the $\wedge$). Yet another convention is to start in the ID $\langle s, \wedge x \$, 1 \rangle$ with a right-endmarker $\$$ too. The symbols $\wedge$ and $\$$ mean "beginning of line" and "end of line" in various text-processing notations. The $\wedge$ also suggests the bottom of a stack when used with a PDA. We can use it as a left-endmarker for general multi-tape TMs too:



The initial ID $I_0(x)$ has all heads in column 1 with the first tape head scanning the first bit X_1 of x (or scanning a blank if $x = \varepsilon$) and all other tapes are blank. Defn of $I \vdash_M J$, $I \vdash_M^* K$, accepting ID, and $L(M)$: same

Alternate convention: Γ includes a dedicated $\wedge$ char. Each tape has $\wedge$ in cell 0. Code in δ always moves R off $\wedge$. Never changes $\wedge$, never moves L off $\wedge$.

For PDAs, $\wedge$ acts as a bottom of stack marker.

We won't go into gritty detail for defining IDs $I, J, K, \ldots$ of multi-tape TMs. It's enough that the definitions of $I \vdash J$ and $I \vdash^* K$ are the same, once the IDs themselves are defined.

An **accepting ID** has q_{acc} as its state and a *rejecting ID* has q_{rej}. Now we can formally define the language of a TM (NTMs too):

Definition 1: $L(M) = \{x : I_0(x) \vdash^*_M I_f \text{ for some accepting ID } I_f\}$.

Given a **D**TM M and an input x, we can write "$M(x)$" to refer to the whole computation that occurs. It might never halt, in which case we write $M(x) \uparrow$ and also say $M(x)$ **diverges**. If it halts, we write $M(x) \downarrow$. If $M(x)$ halts for all inputs x, we say M is **total**.

Definition 2:

- A language A is **computably enumerable** (**c.e.**) if there is a DTM M such that $L(M) = A$.
- A is **decidable** if there is a DTM M such that $L(M) = A$ and M is total.

In the latter case, we say that M **decides** A and call M itself a **decider**. An older synonym for decidable is **recursive**. An older synonym for c.e. is **recursively enumerable** with abbreviation **r.e.** The older terms survive in the standard names of the classes of languages involved:

Definition 3: Over any fixed alphabet Σ,

- the class of c.e. languages is denoted by **RE**.
- the class of decidable languages is denoted by **REC**.
- the class of complements of c.e. languages is denoted by **co-RE**.

More synonyms: c.e. languages are also called **Turing-acceptable** and "partially decidable." The term "recognizable" can be vague on whether it means decidable or just c.e.

Example of a 2-tape TM that decides a non-CFL, the double-word language.

