CS332 Topic 01: Formal Languages

Outcomes: By the end of these lectures you should be able to perform the following.

1. State the definition for an alphabet, string, and language.
2. State the definition of a grammar.
3. State the definition and purpose of the Chomsky Hierarchy of languages.
4. State the language classes composing the Chomsky Hierarchy (regular, context free, context sensitive, unrestricted).
5. Define the characteristics of grammars associated with each language class in the Chomsky Hierarchy.
6. Identify the Chomsky Hierarchy class associated with a given grammar.
7. Define a specific regular language using a regular expression.
8. Define a non-regular language using modified regular expressions.
9. Define a grammar for a given language in Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF).
10. Derive a string using a given grammar.

References:

Notes:

1. Motivation:
   (a) The study of computer science is the study of computation, specifically computation performed by some mindless, automated device. The method by which this is currently acheived is through the use of programming languages to provide instruction to the (mindless) computer. As a result, computer science students study *programming languages*, but they must also study the underlying nature of those programming languages. The study of the underlying nature of programming languages is the study of *formal languages*. The use of abstract formal languages allows the results to apply to all real world programming languages. This study includes language syntax (how legal programs are constructed) and semantics (what the program means). People who study formal languages can then create proofs dealing with the formal languages, with the results of those proofs applying to all real world programming languages.
   (b) Through the use of abstract formal languages, computer scientists can study the nature of computation, compilation, and the construction of languages. Coupled with the use of formal models of computation, computers scientists can study the limits of computability and the difficulty of certain classes of problems.

2. Introduction: *Formal languages* refers to the study of abstract languages whose properties underly all real world programming languages. When students think of a language, they tend to conflate the defintion of what a language is with the use of the language. For example, students tend to define language as something used to communicate, or something used to provide instruction to a computer. These are uses of language, but do not define the essence of what a language is. A language is simply defined as a set of strings (where a string is defined as a sequence of characters or symbols, see definitions below), and this definition applies to both natural languages used for communication and programming languages used for defining a specific computation. Everyone who received the comment "sentence fragment" from their English teacher knows that there are strings that are not part of the English language. Similarly, everyone who has experienced a failed attempt to compile a program intuitively knows that certain strings are not part of the programming language they are working with.

   In this section, we introduce the concept of a formal language and tools used to define them. The primary tool is the *grammar*, which provides a set or rules by which strings in the language can be produced. These grammars are expressed in the Backus-Naur Form (BNF) and the Extended Backus-Naur Form (EBNF). The EBNF provides techniques to simplify the pure BNF grammars, as a pure BNF grammar may be large and messy, making them more difficult to work with. This course will not focus on the difference between the two forms, and generally will refer to either form as BNF.

   For our purposes, a grammar is a set of rules by which strings in a language may be constructed. Basically, the grammar defines legal, or allowable, strings in the language. The BNF grammar provides these rules as a set of *production rules*. Strings in the language may be produced, or *derived*, by iteratively using the production rules. The rules have a left hand side (LHS), and a right hand side (RHS), and are interpreted as "If you have the LHS, then you can replace it with the RHS." The format is $LHS \rightarrow RHS$.

3. Definitions and Notation:

   (a) Symbol: A symbol is a single, distinct, written representation. Typical symbols used in the formal language community are $a, b, 1$ and 0.

   (b) Alphabet: An alphabet, Σ, is a finite collection of symbols.

   (c) String: a string, u, is a sequence of symbols from a single alphabet.

   (d) Length of a String: Given some string u, then the length of the string, designated by $|u|$, is the number of symbols in the string.

   (e) Empty String: The empty string, λ, is the string having no symbols. Its length is zero.

   (f) Language: A language, L, is a set of strings.

   (g) Grammar: A grammar, G, is a specification of the allowable (or legal) construction of strings in a language.

   (h) Backus-Naur Form (BNF): A *BNF grammar*, G, specifies strings in a language through a set of *production rules* composed of a left-hand side (LHS) and a right-hand side (RHS), usually connected by an arrow (some people use a colon, we'll use an arrow). The rule, then, is of the form $LHS \rightarrow RHS$, and specifies that any pattern found in a string that matches the LHS can be replaced by the RHS. Examples will be given below.

(i) Extended Backus-Naur Form (EBNF): An *EBNF grammar*, G, is simply a slightly modified BNF grammar intended to simplify the definition of some language. For example, rather than having ten specific production rules for each digit, 0 - 9, there may be one production rule resulting in $< digit >$. This simplifies the grammar without changing the specification. In this course, we will not distinguish BNF from EBNF, and wll use the term *BNF* generally to refer to any grammar.

(j) Terminal: A *terminal* in a grammar is simply a symbol in the alphabet, Σ in use. Terminals are generally digits or lower case letters.

(k) Non-terminal: a *non-terminal* in a grammar is a symbol used on the grammar that is not in Σ, and used as intermediate forms as the production rules are used to produce strings. Non-terminals are typically upper case letters $(S, A, B, C, \ldots)$.

(l) Start Symbol: Given a BNF grammar, G, the *start symbol*, S, designates the production rule that must be used first when producing a string. S is traditionally used as the start symbol for all grammars.

(m) Derivation: A *derivation* is the act of producing some string, u, using the production rules of some BNF grammar, G.

(n) Chomsky Hierarchy: The *Chomsky Hierarchy* is a categorization of languages and associated grammars. While not fully defined here, the Chomsky Hierarchy identifies four classes of languages, as shown in the table below.

