CS 241 Context-Free Languages and Grammars Handout

1.1 Recall DFAs

Continuing on from Regular languages, we start to identify the boundary of this class of languages and give some examples of languages that we are unable to build a DFA/NFA/ϵ-NFA.

Recall: In a DFA, a states represent the "memory" of the finite automata. Having read a sequence of input symbols, a state in the finite automata represents where in the recognition process the DFA is at in identifying if the input string will match one or more of the patterns accepted by the DFA. For example, a DFA for

$$L_1 = \{\text{all string with an even number of } \texttt{a}\text{'s that also have } \texttt{caa} \text{ as a substring}\}$$

may have states representing:

- even number of `a`'s and have not starting building substring `caa`
- even number of `a`'s and last character was `c`
- even number of `a`'s and last two characters were `ca`
- even number of `a`'s and found `caa`
- odd number of `a`'s and have not starting building substring `caa`
- odd number of `a`'s and last character was `c`
- odd number of `a`'s and last two characters were `ca`
- odd number of `a`'s and found `caa`

As the DFA reads each input symbol (one at a time, never going back or looking ahead), it would transition between these states. For the language L_1 the state representing "even number of `a`'s and found `caa`" would be an accepting state.

1.2 Limits of Regular Languages and DFAs

Now consider a language $L_2 = \{0^i 1^j \text{ where } i \leq j\}$ where $\Sigma = \{0, 1\}$.

If asked to write a program, in a high-level language such as C/C++, to identify if a string w from $\Sigma^* = \{0, 1\}^*$ belongs in L_2 it would be a simple process - one method is to increment a *counter* each time we read a 0 and decrement each time we read a 1 (also checking that we only had 0s first followed by 1s). If after reading all characters of w, the *counter* is not positive then accept. The *counter* in this case is the memory our program

used to keep track of the state the program was at each time it processed a new input symbol.

To create a DFA for L_2, we want to simulate the same idea as above but instead of having a variable *counter*, we need to use the states of the DFA to keep track of the value of the counter. Initially we might think we need states:

- counter is 0
- counter is positive
- counter is negative

but this isn't enough. As the intial 0's are read, an exact count is required to then decrement for each 1 that is read. However, $\Sigma^* = \{0, 1\}^*$ includes strings with any number of initial 0's and similarly, L_2 also includes strings strings with any number of initial 0's, as long as, there are more 1's following. So, the count can be any integer value, an infinite number of possibilities. However, our DFA can only have a finite number of states.

Common Error: If asked to justify why we cannot build a DFA for L_2 some students might say, "We can have a string with an infinite number of 0s that a DFA cannot count since it has finite states.". The idea here is okay but the statement is not correct. Any (every) string used as input to a DFA is a fixed string of finite length so it cannot have an infinite number of 0's. The idea here is rather that the number of 0's in a string is unbounded, so for any DFA with a finite number of states, we can find an input string with more 0's than states in the DFA and the DFA can then not properly keep count of the number of 0's in this string.

A similar example related to building a compiler is to check that all opening parethesis '(' have a following, matching closing parenthesis ')' in arithmetic expressions or in C/C++ that nested blocks have matching pairs of { and }; this is frequently called the language of balanced parenthesis. The number of nested parenthesis (or braces) is not bounded so a DFA with finite number of states is not able to recognize this language.

Exercise: Consider $L_3 = \{$Strings of balanced parenthesis where nesting is limited to k levels$\}$ for some fixed value of k. Is this language Regular? Can you build a DFA for this language?

Since Regular languages and DFAs are not powerful enough to recognize these languages, we need something more: Context-Free Languages (CFLs) and Context-Free Grammars (CFGs).

1.3 Context-Free Languages and Context-Free Grammars (CFGs)

Formally, a Context-Free Grammar is a 4-tuple:

- Σ a finite, non-empty alphabet of terminal symbols

- N a finite, non-empty set of non-terminals (often also called variables)
  Note: in CS 241, we use V ("vocabulary") for the set $N \cup \Sigma$
- P a finite set of productions of the form $A \to \beta$ where $A \in N$ and $\beta \in V^*$
- S a start symbol where $S \in N$

A string within the language of the grammar is derived from the starting symbol, a non-terminal S, and then a sequence of substitutions where at each step a single non-terminal is replaced with the right side of a production rule until no non-terminals remain. All strings that can be derived this way compose the language of the grammar. A production rule $A \to \beta$ defines that A can be substituted with β, a sequence of terminals and non-terminals.

Example: A context-free grammar for balanced parenthesis:

$S \to \epsilon$
$S \to (S)$
$S \to SS$

Since S is the starting symbol, it describes words in the language; i.e. anything that derives from S is a word in the language.

- The empty word is in the language.
- The word in the language with an opening and closing parenthesis around it is also in the language.
- The concatenation of 2 words from the language is also in the language.

In this case, we built the grammar from a recursive definition of balanced parethesis so the left side non-terminal also appears in the right side of the production rules.