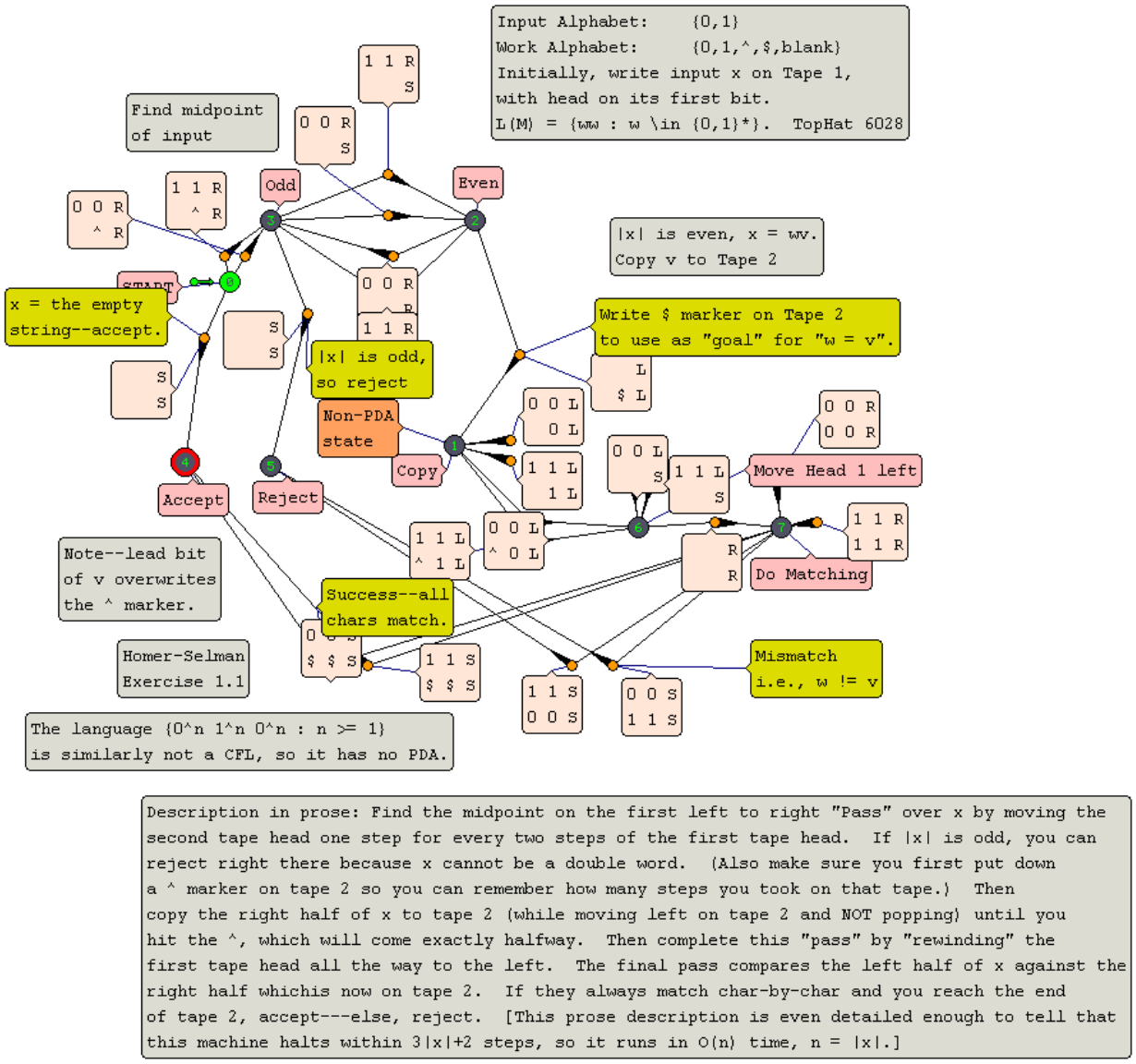


Figure 1: The description in prose at the bottom will **later** be regarded as enough to specify the particular state code with arcs and nodes. The equivalence of TMs to high-level languages (end of chapter 3 and beginning of chapter 4) justifies this theoretically.

We can also define computability for functions $f : \Sigma^* \to \Sigma^*$. We can also identify the natural numbers $\mathbb{N}$ with binary strings (ignoring leading 0s; if you want a bijective encoding there are ways to do that too) and thus include computing numerical functions. Here is an important technote: When one writes $f : A \to B$, one means that the **domain** of f is all of A, but the "actual range" of f need not be all of B. Thus writing $f : \mathbb{N} \to \mathbb{N}$ means that f is **total**, hence that a Turing machine T such that $T(x) = f(x)$ for all x must also be total. But we can also imagine computing partial functions---or even functions g where we don't know if they're only partial, such as

$g(n)$ = the number of steps n needs to go down to 1 in the "Collatz $3n + 1$ Game",

for which a Turing machine computing $g(n)$ was shown in the opening week of the course. To allow for this, we word definitions like so:

Definition 4: A function f is **computable** if there is a Turing machine T such that for all x in the domain of f, T on input x halts in the ID $\langle y, q_{acc}, 1\rangle$ where $y = f(x)$. If the domain of f is all of Σ^* (or all of $\mathbb{N}$ for a numerical function, or all of $\mathbb{Z}$ allowing negative numbers, etc.), so that T is total, then f is **total computable**, also called **total recursive**. Otherwise, f is **partial computable**, also called **partial recursive**.

