Grammars and Languages

This chapter will complete our examination of theoretical models of computing. We have the indention of examining the question “What is computable?”, with the hope of arriving at a definition a bit more precise than “It is what computers can compute”. What can be computed by an automatic electronic device called a computer? We hope to attempt an answer to that question in terms that are more basic than the intuitive ideas we have all developed by actually using the electronic devices called computers.

At this point, we should make the disclaimer that nothing requires a computing machine to be electronic, as there are historical examples of purely mechanical computers. Having noted that distinction, we focus on electronic computing machines as computers, not only because such devices are almost the only computers in use today, but also because that model is sufficiently broad to encompass anything we might intend when speaking of a “computer”.

We shall also ignore the fact that the term “computer” historically referred to a person who does computations; in other words a job title. To support this irrelevant idea, we quote the following two definitions from the 1933 Oxford English Dictionary.

> “**Calculator** – One who calculates; a reckoner”.
>
> “**Computer** – One who computes; a calculator, reckoner; specifically a person employed to make calculations in an observatory, in surveying, etc.”

This last usage of the word “computer” is seen in a history of the Royal Greenwich Observatory in London, England. One of the primary functions of this observatory during the period 1765 to 1925 was the creation of tables of astronomical data; such work “was done by hand by human computers”. While this is interesting, our investigations must focus on electronic stored-program computing machines, now called “computers”.

Finite Automata and Language Recognizers

We continue our study of theoretical models of computation by framing the question in the terms that are most approachable theoretically – what languages are recognized by a given model of computation. Now that we have said it, we must explain what we have just said. It turns out that we have two ways to approach this problem, either the study of finite automata (already discussed briefly) that can be said to recognize certain languages, or the study of formal grammars, that can be said to generate these languages. We now begin a discussion of the idea of formal grammars and then connect the grammars to specific classes of finite automata. As before, these topics are easier than they seem.

Here is an example of a computation that can be viewed as language recognition. Given a positive integer, N, we are to determine whether or not it is a prime number. One of the few reliable algorithms for primality testing is the Sieve of Eratosthenes, often called “the sieve”. This algorithm can be programmed easily. Now consider the integer as represented by a binary string of K bits. We present this string to a FA that recognizes prime numbers. If the number is prime the FA is said to recognize it, otherwise it does not. The set of prime numbers is the language recognized by this FA.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

Strings and Words

We begin with the idea of an **alphabet**, which is defined to be a finite non-empty set of elements called "symbols" or, more informally, "letters". A **string** over an alphabet is a finite sequence of elements from that alphabet. The **length** of a string (word) is the number of elements in the string or word. One string, called the **empty string** and denoted by λ, is important for theoretical considerations.

So far, the definitions have followed exactly what would be expected from the standard definitions using a dictionary as a list of words, with the exception of the empty string. The empty string may seem a strange idea, but it is required by the theory.

The empty string λ is most often used in recursive definitions. The length of the empty string is 0 as it contains no symbols. We denote the length of an arbitrary string s by $|s|$, so that $|s|$ is the number of symbols (elements) in the string (word). Obviously $|\lambda| = 0$. We note that the empty string is unique, so that $|s| = 0$ implies that $s = \lambda$; we speak of "**the** empty string".

Suppose s is an arbitrary string with length $|s| = k > 0$. Then either

1) $k = 1$, $s = a_1$, a single character, and $|s| = 1$or
2) $k > 1$, $s = a_1a_2\ldots a_k$ is a sequence of k elements of A, and $|s| = k$.

If s_1 and s_2 are two strings over an alphabet A, then the **concatenation** of s_1 and s_2 is the string s_1s_2, which is the string s_1 followed by the string s_2. More formally, we have the following definition of concatenation.

1) If s is an arbitrary string and the empty string, then $s\lambda = \lambda s = s$.
2) Otherwise let $s_1 = a_1a_2\ldots a_m$ ($|s_1| = m \geq 1$) and $s_2 = b_1b_2\ldots b_n$ ($|s_2| = n \geq 1$). Then $= s_1s_2 = a_1a_2\ldots a_mb_1b_2\ldots b_n$, and $|s_1s_2| = |s_1| + |s_2| = m + n$.

Example: Let A = {0, 1}. This is an alphabet with only two symbols. Given this, we have

1) The strings 00, 01, 10, and 11 are the only strings of length 2 over A.
2) If $s_1 = 011$ and $s_2 = 0110$, then $s_1s_2 = 0110110$ and $s_2s_1 = 0110011$.
3) If $s_1 = 011$ and $s_2 = 0110$, then $|s_1| = 3$, $|s_2| = 4$, and both $|s_1s_2|$ and $|s_2s_1|$ are 7.

