## CSE491/596 Lecture Mon. 2 Oct. 2023: Defining Computability and (Un)Decidability

[Class began with some general remarks and review of Assignment 2, problem (1).]

**Definition**: For any language $A$ over an alphabet $\Sigma$, or function $f : \Sigma^* \to \Sigma^*$:

- $A$ is **computably enumerable** (**c.e.**) if there is a TM $M$ such that $L(M) = A$.
  - Synonyms: **recursively enumerable** (**r.e.**), *Turing-acceptable*.
- $A$ is **decidable** if there is a total DTM $M$ such that $L(M) = A$.
  - Synonym: **recursive**. (Avoid the term "recognizable"---it is used both ways).
- $f$ is **computable** if there is a transducer $M$ that computes $f(x)$ for all $x \in \Sigma^*$.
  - Note that writing $f : \Sigma^* \to \Sigma^*$ standardly means that the domain of $f$ is all of $\Sigma^*$, so any $M$ computing $f$ must be total. But we often say $f$ is **total computable** to remind about this and clarify when we are not allowing $f$ to be a *partial function*. Other synonyms: **recursive** function, **total recursive**.

Here is a helpful little proposition that helps in understanding these concepts. Recall that with the Myhill-Nerode theorem, we have been writing $L(x)$ as if the langauge $L$ is a Boolean-valued function. We can distinguish the function from $L$ by calling it $\chi_L(x)$, where $\chi$ is the Greek letter chi to stand for characteristic function.

**Proposition**: A language $L$ is decidable if and only if $\chi_L$ is a total computable function.

The proof is "by good housekeeping." The important contrast is that when $L$ is only known to be c.e., then $\chi_L$ need not be computable: on some $x \notin L$, the machine might never halt. For a pivotal example, consider the language of the "3n+1 Problem" shown in the opening week:

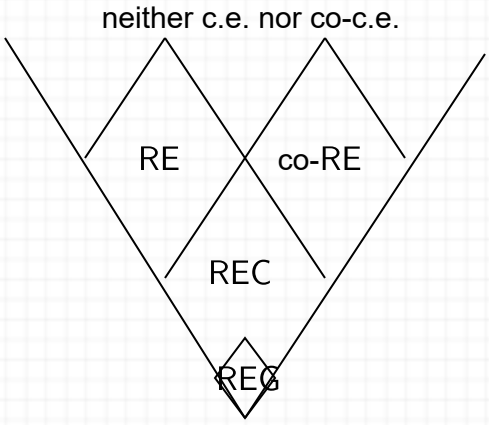
$L = \{x \in \mathbb{N}^+ : (\exists\, r)\, f^r(x) = 1\}$, where $f(x) = \textit{if } x \textit{ is even then } x/2 \textit{ else } 3x+1.$

We can regard binary numbers and binary strings as interchangeable, in various ways. One way specific to $\mathbb{N}^+$, meaning the positive natural numbers, is to delete the leading $1$ in standard binary notation, which gives a 1-to-1 correspondence to a language $L'$ over $\{0,1\}^*$. The demo showed a particular TM $M$ that ends on a single $1$ whenever $x \in L$ but does not halt otherwise.

- The *Collatz conjecture* says that $L$ equals all of $\mathbb{N}^+$, likewise $L' = \Sigma^*$. Then $M$ is actually total and that makes $L$ "trivially" decidable.
- But all we *know* at this point is that $L$ is computably enumerable. The $M$ shown in the demo is, I believe, the *tiniest* program that no one has been able to prove is total.

If a language is not decidable, it is called **undecidable**. This includes c.e. languages that are not decidable. Starting next week we will cover techniques for showing that languages are undecidable. It helps to have notation to map out *classes* of languages:

- The class of c.e. languages is denoted (only) by RE.
- The class of decidable languages is denoted by REC (occasionally, DEC).
- The class of regular languages is denoted by REG. The facts that every regular language is decidable, and some decidable languages are not regular (such as $\{a^n\ b^n\}$) can be neatly captured by writing REG $\subset$ REC.
- The classes of langauges recognized by deterministic and nondeterministic PDAs are denoted by DCFL and CFL, respectively, as we have seen.
- The classes of languages recognized by deterministic and nondeterministic LBAs are denoted by DLBA and NLBA, respectively.
- The progression REG $\subset$ CFL $\subset$ NLBA $\subset$ RE is called the *Chomsky Hierarchy* after Noam Chomsky, who characterized these classes via notions of *grammars*. One can insert DCFL and REC and keep a proper progression, but the corresponding grammar notions are "wonky" in the former case and *nonexistent* in the latter.
- However, although CFL $\subset$ DLBA, whether DLBA is properly contained in NLBA is unknown. It is rather like the P versus NP question. We will not address grammars but we will later see that DLBA and NLBA equal deterministic and nondeterministic space, respectively.
- For any class $C$, the complements of languages in $C$ form the class $co$-$C$. Note that since the complement of a regular language is always regular, we have $co$-REG $=$ REG; the $co$- does not mean "not regular" here.
- We will concentrate on REC, RE, $co$-RE, and "neither c.e. nor co-c.e." for the two coming weeks. Here is a little roadmap:




This diagram conveys some extra information:
◎ REC is closed under complements,
◎ RE $\cap$ co $-$ RE $=$ REC, and
◎ All three classes are closed downward under *computable many-one/mapping reductions*.
We will prove these after we establish the equivalence between Turing machines and high-level programming languages.