Proposition 2: A language L is decidable if and only if the Boolean function $L(x)$ is (total) computable. L is c.e. if and only if the partial function

$$\pi_L(x) = \begin{cases} 1 & if\ x \in L \\ undefined & if\ x \notin L \end{cases}$$

is partial computable. ☒

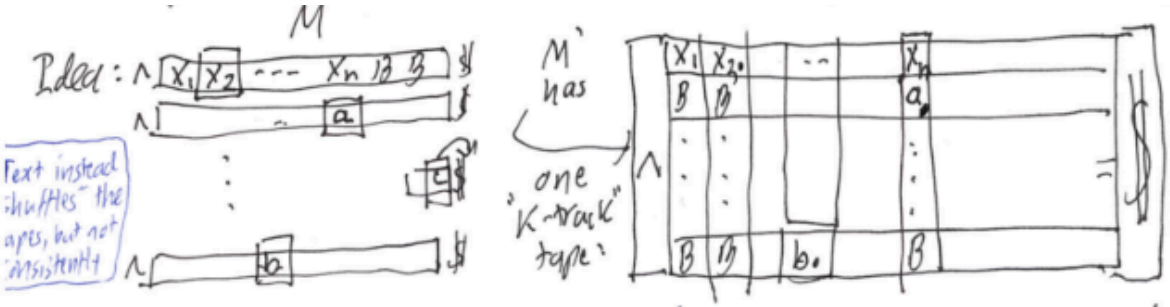
Time Complexity and Simulations

Definition 5: A Turing machine M **runs in time $t(n)$** if for all n and inputs x of length n, $M(x)$ halts within $t(n)$ steps. If M is nondeterministic, all possible computations must halt within $t(n)$ steps.

For example, every DFA---and every NFA without ϵ-transitions---runs in time $t(n) = n + 1$, which is the fastest possible time that reads every input char and the blank that says the input is terminated. (This is sometimes called running in **real time**.) It is convenient to apply O-notation to time without caring about the exact number of steps. All the 2-tape machines we have seen have run in $O(n)$ time, which is called **linear time**, but some of the 1-tape machines have run in $\Theta(n^2)$ time, which is **quadratic time**. This is no accident:

Theorem 3: For any k-tape TM $M = (Q, \Sigma, \Gamma, \delta, \sqcup, s, F)$ that runs in time $t(n)$, we can build a 1-tape TM M' that simulates M and runs in $O(t(n)^2)$ time.

Proof Sketch: M' uses work alphabet $\Gamma' = \Gamma^k$, which can pack the k chars in any "column j" of the k tapes of M into one "superchar" in cell j on the one tape of M'. We also need chars that say whether they are currently being scanned by a tape head of M, so we actually have $\Gamma' = (\Gamma \cup \Gamma_{\odot})^k$ where $\Gamma_{\odot}$ is a "dotted copy" of Γ. Initially, $M'(x)$ converts each char x_i into the "superchar" $[x_i\sqcup\sqcup\sqcup \cdots \sqcup]$ which packs x_i and $k-1$ blanks into one char of Γ and rewinds its single tape head onto the superchar $[\wedge_{\odot}\sqcup_{\odot}\sqcup_{\odot} \cdots \sqcup_{\odot}]$ which lines up the k "virtual" tape heads of M on $\wedge$ and blanks below it to the left of x. Here are snapshots of the layout and idea:



Use • as an extra char marker for the current location of each head

Hence $\Gamma' = \Gamma \cup (\Gamma \cup \Gamma_\bullet)^K$ where $\Gamma_\bullet$ is a dotted copy of Γ.
(includes B and ⊔)

In this manner, even ID I of M is "replicated" by an ID I' of M'