The concatenation operation is **associative**, but (as we have just seen) **not commutative**. Because concatenation is associative, expressions such as $s_1s_2s_3$ have a unique interpretation, as $(s_1s_2)s_3 = s_1(s_2s_3)$. For example, let $s_1 = 11$ $s_2 = 2222$, and $s_3 = 333333$. Then

$s_1s_2 = 112222$ and $(s_1s_2)s_3 = (112222)333333$ $= 112222333333$, while
$s_2s_3 = 2222333333$ and $s_1(s_2s_3)$ $= 112222333333$.

We can reinforce the idea that concatenation is not commutative by noting that, for this last example, that $s_1s_2 = 112222$, while $s_2s_1 = 222211$.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

An Extra Remark on Commutativity

This author was once a student (hard to believe) in a modern algebra class. The professor was asked to name a simple operation that was not commutative. He was not ready for that question and could not name one. The answer is quite simple – **subtraction**. It is easily shown that subtraction is neither associative nor commutative.

Associativity test: $(4 - 2) - 1 = 2 - 1 = 1$
$4 - (2 - 1) = 4 - 1 = 3$

Commutativity test: $(4 - 2) = \ \ 2$
$(2 - 4) = -2$

Remember that one cannot prove an assertion by a simple example, but certainly can use a counterexample to disprove an assertion.

A Simple Lemma

Before continuing our discussion, we present the proof of a simple lemma. By itself, this lemma is of little consequence and unlikely to be used again in this course. It is the intent of this author to demonstrate the proof method, from which the reader can learn a few tricks.

Lemma: For any two strings s_1 and s_2 , $|s_1s_2| \geq |s_1|$, with equality if and only if $s_2 = \lambda$.
Proof: For any two strings s_1 and s_2 , $|s_1s_2| = |s_1| + |s_2|$. Now $|s_2|$ is a counting number, so it is not negative. From that remark, we immediately infer that $|s_1s_2| = |s_1| + |s_2| \geq |s_1|$.

Now suppose that $|s_1s_2| = |s_1| + |s_2| = |s_1|$. Then obviously $|s_2| = 0$ and $s_2 = \lambda$, the unique string of zero length. If $s_2 = \lambda$, then $|s_2| = 0$ and $|s_1s_2| = |s_1| + |s_2| = |s_1s_2| = |s_1| + 0 = |s_1|$.

Alphabets, Words, and Languages

We shall repeat a few of the above definitions and present them more formally. While struggling with these seemingly abstract ideas, the reader should remember that the terms as we use and precisely define them reflect the common usage learned in grammar school.

Words are the basic units used in definition of any language, although in some contexts the words are called "**tokens**". These words are built by concatenating a number of symbols from the finite set of symbols called the **alphabet** of the language. Following standard practice, we use the Greek symbol Σ to denote the alphabet. For standard English, we might say that Σ is the set of 26 standard letters, with $|\Sigma| = 26$.

Words are usually understood to be non-empty strings of symbols taken from the alphabet. Because of the significance of non-empty strings, we use the symbol Σ^+ to denote the set of all non-empty strings formed from the alphabet Σ. Note that $\lambda \notin \Sigma^+$, as λ is the empty string and Σ^+ is the set of non-empty strings. To allow the empty string to join the fun, we define another set (Sigma-Star) $\Sigma^* = \Sigma^+ \cup \{\lambda\}$.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

The reader is cautioned to remember that while λ is the empty string, the set {λ} is a set with one member – the empty string. Thus |λ| = 0, but |{λ}| = 1. The reader should also recall that the process $\mathbf{\Sigma}^* = \mathbf{\Sigma}^+ \cup \{\lambda\}$ is the standard way of adding a single element to a set; first we pop the element into a singleton set and then take the set union. The reader should also note the use of an overloaded operator in this paragraph. For a string ***s***, we use |***s***| to denote the number of symbols (characters) in ***s*** , while for a set S, we use |S| to denote the number of elements in the set. The definitions are consistent, just not identical.

Definition: For an alphabet **Σ**, we use the symbol $\mathbf{\Sigma}^*$ to denote the set of all strings over that alphabet, including the empty string.

Definition: For an alphabet **Σ**, we define a language on the alphabet as a subset of $\mathbf{\Sigma}^*$.
In set terminology, we say that $L \subseteq \mathbf{\Sigma}^*$. L is called a "**language on alphabet Σ**".

The reader will certainly note that the above definition is a bit broad for what we people normally consider to be a language. Indeed, according to this precise definition one could concatenate all words listed in an unabridged dictionary, producing a single word of about 600,000 letters in length that would by itself be a language. From the view of natural languages, a language should be mutually comprehensible with well established word meanings derived from a dictionary. This is not a requirement of a formal language.

Example: Hexadecimal numbers are base-16 numbers often found in discussions of computer architecture and organization. The "alphabet" for hexadecimal numbers is the hexadecimal digit set: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. This can be written as:

Hex_Digits = Decimal_Digits ∪ {A, B, C, D, E, F}

The following is a language defined over the alphabet Σ = {A, B, C, D, E, F}.