Instead of writing out each rule in full, we can group all the rules with the same left non-terminal separating them with |:

$S \to \epsilon \mid (S) \mid SS$

Conventions, Notation and Terminology

Conventions

- $a, b, c, \ldots$ terminals (also called symbols or characters) from Σ.
- $w, x, y, \ldots$ words (also called strings) from Σ^*.

- $A, B, \ldots, S, \ldots$ non-terminals (also called variables) from N.
- S start symbol, a non-terminal from N.
- $\alpha, \beta, \gamma, \ldots$ words from $V^* = \{N \cup \Sigma\}^*$.
  These words may contain both terminals and non-terminals and use in the right side of production rules.

Notation and Terminology

- $\Rightarrow$ means "derives" in one step (do not mistake this for $\rightarrow$ used in the definition of production rules).
- $\alpha \Rightarrow \beta$ means β can be derived in one step from α; i.e. by one application of a grammer rule.
- $\alpha \Rightarrow^* \beta$ means β can be derived from α in 0 or more derivation steps; i.e. $\alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_k = \beta$ where $k \geq 0$.
- We write $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if there is a production $A \rightarrow \gamma$ in P.
  Note: we can replace A into γ regardless of its context (regardless of what α and β are).

Definition: Language of a Grammar
Language L is context-free if it is the language of a context-free grammar.
Given a CFG G, the language of G, $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$.
All words in the language

- are derivable from the start symbol S
- contain no non-terminals

Exercise: Give a CFG for $L_2 = \{0^i 1^j \text{ where } i \leq j\}$ where $\Sigma = \{0, 1\}$.

1.4 Creating Context-Free Grammars

Building a CFG from an English description, mathematical description, regular expression, etc for a language can be a challenging task. The previous examples have been relatively simple and only required a single non-terminal. Consider some more challenging language descriptions.

Example: Give a CFG for: $L_4 =$ the language of $(a(bb)^*a(a + \epsilon))$.

Similar to programming, a useful skill is to be able to identify how the problem can be broken into smaller pieces - what patterns are common to all strings in the language, which pieces have variations, what order do we put things in, etc. Observe:

- $a(bb)^*a$ is a common prefix of all strings in the language
- $(bb)^*$ allows the number of b's to vary
- all strings in the language may end with an additional a or not

We can use different non-terminals to build the different pieces identified above:

$S \to aBaE$
$B \to bbB \mid \epsilon$
$E \to a \mid \epsilon$

We can remove E by adding the two possible ending patterns directly into S:

$S \to aBaa \mid aBa$
$B \to bbB \mid \epsilon$

Exercise: Modify the CFG above to give a CFG for L_4^*, the language of $(a(bb)^*a(a+\epsilon))^*$.

Complicated example: Given $\Sigma = \{a, b\}$, create a CFG for:

$$L_5 = \{w \mid w \text{ is a palindrome and number of } a\text{'s in } w \bmod 3 = 1\}$$

The CFG for palindromes is straightforward: $S \to aSa \mid bSb \mid a \mid b \mid \epsilon$.

Next, modify the grammar to keep track of the number of a's by creating different levels in the CFG to represent the different values of "mod 3" (while counting a's).

To begin to create the CFG, consider the base cases a, b, ϵ. An a by itself is a palindrome and the number of a's mod 3 is 1. Both b, ϵ are palindromes but have have no a's. So, we need to different rules:

$S \to a$
$B \to b \mid \epsilon$

Note: S generates palindromes where number of a's mod 3 is 1.
B generates palindromes where number of a's mod 3 is 0.
We should also add non-terminal A that will generate palindromes where number of a's mod 3 is 2.

Next, add the rules for longer palindromes to each non-terminal (S, A, B). The rules will have the forms aXa and bXb where X is the non-terminal that will generate a palindrome with the number of a's required to satisfy the property. For example:

- For S, aXa adds 2 a's so X must generate a palindrome with number of a's mod 3 = 2; i.e. replace X with A.
- For S, bXb adds no a's so X must generate a palindrome with number of a's mod 3 = 1; i.e. replace X with B.

This results in the rule: $S \rightarrow aAa \mid bSb \mid a$

Exercise: Complete the grammar for L_5 by changing each occurrence of X into the appropriate non-terminal S, A or B.

$S \rightarrow aAa \mid bSb \mid a$
$A \rightarrow aXa \mid bXb$
$B \rightarrow aXa \mid bXb \mid b \mid \epsilon$

Okay the answer is as follows (but try to do the above exercise before you look at the solution):

$S \rightarrow aAa \mid bSb \mid a$
$A \rightarrow aBa \mid bAb$
$B \rightarrow aSa \mid bBb \mid b \mid \epsilon$

Further topics discussed in class:

- Leftmost and rightmost derivations
- Parse trees
- Ambiguity
- Determining if two grammars generate the same language
- How to build an automata that recognizes CFL
- How to automatically determine the sequence of grammar rules used to derive a word, if it is part of the language