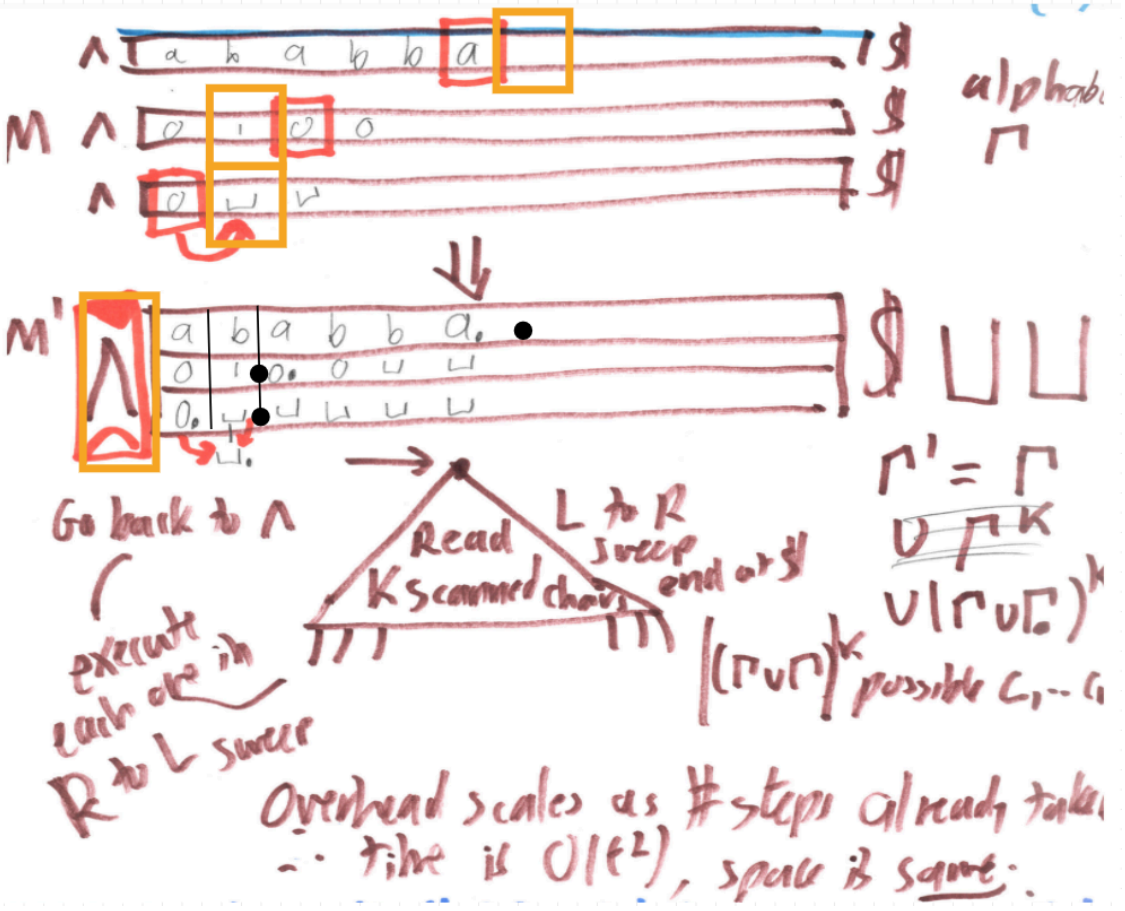
If $I \vdash_M J$, then M' can build the corresponding J' from I', but it usually will require one L-to-R-to-L pass between ^ and $.

Thus M' simulates M. ☒

(i.e. M' is operationally equivalent to M.)

initially $s = n$, always $s \le t$

$s \le \max\{t, n\}$



Thereafter, M' simulates each step of M in one left-to-right **pass** that reads the k-tuple of scanned characters according to which parts of superchars have $\odot$ and then a right-to-left pass that performs the corresponding instruction of δ.