L1 = {A, BE, ADD, BAD, CAD, FAD, FED}

This author uses words such as found in L1 to add a bit of variety to his problems for homework and tests. The favorite use of the hexadecimal number "BAD", usually written in the problems as "BAD1" or "0BAD". If a memory location has contents "BAD1", it is usually a subtle hint that the correct answer lies elsewhere.

We should also note that the following language is a formal language over the same alphabet.

L2 = {AA, BB, CC, DD, EE, FF, AAA, AAB, AAC, AAD, AAE, AAF}.

Some formal languages bear no resemblance to any spoken language.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

Definition: Let Σ be an alphabet and let L and M be two languages on Σ. The **concatenation of languages** L and M, written LM, is the set LM = { xy | x ∈ L and y ∈ M}. Concatenation of a language with itself is also supported, according to the following recursive definition.

$$L^0 = \{\lambda\}$$
$$L^N = L^{N-1}L$$

Example: Let Σ = {0, 1, 2} and let L = {00, 01} and M = {0, 20, 21}.
Then $L\{\lambda\} = \{ xy \mid x \in L \text{ and } y \in \{\lambda\} \}$ think LM with M = {λ}
= { 00, 01 }

LM = { 000, 0020, 0021, 010, 0120, 0121 }

L^2 = { 0000, 0001, 0100, 0101 }

Definition: Let L be a language on alphabet Σ. The **Kleene Star of L**, written $\mathbf{L}^*$, is the infinite union of sets: $\{\lambda\} \cup L \cup L^2 \cup L^3 \cup L^4 \cup L^5 \cup L^6$ …., etc. The Kleene Star is obviously an infinite set.

Example: Let Σ = {0, 1, 2} and let L = {00, 01}. The Kleene Star of L is the infinite set beginning $L^* = \{\lambda\} \cup \{00, 01\} \cup \{ 0000, 0001, 0100, 0101 \}$ …, etc.

Example: Let Σ = {0, 1, 2} and let M = { 22 }. The Kleene Star of L is the infinite set beginning $M^* = \{\lambda\} \cup \{ 22 \} \cup \{ 2222 \} \cup \{ 222222 \} \cup \{ 22222222 \}$ …, etc. Here the language M^* is easily described as the set of all strings of even length containing only "2".

Grammars and Formal Languages

We first make an important distinction between natural languages (those spoken or once-spoken by humans) and formal languages. The difference focuses on the role of the grammar in the evolution of the language.

A **formal language** is completely specified by its grammar. Put another way, we say that the grammar **generates** the formal language. Any construct not supported by the grammar is not in the language and will remain outside the language until a new formal grammar is agreed upon and published. Often these published grammars are called "**standards**".

A **natural language** is one that has evolved, and often continues to evolve, by being spoken by humans. The role of grammar in this case is to codify the currently accepted usage. The grammar does not lead the language, but follows it and evolves as usage patterns change. One example of this evolution is the recent developments in the English grammar rule that a pronoun should agree with the noun to which it refers. The following sentences are each grammatically correct, but one is considered politically incorrect.

"Let each programmer examine the output of his program"
"Let each programmer examine the output of her program"

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

In standard (pre-1960's) grammar, the first form was to be used unless it was specifically known that the programmer was a female. This usage is now considered "sexist", as a result of which, we quite often will hear the sentence spoken as follows.

"Let each programmer examine the output of their program."

In strict grammar, the programmer is viewing the output of a program written by another group of programmers. Sentences, such as this, are often used because the pronoun "their" denotes neither male nor female and thus is thought to be non-sexist. Despite the feeble efforts of this old fuddy-duddy, the usage of grammar with the plurals "their" and "them" associated with single nouns will probably become the normative grammatical usage.

The following sections of these notes will focus on the use of a grammar to generate a formal language. In order to illustrate the discussion, we shall begin with a formal grammar that will generate valid sentences in the English language. More precisely, every sentence generated by this grammar will follow correct English syntax, in that the combination of words produced will have valid form. The sentences so produced might be nonsensical.

Before launching on formal grammars and formal languages, this author wishes to admit that the ambiguity in the English language can be the source of great literature and some raunchy humor. Those who like word-play should read the comedies of William Shakespeare, whom the author of these notes will not vilify by attempting to quote.

As for the humor in very bad taste, this author will mention a college classmate of his who claimed to be able to render any song in a vulgar mode without changing any of the words. Thus, he would sing the song line "I wonder who's kissing her now?" and then ask "What is a now?", pretending to consider it a part of the body.

Another ridiculous song rendition was to begin singing.

