University of Nevada, Las Vegas Computer Science 456/656 Fall 2021

These are questions sent to me by a student in the class. After each question, I will give a response.

Dr. Larmore, As requested, here is a list of concepts I'm struggling to comprehend:

1. Context-free/Context-Sensitive: What exactly do these mean in the scope of our class and how do they relate to programming languages?

   Response: These are grammar classes defined by Noam Chomsky. Each grammar class gives rise to a language class, the class of all languages derived from grammars of that class.

   Modern programming languages are not context-free. I am not sure whether they are context-sensitive. However, there is typically a CF grammar used to parse programs. This seems like a contradiction.

   Consider the programming language Pascal. There is a CF grammar, call it G, for Pascal in the back of the textbook, where the productions are sometimes given as "railroad diagrams," instead of the usual way productions are written.

   The first part of a compiler is the lexical analyzer, and the second part is the parser. Every Pascal program can be parsed by a parser, basically, a DPDA with output, using the grammar G. However, the parser will accept some strings which are not valid programs. These will not be found to be invalid by the parser, but rather by a later part of the compiler. Let $L(G)$ be the language generated by G, i.e., the language accepted by the parser. Then Pascal $\neq L(G)$, but Pascal $\subset L(G)$.

2. NFA vs. DFA (Decidable or not, the concept is unclear to me): Is there anything specific this conversion accomplishes other than a state reduction? Can you possibly provide examples of what an NFA vs. a DFA are again and what specifically makes them (with respect to the visual aspect in state machine form) nondeterministic or deterministic?

   Response: Decidability is not an issue. A language is regular if and only if it is accepted by some DFA. Also, a language is regular if and only if it is accepted by some NFA.

   Think of a DFA for a regular language L as a computer program, where the computer has finite memory. If you enter a string of symbols w, then the program will write "Yes" if and only if $w \in L$.

   It is a bit harder to visualize an NFA. The famous baseball player Yogi Berra, giving directions to his house, once said, "When you come to the fork in the road, take it."

   An NFA can be thought of as a program with choices. If the program encounters a choice between two or more computation paths, it picks ("guesses") one of them arbitrarily. We say that an NFA M accepts a language L if the following two conditions hold:

   (a) If M is run with input $w \in L$, it is possible for M to output "Yes." In order to do this, M must make all the right choices.

   (b) If M is run with input $w \notin L$, M cannot output "Yes," regardless of its choices.

   The problem of making the right choices can be eliminated by replacing an NFA with an equivalent DFA. Every NFA with n states is equivalent to a DFA with at most 2^n states.

3. Grammar: What exactly is encompassed within grammar? How would you like us to present the grammar of a language (syntax-wise on exams, homework, etc.)?

4. Derivations: Can you please provide a few more derivation examples and relate them to parse trees?

   Response: In our course, the most important grammars are context-free grammars. A CF grammar G has an alphabet of terminals, which we call Σ, an alphabet of variables, which we call Γ, and a finite set of productions, also called replacement rules. The terminals and variables are together called grammar symbols. There is always one variable which is the start symbol, usually (but not always) called S.

   For a CF grammar, a replacement rule has a left hand side, which is a variable, and a right hand side, which is a string of grammar symbols.

   A derivation of a string w is a sequence of strings, which are called sentential forms, starting with the start symbol and ending with w. Each string in the sequence, except the first, is derived from the previous string by using one replacement rule, replacing the left hand side of that rule by the right hand side.

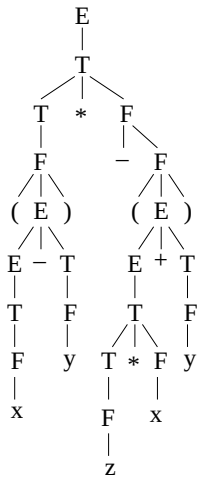
   Example. Consider the CF grammar given on page 3 of homework 3, with terminals $\Sigma = \{+, -, *, (,), x, y, z\}$. variables $\Gamma = E, T, F$, and start symbol E, and let $L = L(G)$.

   1. $E \to E + T$
   2. $E \to E - T$
   3. $E \to T$
   4. $T \to T * F$
   5. $T \to F$
   6. $F \to -F$
   7. $F \to (E)$
   8. $F \to x$
   9. $F \to y$
   10. $F \to z$

   The string $w = (x-y)*-(z*x+y)$ is in L. Here is a derivation of w, along with the corresponding derivation tree.