The total time for each pass is initially $2n + 4$ but can grow if and when M uses more tape cells beyond the end(s) of x. The width of a pass cannot be more than (twice the) time taken by M thus far, so it is always less than $t(n)$ (or less than $2t(n)$, if M uses cells to the left of x as well). Thus the total time is $O\big(t(n)^2\big)$. $\boxtimes$

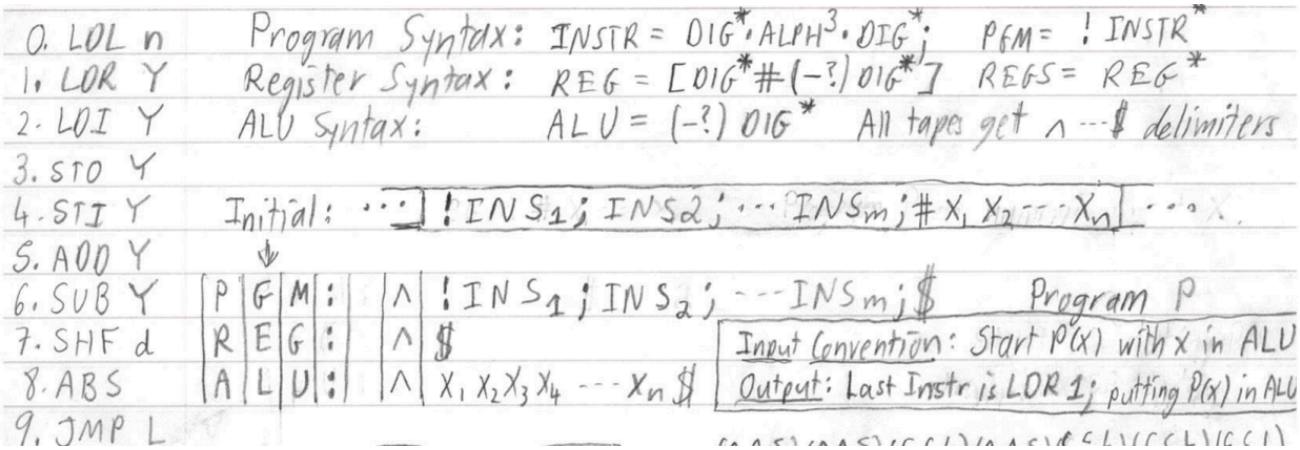
If we forbid a TM M, whether 1-tape or k-tape, from altering or going beyond the $\wedge$ and $\$$ endmarkers, then M is a **linear bounded automaton** (**LBA**). An LBA is allowed to have an arbitrarily large work alphabet Γ, but given an input of length n, it must fit all storage in the n alotted tape cells. Whereas "TM" defaults to DTM, "LBA" defaults to nondeterministic: **NLBA**. Just FYI at this point: NLBAs recognize the same languages as so-called **context-sensitive** grammars (**CSG**s). We will revisit them at the end when we talk about and formalize **space complexity**.