"I'm in the mood for love
Simply because you're near me.
Funny but, when you're near me"

After singing these lines, he would stop and ask what sadist would name his daughter "Funny Butt". I guess one had to be there and drunk to appreciate such humor.

Those who think that English grammar is easily described should examine these sentences.

"Time flies like an arrow."
"Fruit flies like a banana."

The truly perverse will render the first sentence as a command to take a watch and time the flight of flies in the same way that one times the flight of an arrow. This is really silly.

There are also some valid sentences that show that English is not context-free (a term to be defined later). For example consider a sentence beginning with the words "The old man". In a context-free language we would immediately know what to make of this. But now consider the two following sentences, each of which begins with the same three words.

"The old man likes to fish."
"The old man the boats."

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

Here is the set of rules describe a grammar that produces a subset of English.

1. A **sentence** comprises a **noun phrase** followed by a **verb phrase**.
2. A **noun phrase** is either
   a) an **article** followed by an **adjective** followed by a **noun**, or
   b) an **article** followed by a **noun**.
3. A **verb phrase** is either
   a) a **verb** followed by an **adverb**, or
   b) a **verb**.
4. **article** = {a, the} -- this is a list of "terminal symbols"
5. **adjective** = {silly, clever}
6. **noun** = {frog, cow, lawyer}
7. **verb** = {writes, argues, eats}
8. **adverb** = {well, convincingly}

Here is one sentence validly generated. Note that it is nonsensical.

sentence
noun phrase verb phrase
article adjective noun verb adverb
the **adjective noun verb adverb**
the silly **noun verb adverb**
the silly cow **verb adverb**
the silly cow writes **adverb**
the silly cow writes convincingly

Note that the generation of the sentence proceeds by replacing a non-terminal symbol, denoted in bold font, by another non-terminal symbol or by a terminal symbol.

Just to show that the above grammar is not restrained to producing nonsense, we generate another valid sentence.

sentence
noun phrase verb phrase
article adjective noun verb adverb
the **adjective noun verb adverb**
the clever **noun verb adverb**
the clever lawyer **verb adverb**
the clever lawyer argues **adverb**
the clever lawyer argues convincingly

It is easily shown that the number of sentences validly generated by this grammar is computed by 2•3•3•3•3 = 162. Most are nonsensical; a few are not.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

The reader should note that the process of generating valid sentences in a language differs from the process of recognizing the sentence as valid. When in middle school, this author daily had to "diagram sentences", a process of applying a mechanical construct to a sequence of words to determine if that sequence was consistent with accepted English grammar.

For those who cannot remember the joys of High-School English class, here are the above two sentences diagrammed, using the notation taught to this author.

What is to be noted about these diagrams is that they do not generate the sentences, but verify the structure of the sentences. They are "acceptors" or "recognizers", not "generators".

Four Types of Grammars for Formal Languages

Here we follow the work of the great linguist Noam Chomsky (who is not related to Noam Chimpsky, the chimpanzee that can use sign language) who taught in the Department of Foreign Languages and Linguistics at the Massachusetts Institute of Technology. We begin with a description of the methods used to describe a grammar and then quickly describe four grammars, each more restrictive than the previous one.

Most standard texts on formal language theory use the terms "alphabet" (defined above) and "vocabulary" (not yet defined) interchangeably. It will facilitate our understanding to assign differing definitions to the two terms. We begin with this set of definitions.

Definitions:

1. A **vocabulary** V is a finite, non-empty, set of elements called **symbols**.
2. A **sentence** over V is a sequence of a finite number of elements of V.
3. The **empty sentence** λ (also called the empty string) is the sentence containing no symbols.
4. The set of all sentences over V is denoted by V*.
5. A language over V is a subset of V*.

Example:

In the above grammar producing a subset of English, the vocabulary is given by the set
{ **sentence**, **noun phrase, verb phrase, article, adjective, noun, verb, adverb,**
a, the, silly, clever, frog, cow, lawyer, writes, argues, eats, well, convincingly}.

Note that each of the eight symbols in the first line of the set listing is written in bold font. The twelve symbols in the second line are called **terminals** in the grammar, because they cannot be replaced by other symbols. Other symbols in the vocabulary, those that can be replaced by other symbols, are called **non-terminals**. The set of terminals is denoted by T, the set of non-terminals is denoted by N; thus $V = N \cup T$.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

There is one distinguished non-terminal symbol, called the **start symbol** and denoted S, that is the non-terminal from which we always begin our grammatical productions. In this grammar, the start symbol is **sentence**. We shall now examine each of the rules of this formal grammar (called "**productions**") and classify the symbols by how they appear. This analysis will function mostly to justify the results already stated above.

1. A **sentence** comprises a **noun phrase** followed by a **verb phrase**.

   Here the symbol **sentence** is clearly a non-terminal as it is replaced by the symbol **noun phrase** followed by the symbol **verb phrase**. This is the first rule in the grammar, so the symbol is the **start symbol**.