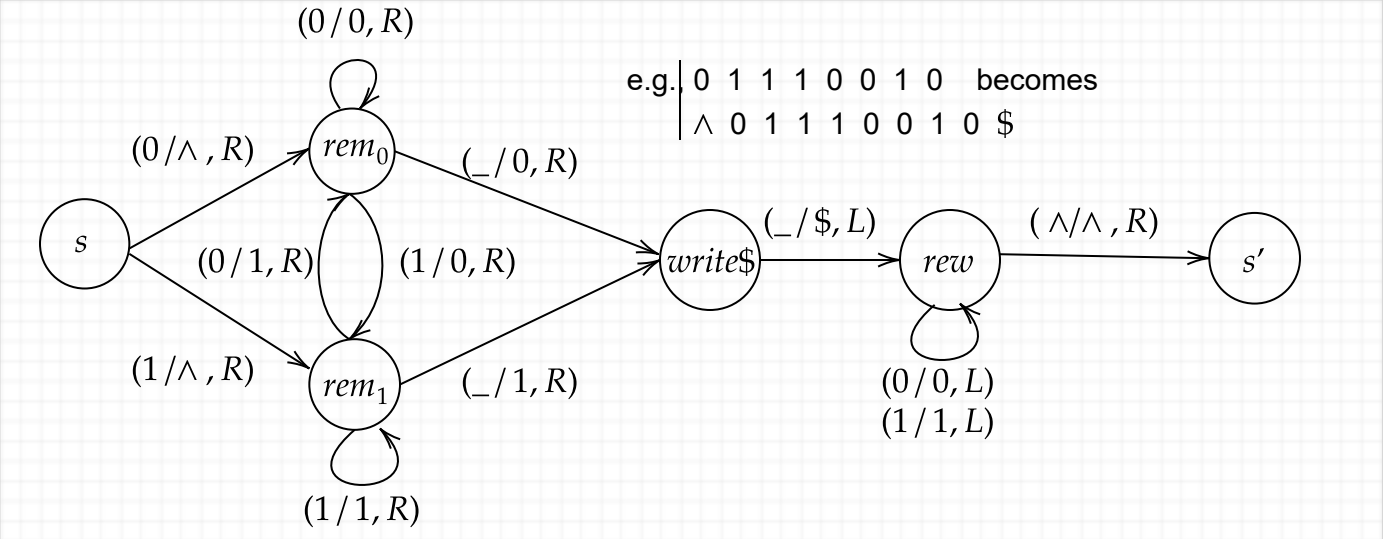
Our next objective is to convey that these concepts apply not just to Turing machines but to the full extent of computing. We will show that deterministic Turing machines are reasonably efficiently equivalent to high-level programming models [quantum ones maybe not so efficiently].

## From Turing Machines to "Real" Programs

What kinds of operations can Turing machines carry out? We have seen most of the following:

1. Copy a string from one tape to a second tape
2. Compare two strings on separate tapes to test whether they are equal
3. Search leftward or rightward on a tape until reaching the end or a sought-for char
4. Loop with back-and-forth *passes* until an exit condition is met
5. Multiply numbers by 2 (by appending a 0), or divide by 2 if even, or multiply by 3...
6. Add two binary numbers
7. Remember a char in a state while shifting an entire string over one cell
8. Use dedicated states to write a dedicated string to a desired location

We haven't seen the last two. Here is an example of how to insert a $\wedge$ in front of a binary string $x$ and also put a $ after it:




This shows that the " $\wedge$ convention" for $I_0(x)$ can always be emulated by the bare-startup convention, at the cost of only $2n + 1$ extra steps on inputs of length $n$. Moreover, if we need to insert a special char, say @, in the middle of a tape string at any given state $q$, we can attach this entire routine at state $q$ by making $s = s' = q$ and using @ in place of $\wedge$, except that the last arc becomes $(@ / @, S)$ so that the head is scanning @ in state $s' = q$ and so can execute an option that was not available before. This "shift-over" routine can thus act like an invokable process that returns control to its point of call. We can repeat it to make more room. We can also compose it with operation 8 by having more states in place of "$write\$$" that lay down whatever fixed string $y$ we want to append after $x$.