Grammar G_1

Type-0	Unrestricted
Type-1	Context Sensitive
Type-2	Context Free
Type-3	Regular

4. Discussion

   (a) *Language*: The world of formal languages define a *language* quite simply. A symbol is a single mark or representation, such as the letters a and b, or the digits 1 and 2. An alphabet, Σ, is a finite set of symbols. A string, typically represented by u, v, or w, is a sequence of symbols (it does not have to be finite, but for our purposes we will only consider strings of finite length). Give that, a language, L, is simply a set of strings. IT is easy to construct a language of infinite size, meaning it contains an infinite number of strings.

   (b) *BNF Grammar*: A BNF grammar, G, is a set of production rules having the form $LHS \rightarrow RHS$, and having the interpretation that if one has the LHS in the current string, it may be replaced by the RHS. Strings are *derived* by the grammar through iterative application of the production rules, beginning with the start symbol. The rules are typically numbered, and are referenced as they are used in a derivation.

   Example: Let alphabet $\Sigma = \{a, b\}$, and let language L_1 be the strings that are composed of only a's and having at least one a. Construct a grammar, G_1, for this language:

ID	Rule
1.	$S \rightarrow a$
2.	$S \rightarrow aS$

   This grammar constructs all strings in L. For example the string aaa is derived as follows:

String	Rule Used
S	Start symbol (always begin here)
aS	2
aaS	2
aaa	1

   (c) The Chomsky Hierarchy. Chomsky defined four classes of languages. We can now provide a little more detail on the Chomsky Hierarchy:

Type	**Name**	**Characteristics**
Type-0	Unrestricted	No restrictions.
Type-1	Context Sensitive	The number of terminals in the RHS may not decrease.
Type-2	Context Free	A single non-terminal on the LHS.
		The number of terminals in the RHS may not decrease.
Type-3	Regular	A single non-terminal on the LHS.
		The number of terminals in the RHS may not decrease.
		At least one grammar can be constructed creating strings wholly from left to right, or from right to left.

(d) Example 2: Let $\Sigma = \{a, b\}$ (used for all examples, unless otherwise specified), and let L_2 be the set of strings that both begin and end with a b. Strings in L_2 include b, bb, bbb, bab, $baabbabaabbaaab$. L_2 is of infinite size.

Grammar G_2

ID	Rule
1.	$S \to b$
2.	$S \to bNb$
3.	$N \to aN$
4.	$N \to bN$
5.	$N \to \lambda$

This grammar constructs all strings in L_2. For example the string bb is derived as follows:

String	Rule Used
S	Start symbol (always begin here)
bNb	2
bb	5

As another example, derive the string $baabb$:

String	Rule Used
S	Start symbol (always begin here)
bNb	2
$baNb$	3
$baaNb$	3
$baabNb$	4
$baabb$	5

(e) Categorizing Languages: Where does L_2 fall on the Chomsky Hierarchy? Since every rule has a single non-terminal on the LHS, it is either regular of context free. If a grammar can be constructed that derives strings from left to right, or right to left, then it is regular. This can be done, as shown by grammar G_2', and L_2 is regular.

Grammar G_2'

ID	Rule
1.	$S \to b$
2.	$S \to bN$
3.	$N \to aN$
4.	$N \to bN$
5.	$N \to b$

Note: Later in the course we'll discuss other ways to show a language is regular using models of computation.

(f) Extended BNF (EBNF): The BNF form shown so far can easily become visually cluttered. The EBNF form allows for a simplified visual form, while retaining the information given in the BNF form. There are two basic simplifications. First, the line-by-line listing of rules can be simplified using a vertical bar as an *or* operator. In the grammar, G_2'' below, rules 3, 4, and 5 are collapsed into one line. That line is read as "N may be replaced by aN, or bN, or b."

Grammar G_2''

ID	Rule
1.	$S \to b$
2.	$S \to bN$
3, 4, 5.	$N \to aN \mid N \to bN \mid N \to b$

The second simplification is used when the language is built on a large alphabet. For example, if an alphabet contains 26 characters and 10 digits, the 36 rules that would be required to specify them may be replaced with two rules, such as: $N \to <\textit{character}>$ and $N \to <\textit{digit}>$. This is particularly useful when specifying actual programming languages where variable names are used. The effectively infinite legal variable names may be represented in a single rule of the form: $N \to <\textit{variable}>$.

(g) Regular Expressions: The languages, L_1 and L_2 are regular, but are specified poorly using natural language. A more precise and succinct form of representing regular languages is to use *regular expressions.* Regular expressions are defined recursively as follows:

i. λ (the empty string) is a regular expression.

ii. $\forall a \in \Sigma$, a is a regular expression.

iii. If u and v are regular expressions, then $u + v$ is a regular expression. The $+$ operator is the OR operator for regular expressions, and this represents u OR v.

iv. If u and v are regular expressions, then uv is aregular expression (concatenation).

v. If u is a regular expression, then u^* is a regular expression. The * operator is called the *Kleene star* and represents zero or more repetitions of u.

vi. If u is a regular expression, then u^+ is a regular expression. The $+$ operator represents one or more repetitions of u, and is equivalent to uu^*.

Using regular expression, languages L_1 and L_2 can be more succintly expressed as:
$L_1 = a^+$ (strings composed of only a's, and having at least one a.)
$L_2 = b + b(a + b)^*b$ (strings composed of a single b, or beginning an ending with a b with anything in between them.)