2. A **noun phrase** is either
   a) an **article** followed by an **adjective** followed by a **noun**, or
   b) an **article** followed by a **noun**.

   Here the symbol **noun phrase** is clearly a non-terminal as it can be replaced by either one or two other symbols.

3. A **verb phrase** is either
   a) a **verb** followed by an **adverb**, or
   b) a **verb**.

   The symbol **verb phrase** is also clearly a non-terminal.

4. **article** = {a, the} -- this is a list of "terminal symbols"
5. **adjective** = {silly, clever}
6. **noun** = {frog, cow, lawyer}
7. **verb** = {writes, argues, eats}
8. **adverb** = {well, convincingly}

   The symbols **article, adjective, noun, verb, adverb** are clearly non-terminals.

What we have left over are the symbols for which no substitution rules are specified. This is the set of terminal symbols, denoted by T. Thus we have the two sets,

N = { **sentence**, **noun phrase, verb phrase, article, adjective, noun, verb, adverb** }
T = { a, the, silly, clever, frog, cow, lawyer, writes, argues, eats, well, convincingly }

Again, this approach is not so different from standard English grammar in which we speak of grammatical terms, such as nouns, adjectives, adverbs, and verbs. We are not even bothered by the fact that the words "adjective", "adverb", and "verb" are all nouns. In terms of formal grammar, we instinctively know the difference between terminals and non-terminals.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

Productions and Grammars

We now introduce the more formal notation for **productions**, which are the rules that specify how to replace one sequence of symbols from V* by another sequence of symbols. We shall denote a production with notation such as $X \to Z$, specifying that X can be replaced by Z. We shall extend the usual notation by notation such as $X \to Y \mid Z$, which indicates that X can be replaced either by Y or by Z. Thus the following set of productions.

$$X \to Y$$
$$X \to Z$$

is equivalently written as

$$X \to Y \mid Z$$

Each production has a left side and a right side. In the above examples, the non-terminal X is on the left side. The right side of the above examples are Y, Z, and Y | Z respectively.

We now use this notation to present the sample grammar previously discussed in this chapter.

1. **sentence → noun_phrase verb_phrase**
2. **noun_phrase → article adjective noun | article noun**
3. **verb_phrase → verb adverb | verb**
4. **article** → a | the
5. **adjective** → silly | clever
6. **noun** → frog | cow | lawyer
7. **verb** → writes | argues | eats
8. **adverb** → well | convincingly

Definition: A **phase-structure grammar** G = (V, T, S, P) consists of a vocabulary V; a subset $T \subset V$, consisting of terminal elements; by implication a set $N = V - T$, called the non-terminal elements; a start symbol $S \in N$; and a set of productions P. Every production must include at least one non-terminal on its left side.

The clear implication of the above is that terminal symbols may appear on the left side of a production. In our sample grammar we might change rule 7 to three distinct rules.

7a frog **verb** → frog eats
7b cow **verb** → cow eats
7c lawyer **verb** → lawyer writes | lawyer argues | lawyer eats

Note that this version of the grammar, with the more restrictive three productions, excludes the sentence "the silly cow writes convincingly" but allows the sentence "the silly cow eats convincingly", while still allowing "the clever lawyer argues convincingly".

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

Definition: Consider a production of the form X → Y. The **length** of X is the number of symbols in X. Similarly the length of Y is the number of symbols in Y. We shall not extend the concept of length to the right side of productions such as X → Y | Z.

Given the above definition, we may now define Chomsky's four grammars.

A **type 0 grammar** (**phase-structure grammar**) is a grammar with no restrictions on its productions other than the previously mentioned rule that the left side contain at least one non-terminal symbol.

A **type 1 grammar (context-sensitive grammar)** is a grammar with the added restriction that productions must either be of the form X → λ, or X → Y, where the length of Y is not less than the length of X.

A **type 2 grammar (context-free grammar)** is a type 1 grammar with the additional restriction that the left side of a production have a single non-terminal symbol.

A **type 3 grammar (regular grammar)** is a type 2 grammar with the additional restriction that productions be only of one of these three forms

$$X \rightarrow xY$$
$$X \rightarrow x$$
$$S \rightarrow \lambda$$

where X and Y are non-terminals and *x* is a terminal.

Put in words, the only productions allowed by a regular grammar are
1. A single non-terminal symbol produces a terminal symbol followed by a non-terminal.
2. A single non-terminal symbol produces a single terminal symbol.
3. The start symbol produces the empty sentence.

Example: We first consider the language described by the set $\{0^n1^n2^n \mid n \geq 0\}$. Since this notation may not be familiar to the reader, we first explain its meaning.