We can make other operations by composing two or a few of the above. By combining 2 and 3 we can solve the problem of finding a substring $w$ inside a larger string $x$ (by testing place-by-place, but there are also quicker ways used by compilers). We can multiply two binary numbers by using repeated addition or by using shifts to emulate the grade-school algorithm and adding up the shifted copies of the first number. This small vocabulary of machine ops suffices to simulate a rudimentary assembly

language. The following one is coded to use just one argument for each instruction, but it is fairly flexible: it even has indirect load (LDI) and store (STI):

1. LDL $n$ : load a hard-coded literal integer $n$ into the ALU
2. LDR $Y$ : load the contents of register $Y$ into the ALU
3. LDI $Y$ : read the contents of $Y$ as another address $Z$, then do as in LDR $Z$
4. STO $Y$ : copy the contents of the ALU into register $Y$, replacing whatever is there
5. STI $Y$ : read $Y$ to get the indirect address $Z$, then do as in STO $Z$.
6. ADD $Y$ : add the contents of register $Y$ to the number currently in the ALU
7. SUB $Y$ : subtract the contents of register $Y$ from the number currently in the ALU
8. SHF $d$ : shift the ALU by the hard-coded number $d$ of places (shift left if $d$ is negative)
9. ABS : take the absolute value of the number in the ALU
10. JMP $\ell$ : jump to the hard-coded instruction number $\ell$ if the ALU currently holds $0$.

Indeed, the main reason for real assembly languages having many more primitive instructions is having different types and sizes of operands: 8-bit char, 16-bit int, 32 or 64-bit float, etc. Whereas a real "RAM computer" has fixed-size registers, our Turing machine can emulate arbitrary-size registers thanks to how the "shift-over" routine can be sprinkled into its state code to make any extra room needed to store a bigger value. Thus our "mini assembly" language is actually rich enough to be a compilation target for any high-level programming language (ignoring special object features put at machine level and the like).

The "Universal RAM Simulator" handout
https://cse.buffalo.edu/~regan/cse396/UTMRAMsimulator.pdf
has enough hand-drawn detail to serve as proof-of-concept. Immediately what it proves is:

**Theorem 1**: We have built a single DTM $U$ such that for any mini-assembly program $A$ and integer argument $x$ to $A$, $U$ on input $\langle A, x\rangle$ outputs the result (if any) of running $A$ on input $x$. ☒

[Monday's lecture stopped here; what follows is blended into the notes for Wednesday.]

Here the angle brackets in $\langle A, x\rangle$ stand for "some transparent way of combining $A$ and $x$ into a single string." We've already been using this notation with IDs, in case we would want to read them as strings (as later, we will). One way to implement it is fine provided the angle brackets and comma don't occur inside $A$ and $x$: we can treat them as literal characters over, say, the ASCII or UNICODE alphabets---which we can then convert to binary if we wish. Another that works in this case is to just ram the two strings together as $xA$ or $Ax$, which is fine in the handout since $A$ begins with a ! and ends with a semicolon, both of which we suppose do not occur inside $x$. In general, one can regard the angle brackets as applying a *pairing function*. (Some sources devote time to pairing functions, which can be composed to encode any tuples as strings and also decode them, but we can dispense with the details.)

Now we add a further wrinkle using operation 8 above when $A$ is a single fixed program, rather than one given on-the-fly.

**Theorem 2**: Given any program $P$ in any known high-level programming language (HLL) that uses standard input and output, we can build a Turing machine $U_P$ such that for any input $x$ to $P$, $U_P$ on input $x$ replicates the stream output of $P(x)$. In particular, if $P$ computes a function $f$ then $U_P$ computes the same function---and moreover, does so with roughly comparable efficiency.

**Proof**: First use our compilation target to create a mini-assembly program $A_P$ that simulates $P$. Now $A_P$ is a fixed literal string over the ASCII alphabet as encoded in the handout. We can therefore create $U_P$ to use something like our "shift-over" routine to convert its input $x$ into the string $\langle A_P, x \rangle$, which is just $A_P x$ in the handout, on the first tape. (Or we could use $x A_P$ which would work similarly.) Then we rewind the head on the first tape and send control to the start state of the fixed program $U$ we wrote in Theorem 1. Then for any $x$,

$$U_P(x) \simeq U(A_P, x) \simeq A_P(x) \simeq P(x),$$

where the "\simeq" symbol $\simeq$ says the computations must give the same result if they converge but allows both to diverge, as of course can happen.

To address efficiency, note first that $P$ and $U_P$ use basically the same volume of memory registers, so the *space* usage is about the same. The TM $U_P$, however, needs extra time because it does not enjoy true *random access*---it has to scroll up and down the whole line of registers to find a desired one. The length $s$ of that line of registers, however, includes only the ones that have been previously allocated, and those allocations used at least $s$ steps of $P$. The linear search by $U_P$ compounds that time by an $O(s)$ factor per next instruction of $A_P$. Since $s$ is also at most the total time $t$ taken by $P$ up to a given point, the total compounded time is $O(t^2)$. There is, however, a further complication: The "shift-over" routine may need to be invoked to make more room after repeated addition, for instance. The literal code in my handout would bump up the time to $O(t^3)$ or maybe even $O(t^4)$, depending on implementation details. However, in 1972, Stephen Cook and Robert Reckhow of Toronto worked out a clever caching scheme by which, if $P$ uses the *fair-cost* time measure, which charges for the lengths of the operands in a RAM instruction, then the time goes back down to $O(t^2)$. $\boxtimes$