Universal Computation

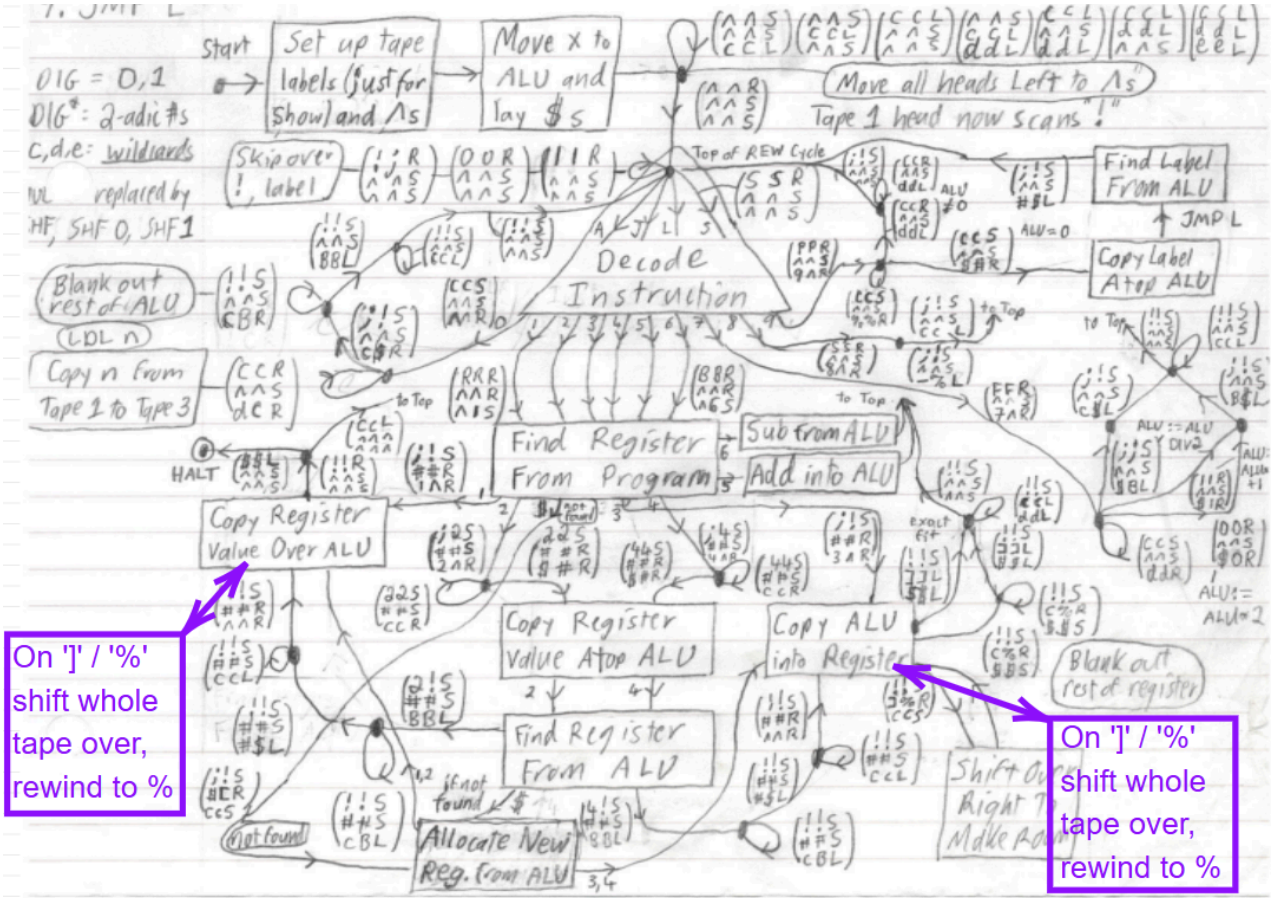
The TMs we have seen show off some basic capabilities of Turing Machines

- copying a string
- comparing two strings
- searching for a matching (sub-)string on a tape
- arithmetic like $3n + 1$
- branching according to what char is read
- looping.

These operations suffice to build an interpreter for assembly code. Here is a handwritten sketch of one that is actually richer than it needs to be by allowing indirect store and retrieve. The ten command names all have three letters, like the keys of a 1970s-vintage Texas Instruments programmable calculator, but instead of using "Reverse Polish Notation" (like a stack-based interpreter) it has arguments to the right like in more conventional assemblers. The main differences from the latter are not having *type-specific* instructions and using only one argument at a time, with the current contents of a special "ALU tape" being the implicit other argument. The input tape and three worktapes of the Turing machine T are structured as follows:



And here is a big chunk of the code, with the basic *copy*, *compare*, and *search* routines put in square blocks. The Turing Kit actually implements a "wildcard" feature used with c, d, and e in the code. Definite instructions take precedence over ones with wildcards, and constrained ones where, say, the first two tapes both have c to mean the case where characters match, take precedence over ones with c then d allowing the chars processed to be independent. (The tiebreak rules with three or more tapes get tricky.) The basic architecture is a single "**Read-Evaluate-Write**" (**REW**) loop like with the k-tapes-to-1 simulation:



Theorem 4: For every program P written in any known executable programming language $\mathscr{L}$ (high-level or otherwise) that uses standard input and standard output, we can build a 3-tape Turing machine M_P such that whenever P given x on standard input writes y to standard output, M_P given x on its input tape writes y to a special output tape. If $P(x)$ halts, then $M_P(x)$ halts.