Let $n = 0$. Then $0^n1^n2^n = \lambda$, the empty sentence with no 0's, no 1's, and no 2's.
Let $n = 1$. Then $0^n1^n2^n = 012$.
Let $n = 2$. Then $0^n1^n2^n = 001122$.
Let $n = 3$. Then $0^n1^n2^n = 000111222$.

So the set $\{0^n1^n2^n \mid n \geq 0\} = \{\lambda, 012, 001122, 000111222, 000011112222, \ldots\}$.

Similarly the set $\{0^n1^n \mid n \geq 0\} = \{\lambda, 01, 0011, 000111, 00001111, \ldots\}$.

However, the set $\{0^m1^n \mid m \geq 0,\ n \geq 0\}$ is more complicated, beginning with $\{0^m1^n \mid m \geq 0,\ n \geq 0\} = \{\lambda, 0, 1, 00, 01, 11, 000, 001, 011, 111, \ldots\}$. This set is just a bunch of 0's followed by a bunch of 1's.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

Here is the grammar that generates the language $\{0^n1^n2^n \mid n \geq 0\}$.
It is G = (V, T, S, P), with

V = {0, 1, 2, S, A, B} -- the set of symbols or vocabulary
T = {0, 1, 2} -- the terminal symbols
N = {S, A, B} -- the non-terminal symbols: N = V – T
Productions: $S \to \lambda \mid 0SAB$
$BA \to AB$
$0A \to 01$
$1A \to 11$
$1B \to 12$
$2B \to 22$.

There are methods for constructing this set of productions from the description of the language, but we leave that to the compiler writers. The student may want to verify that the grammar generates the given language, but it is acceptable to take the author's word on that.

The point is that this is a type 1 (context sensitive) grammar. Note that the interpretation of the non-terminal symbols A and B depends on the terminal symbol preceding each, thus the production is sensitive to the context.

Example: Consider the language $\{0^n1^n \mid n \geq 0\}$. The grammar that generates this language is G = (V, T, S, P) with

V = {0, 1, S}
T = {0, 1}
N = {S}
Productions: $S \to \lambda \mid 0S1$.

It should be obvious that this grammar generates the language, as we increase the size of a string by adding a 0 to its start and a 1 to its end. The number of 0's always equals the number of 1's and every zero precedes the first 1.

The form of the production identifies this as a type 2 (context-free) grammar.

Example: Consider the language $\{0^m1^n \mid m \geq 0, \; n \geq 0\}$. Just to be contrary, we shall show two grammars that generate this language. This will show that two grammars can generate the same language. We see also that these two grammars are of distinct types.

The first grammar is G1 = (V1, T, S, P1) with

V1 = {0, 1, S}
T = {0, 1} -- same set as G2
N1 = {S} -- the non-terminal symbols in this grammar
Productions: $S \to \lambda \mid 0S \mid S1$.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

G1 clearly produces this language, since using the production S → 0S $m \geq 0$ times puts m 0's at the beginning of the string and using the production S → S1 $n \geq 0$ times puts n 1's at the end of the string. The reader is invited to verify this claim, as the verification is easy. Examination of the productions shows this to be a type 2 (context-free)grammar.

The second grammar to generate this language is G2 = (V2, T, S, P2) with

V2 = {0, 1, A, S}	
T = {0, 1}	-- same set as G1
N2 = {A, S}	-- the non-terminal symbols for this grammar
Productions:	S → λ | 0S | 1A | 1
	A → 1A | 1.

The reader should recall that the above set of productions is shorthand for these six.

S → λ
S → 0S
S → 1A
S → 1
A → 1A
A → 1

When the production rules are written in this fashion, it is easy to see that the grammar is regular. So what do we say about the language?

1. It is generated by a context-free grammar G1.
2. It is generated by a regular grammar G2.

Since there exists a regular grammar that generates this language, we consider it to be a **regular language**, also called a **regular set**. As all regular languages are also context-free, it should come as no surprise that there exists a context-free grammar that generates this one.

Backus-Naur Form

This notation for describing a grammar was developed by John Backus of IBM, an inventor of the FORTRAN language, and first called **Backus Normal Form**. After significant contributions by Peter Naur, the notation was renamed Backus-Naur Form, the name it has today. It is also called BNF. It is used to describe type 2 (context-free) grammars. Since all type 3 (regular) grammars are also type 2 grammars, BNF can describe regular grammars.

The notation for productions in BNF is slightly different. All non-terminal symbols are enclosed in brackets, so that the symbol <A> is a non-terminal, while the symbol A is a terminal. The notation "→" used for a production is replaced by "::=". This comes from an earlier ALGOL-like notation for assignment and equality.

X = Y	An equality statement, is X equal to Y.
X := Y	An assignment statement, X is set to Y
X ::=Y	X is defined to be Y.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

In our previous notation for productions, we have already borrowed from BNF. Consider the productions written in these notes in the form $A \rightarrow Aa \mid a \mid AB$, with A and B non-terminals. Strictly speaking, this set should be written as three distinct productions.

$$A \rightarrow Aa$$
$$A \rightarrow a$$
$$A \rightarrow AB$$

The convention in BNF is to write these as one production, unless that is hard to read. So the standard BNF statement of this trio of productions would be.