   $$E \overset{3}{\Rightarrow} T \overset{4}{\Rightarrow} T*F \overset{5}{\Rightarrow} F*F \overset{7}{\Rightarrow} (E)*F \overset{2}{\Rightarrow} (E-T)*F \overset{3}{\Rightarrow} (T-T)*F \overset{5}{\Rightarrow} (F-T)*F \overset{8}{\Rightarrow} (x-T)*F \overset{5}{\Rightarrow} (x-F)*F \overset{9}{\Rightarrow} (x-y)*F \overset{6}{\Rightarrow} (x-y)*-F \overset{7}{\Rightarrow} (x-y)*-(E) \overset{1}{\Rightarrow} (x-y)*-(E+T) \overset{3}{\Rightarrow} (x-y)*-(T+T) \overset{4}{\Rightarrow} (x-y)*-(T*F+T) \overset{5}{\Rightarrow} (x-y)*-(F*F+T) \overset{10}{\Rightarrow} (x-y)*-(z*F+T) \overset{8}{\Rightarrow} (x-y)*-(z*x+T) \overset{5}{\Rightarrow} (x-y)*-(z*x+F) \overset{9}{\Rightarrow} (x-y)*-(z*x+y)$$

   Note that this is a left-most derivation.



5. Recursive Enumeration: is this simply the repetitive (recursion with respect to computers and their functions) counting a machine performs?

   Response: "Recursive" means computable by a machine. It has nothing to do with recursion as you use it in programming.[1]

   An *enumeration* of a set S (for example, of a language) is a sequence of members of that set such that every member of S is a term of the sequence. For example, $1, 2, 3, \ldots$ is an enumeration of the natural numbers. A set S is said to be enumerable, or countable, if it has an enumeration.

   Existence of an enumeration of S does not imply that you can "find" one, that is, design a machine that prints an enumeration of S. Every language is countable, that is, enumerable, but you may not be able to compute an enumeration of the language. If such a computation exists, we say that the language is *recursively enumerable.*

   Any subset of an enumerable set is enumerable, but a subset of a recursively enumerable set may not be recursively enumerable. Let $\Sigma = \{0, 1\}$, the binary alphabet. Σ^*, the language of all binary strings, is recursively enumerable. By encoding all machine descriptions as binary strings, we have DIAG$\in \Sigma^*$. But DIAG is not recursively enumerable.

[1] Actually, it does, but that's too deep for this class.

Every decidable language is recursively enumerable, but not vice versa. HALT is recursively enumerable but not decidable.

6. P-Space: I'm having trouble with this one... Is it essentially a region or categorization of language classes? Potentially a classification that encompasses all subsets?

   Response: The class $\mathcal{P}$-SPACE is interesting, but has lower importance than $\mathcal{P}$-TIME, in my opinion. We say that a language L is in the class $\mathcal{P}$-SPACE if the membership problem for L, i.e., whether a given string w is in L, can be decided by a machine (program) that uses no more memory than some polynomial function of $|w|$.

   Sliding block puzzles are in the class $\mathcal{P}$-TIME. That means that if you have a start configuration of a sliding block puzzle, such as Rush Hour, you can decide whether you can win by running a program which uses polynomial memory.

   Regular expression equivalence is $\mathcal{P}$-SPACE complete.

   No one knows whether $\mathcal{P}$-SPACE = $\mathcal{P}$-TIME.

7. NP-Time: I understand that P-time is decidable in Polynomial time, but can you provide an example of a machine or language that requires NP-time to complete? I want to understand this in mathematical terms, so an equation might be more helpful for me to connect the dots.

   Response: $\mathcal{NP}$-TIME is not a measure of time. Rather, it is the class of languages which are accepted by some non-deterministic machine in polynomial time. No one knows whether $\mathcal{P}$-TIME is the same as $\mathcal{NP}$-TIME, so no one can provide you with a language that is known to be in $\mathcal{NP}$-TIME but not in $\mathcal{P}$-TIME

   A language L is in $\mathcal{NP}$-TIME if and only if there is some integer k such that any $w \in L$ can be proved to be in L by a proof whose length is at most $|w|^k$.

   A language L is in co-$\mathcal{NP}$-TIME if and only if there is some integer k such that any $w \notin L$ can be proved to not be in L by a proof whose length is at most $|w|^k$.

   Integer factorization is known to be in both $\mathcal{NP}$-TIME and co-$\mathcal{NP}$-TIME, but not known to be in $\mathcal{P}$-TIME. If it is, then RSA coding is breakable in polynomial time.

8. Regular: Can you provide a more concrete definition than what was provided in class by any chance, perhaps in layman's terms? All I understand is that a Regular language is accepted by some DFA.

   Response: Since you are a CS student, you are not a "layman." You are required to understand CS topics in technical terms.

   Modern programming languages allow you to ask for additional memory in various ways. If your program implements a stack using a linked list, there is (in principle) no limit on the amount of memory the stack uses.

   A language L is regular if and only there is a program M which decides the membership problem for L which never calls for additional memory no matter how long the input string is. That implies that, for example, the program cannot have pointer variables, recursive functions, or variable length arrays. Such a program can always be modeled by a DFA.