Proof: First, any compiler for $\mathscr{L}$ to a known code target can be converted into a compiler from $\mathscr{L}$ to the "mini-assembler"---which is essentially similar to what the text calls a RAM. So we can compile P to make an equivalent RAM program R_P. Then take M_P to be the Turing machine T in the handout, but with the binary text of R_P already written on its input tape. More precisely, M_P begins with a series of dedicated instructions that write out R_P char-by-char in front of any input x on its first tape, so it has $R_P\#x$ there. Then it just segues to the start state of T. $\boxtimes$

Theorem 5: We can build a **universal Turing machine**, meaning a single TM U that takes inputs of the form $\langle M, x\rangle$ and simulates $M(x)$.

Here $\langle M, x\rangle$ denotes an unspecified but transparent way of combining the code of M and the bits of x into a single string over whatever alphabet we need. In the Turing Kit, user-designed Turing machines M are stored as ASCII files, so that can be the code $\langle M\rangle$ of M. ASCII can be converted to strings over $\{0, 1\}$ if we so desire. The files are self-delimiting, so we can then define $\langle M, x\rangle$ by just appending x to $\langle M\rangle$. Or, assuming that neither M nor x has any commas or angle brackets, we can regard $\langle M, x\rangle$ as literally '⟨' then whatever string code of M, then comma, then x, and finally '⟩'. The choice of **tupling scheme** does not matter in detail.

Proof: The *Turing Kit* is a high-level Java program P that reads a TM M and an input x and executes $M(x)$. That is (essentially), $P(\langle M, x\rangle) = M(x)$. Then compile P to M_P as above and call it U. Then $U(\langle M, x\rangle) = P(\langle M, x\rangle) = M(x)$. This notation includes that $U(\langle M, x\rangle)\downarrow$ if and only if $M(x)\downarrow$. (The down arrow means "halts" while $\uparrow$ is read as "diverges" or "does not halt.") $\boxtimes$

Both this and the next theorem are usually proved in more specialized ways in textbooks.

Theorem 6: For every nondeterministic TM N we can build a deterministic TM M such that $L(M) = L(N)$. Thus the class **RE** is the same for NTMs as for DTMs.

Proof: The Turing Kit could be upgraded to a version T' that simulates a given NTM N on an input x by branching to try all possibilities, accepting if and when some branch accepts x. The program T' itself is deterministic. Hence so is the equivalent Turing machine $M_{T'}$ obtained from T' via Theorem 1. $\boxtimes$

The one thing we don't know how to do is make T' avoid exponential branching, which slows down the time exponentially. This is different from the situation with an NFA N on a given input x, where we can simulate $N(x)$ by the trick of maintaining the current set R_i of possible states after each bit i of x, and thus avoid the exponential blowup of converting N into a DFA. Whether we can do a similar trick for a

general NTM N is the infamous NP $=?$ P problem, which we will confront in the last week of the course.

The RAM simulation does not blow up exponentially. As coded above, the time to simulate t steps the way algorithms are counted in CSE331 is $O(t^4)$ steps of the Turing machine. The biggest overhead is the need to continually "shift over" the entire register tape when needing to make more room in any given register (**purple box** routines above). A "doubling" scheme analogous to how C++ and Java manage their `vector`/`Array` classes shaves the time down to $O(t^3 \log t)$, which we can call $\widetilde{O}(t^3)$. Coupled with the quadratic slowdown of the 3-tapes-to-1 simulation (which cannot be improved in general), we get what going from an unbounded-wordsize RAM (as implicit in CSE331) to a single-tape Turing machine incurs only an $\widetilde{O}(t^6)$ slowdown. If the wordsize is fixed or the "logarithmic fair cost" measure is used for RAMs, then the time gap is only $\widetilde{O}(t^4)$. In any event we have the following theorem about "**robustness**" of the definition of polynomial-time computation:

Theorem 7: If a function $f(x)$ is computable in $n^{O(1)}$ time on a RAM, i.e., as reckoned in standard classical algorithm analysis, then it can be done in $n^{O(1)}$ time on a single-tape Turing machine. ☒