<A> ::= <A>*a* | *a* | <A><B>

The brackets around the upper case A and B indicate that those two are non-terminals. In BNF, it is best to view the non-terminals as having brackets, so that the non-terminals in the above are really <A> and <B>. The only terminal in this production is the symbol *a*. The use of *italics* in displaying the terminal symbol has nothing to do with BNF, but is just a convention often used by this author when writing MS-Word documents.

Here is the grammar for the pseudo-English, written in BNF.

1. <sentence> ::= <noun phrase> <verb phrase>
2. <noun phrase> ::= <article> <adjective> <noun> | <article> <noun>
3. <verb phrase> ::= <verb> <adverb> | <verb>
4. <article> ::= a | the
5. <adjective> ::= silly | clever
6. <noun> ::= frog | cow | lawyer
7. <verb> ::= writes | argues | eats
8. <adverb> ::= well | convincingly

For more examples of the use of BNF, we turn to Chapter 17 (Grammar Summary) of The Annotated C++ Reference Manual by Margaret A. Ellis and Bjarne Stroustrup, published by Addison-Wesley in 1990. It has ISBN 0 – 201 – 51459 – 1. Line breaks have been introduced into these examples to keep individual productions on one line.

<expression> ::= <assignment expression> | <expression> , <assignment expression>

<assignment expression>
  ::= <conditional expression>
  | <unary expression> <assignment operator><assignment expression>

<assignment operator> :: = = | *= | /= | %= | += | −= | >>= | <<= | &= | ^= | | =

<conditional expression>
  ::= <logical or expression>
  | <logical or expression> ? <expression> : <conditional expression>

Rather than continue with this for several pages, let's examine and translate each.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

<expression> ::= <assignment expression> | <expression> , <assignment expression>

This states that an expression is defined to be either an expression or the sequence of an expression followed by a comma followed by an assignment expression. Note that the comma in the above BNF production is a terminal symbol in the grammar and is to be read literally. This allows expressions such as the following

x = 0 , y = 1

<assignment expression>
::= <conditional expression>
| <unary expression> <assignment operator><assignment expression>

Here we state that an assignment expression is either a conditional expression or a unary expression followed by an assignment operator followed by an assignment expression.

<assignment operator> ::= = | *= | /= | %= | += | −= | >>= | <<= | &= | ^= | |=

Here we list the valid assignment expressions. There are 11 of them. Some, such as "−=" and " | =" are hard to display in the font used in this document.

<conditional expression>
::= <logical or expression>
| <logical or expression> ? <expression> : <conditional expression>

Here we begin to define the conditional expressions. It may seem strange that the non-terminal <logical or expression> is the only specific conditional expression listed, but that is the document's way of specifying the hierarchy of logical operations. Note again that the second part of this production has the non terminals "?" and ":", used as a part of the conditional expression operator "?:" that is unique to C and C++.

We close this section on BNF with the standard example of the grammar for signed integers.

<signed integer>	::= <sign><integer>
<sign>	::= + | −
<integer>	::= <digit><integer>
<digit>	::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

Grammars and Automata

Now we have the following facts.

1) Regular languages are generated by regular grammars.
2) Context-free languages are generated by context-free grammars.
3) Context-sensitive languages are generated by context-sensitive grammars.
4) Phrase-structure languages are generated by phrase-structure grammars.

More strictly speaking, we should state that only regular languages can be generated by regular grammars, that context-free grammars can generate either context-free languages or regular languages (if properly constrained), etc. However, the association of each language class with grammar type is standard and not to be trifled with.

We now ask for an association of languages with the types of automata that recognize the languages. We might as well list the associations and then give the definitions.

1) Regular languages are recognized by finite state automata, such as we have studied.
2) Context-free languages are recognized by **pushdown automata**.
3) Context-sensitive languages are recognized by **linearly bounded automata**.
4) Phrase-structure languages are recognized by **Turing machines**.

The reader will note immediately that the regular languages are the most constrained of all of the four language types, in that the grammar that generates a regular language has the most constraints placed upon it. It stands to reason that the most constrained language is recognized by the simplest automaton. We need more powerful automata to recognize the less constrained languages.

The main problem with a finite state automaton is that, by definition, it has a finite amount of memory. A **pushdown automaton** is a finite state automaton to which a standard push-pop stack has been added. The stack, in effect, adds a particular type of infinite memory to the finite automaton, which now includes operations to push symbols onto the stack and pop them for comparison with input symbols.

We give an example of a language that can be recognized by a pushdown automaton that cannot be recognized by a finite state automaton. This is the language of palindromes with distinct middle symbols. For example, let the set of terminals in the language be {0, 1, #}, with the symbol "#" being the distinct middle symbol. Let W be a sequence of symbols from the set {0, 1}. A palindrome would be of the form $W\#W^R$, where W^R is W written in reverse. A sample palindrome would be "1010#0101".

The "program" for the pushdown automaton to recognize this palindrome is as follows.

1) Read a symbol. If not "#", push the symbol onto the stack and continue.
2) If the symbol is "#" go to the next phase.
3) After reading "#", continue reading symbols. For each symbol read, pop a symbol from the stack and compare. If the two are not equal, stop and say "No".
4) If the stack and input are both empty, stop and say "Yes".

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

A linearly bounded automaton is a restricted Turing machine, so we define the Turing machine next and then define the linearly bounded automaton.

A **Turing machine** is also a finite state automaton to which additional memory has been added. The memory for a Turing machine can be considered as infinite, possibly in both directions. The best way to think of such an infinite tape is to consider it as having a number of cells, each containing a single symbol. Each cell can be identified by an integer. Recalling that the set of integers is infinite is both directions, so too is our theoretical tape.

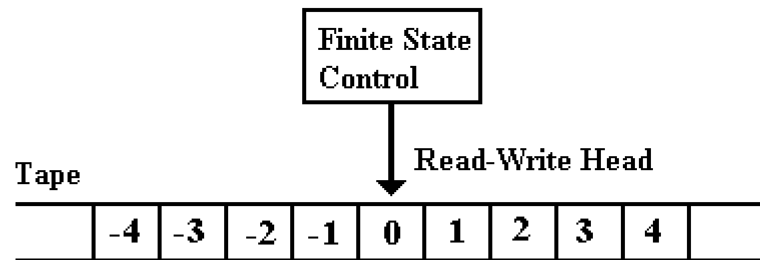

Figure: A Turing Machine at the Start State

The specification of the Turing machine includes the following.

1) A finite set Γ of **tape symbols**, including a subset $\Sigma \subset \Gamma$ of **input symbols** and a distinguished **blank symbol**, which we can denote ◊.
2) A finite set Q of **states** for the finite automaton that serves as the control of the Turing machine. This set must include three distinguished states
   the start state, denoted q_0, and
   a halt state, q_Y, called the **accept state**, and
   a halt state q_N, called the **reject state**.
3) A transition function that causes the finite state controller to move through the states, possibly reading from the tape and writing to the tape as it goes.

The operation of a Turing machine is straightforward. At the start of the process, the finite state controller is in state q_0, and the input X is on squares 1 through |X| of the tape, where |X| is the length of the input X, which is a string $X \in \Sigma^*$. The Turing machine begins operating by moving to square 1 and reading the first symbol from the string X. After that, it proceeds as directed by the transition function. There are three possible outcomes.

1) It arrives at the accept state q_Y, at which time it halts and accepts the string.
2) It arrives at the reject state q_N, at which time it halts and rejects the string.
3) It loops, which is to say it never arrives at a halt state but continues to process indefinitely. This leads indirectly to the halting problem, defined below.

The **halting problem** is one of the few provably undecidable problems in computer science. This is to determine whether or not a Turing machine will halt after processing an arbitrary string of input. The best we can do it let the Turing machine process the input and wait a while, after which it either has halted or has not halted. If it has not halted, all we can say is that it might halt after some more operations, but we cannot be sure.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.

While the input to a Turing machine is a finite sequence of symbols, the string X, stored in squares 1 through |X| of the tape, the tape itself is infinite. The Turing machine can write additional symbols in any other part of the tape and return to read them later as directed by the transition function.

A **linearly bounded automaton** in a Turing machine with the restriction that the only part of the tape that can be accessed is the finite part of the tape originally occupied by the input.

Let's now compare the three automata just defined.

1) A pushdown automaton has an infinite memory, but it is a push-down stack.
2) A linearly bounded automaton has a random access memory, but it is finite.
3) A Turing machine has an infinite random access memory.

The Church-Turing Thesis

We have just claimed that any language can be recognized by a Turing machine. This is a conjecture put forward by Alonzo Church and Alan Turing in the late 1930's. It basically claims that anything computable by a standard algorithm (with conventional definition) is computable by a Turing machine.

We might ask why this is not called the "Church-Turing Theorem". The answer is that it cannot be proven. We might then ask why it cannot be proven. The simple answer is that all of our intuitive concepts of algorithms, as well as the formal definitions, are already equivalent to statements in terms of Turing machines.

In order to prove the Church-Turing thesis, we would have to arrive at a formal definition of an algorithm that is independent of the Turing machine model. We have yet to be able to arrive at such an independent definition, so we cannot even begin a proof.

Copyright © 2005 by Edward L. Bosworth. All rights reserved.