The Church-Turing Thesis and the Quantum Challenge

The equivalence of high-level programming languages (HLLs) and Turing machines is the greatest evidence for the following claims of universal law---in several senses of "universal":

Church-Turing Thesis (three-part version):

1. Any HLL ever devised will have the same computing power as the Turing machine. I.e., given a notion of "accepting" an input---such as outputting 1 or exit code 0 or executing `System.exit(0)`---the resulting classes RE and REC are the same as for DTMs.
2. Any physical device that will ever be built will have no more computing power than a Turing machine.
3. For any human being H who follows a consistent functional procedure to convert (sensory) inputs x into outputs y, there exists a Turing machine M_H that on the same inputs x (under a natural string encoding, e.g., pixels for optical input) outputs the same values y. Moreover, M_H has comparable program size and efficiency to the "grey matter" of H, or better.

Plank 1 is often considered a "truism" but maybe it depends on plank 2. Plank 2 maybe harks back to plank 1 because ***computability*** is an abstract notion for languages and functions over infinitely many strings---almost all of which have more chars than the number of quarks in te observed universe, which is under 2^{270}. Plank 3 comes from Alan Turing's seminal 1936 paper titled, "On Computable Numbers, With an Application To the *Entscheidungsproblem*"---but gets an interpretive twist from AI today. The **Turing Test** used to be considered to be *about* chatbots: can a chatbot be good enough that no one---over a connection showing just the chat---can distinguish it from a human respondent? Well, this can

be more general: e.g., a chess cheater using a computer is hoping to avoid being distinguished from a human player. My formulation is more specific even in the chatbot context, though: is there a chatbot that is indistinguishable **from you**? Well, if you are really a machine to begin with, then this is tautologically true... Less tautological and more controversial, the program and memory size S needed to simulate human cognition is the threshold that "The Singularity" talks about.

The "Part Deux" of the C-T thesis is often ascribed to Alan Cobham and Jack Edmonds from papers they wrote in 1965, in which they justified **polynomial time** as a benchmark for feasible problem-solving:

Polynomial-Time C-T Thesis: As above, plus the assertion that whatever the HLL and/or device physically implementing its programs, there will always be a constant k such that whatever the program/device does in time t can be emulated by $O(t^k)$ steps of the Turing machine.

This was almost-universally believed until 1994, when Peter Shor proved:

Theorem Seven: Quantum computers can factor n-digit numbers in $\widetilde{O}(n^2)$ time.

This is idealized---no one has yet built quantum technology that can *scale up*. For comparison, the security of most Internet commerce and many other cryptosystems relies on concrete scaling of the belief that factoring requires roughly $2^{\Omega(n^{1/3})}$ time, well maybe $2^{\Omega(n^{1/4})}$ or $2^{\Omega(n^{1/5})}$ time in most cases... [See this about the 1992 movie *Sneakers* and this review of the novel *Factor Man*.]

But as long as we stick with "classical" machines---meaning non-quantum hardware---we can take both theses as given. (Note: Actually, transistors and other chip elements *are* quantum devices, but the point is that they treat information in the classical manner of *bits*, as opposed to *qubits*.) The import is:

The classes REC, RE, *and co*-RE, *and later* P, NP, *and co*-NP, *remain the same whenever we transfer their defining notions to any HLL or classical machine model. Moreover, it is perfectly legitimate to describe Turing machines via pseudocode---provided the pseudocode gives enough detail to pin down the running time t within a linear $O(t)$, a quasi-linear $\widetilde{O}(t)$, or at worst a polynomial $t^{O(1)}$, factor.*

For example, the 2-tape TM we built to recognize $\{a^m b^n : m = n\}$ can be described by saying, "Copy leading a's to tape 2, then count against b's on the rest of tape 1, and accept iff the counts are equal and the end is reached on tape 1 without any further a appearing. Runtime: $O(m+n)$ steps, which is linear in the length $m+n$ of the input."