Chapter 2

Language

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean - nothing more nor less."

"The question is," said Alice, "whether you can make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master that's all."

Lewis Carroll, *Through the Looking Glass*

Topics: what is language, what a language is made of, what properties a language should have for programming computers, how to describe languages

The most powerful tool we have for communication is language. This is true whether we are considering communication between two humans, communication between a human programmer and a computer, or communication between multiple computers. This chapter considers what a language is, how language works, and introduces the techniques we will use to define languages.

2.1 Languages and Infinity

A *language* is a set of surface forms, *s*, meanings, *m*, and a mapping between the surface forms in *s* and their associated meanings[1]. In the earliest human languages, the surface forms were sounds. But, the surface forms can be anything that can be perceived by the communicating parties. We will focus on languages where the surface forms are text. A *natural language* is a language spoken by humans such as English. Natural languages are very complex since they have evolved over many thousands years of individual and cultural interaction. We will be primarily concerned with designed languages that are created by humans for some purpose (in particular, languages created for expressing computer programs).

A simple communication system could be described by just listing a table of surface forms and their associated meanings. For example, this table describes a communication system between traffic lights and drivers:

Surface Form	Meaning
Green Light	Go
Yellow Light	Caution
Red Light	Stop

Communication systems involving humans are notoriously imprecise and subjective. A driver and a police officer may disagree on the actual meaning of the *Yellow Light* symbol, and may even disagree on which symbol is being transmitted by the traffic light at a particular time. Communication systems for computers demand precision: we want to understand what our programs will do, so it is important that every step they make is understood precisely and unambiguously.

The method of defining a communication system by listing a table of <*Symbol*, *Meaning*> pairs can work adequately only for trivial communication systems. The number of possible meanings that can be expressed is limited by the number of entries in the table. Thus, it is impossible to express any *new* meaning using the communication system: all meanings must already be listed in the table!

A real language must be able to express *infinitely* many different meanings. This means it must provide infinitely many surface forms, and a way of inferring the

[1] Thanks to Charles Yang for this definition.

meaning of each possible surface form. No finite representation such as a printed table can contain all the surface forms and meanings in an infinite language.

One way humans can generate infinitely large sets is to use repeating patterns. For example, most humans would recognize the notation:

1, 2, 3, . . .

as the set of all natural numbers. We interpret the ". . . " as meaning keep doing the same thing for ever. In this case, it means keep adding one to the preceding number. This technique might be sufficient to describe some languages with infinitely many meanings. For example, this table defines an infinite language:

Surface Form	Meaning
"I run today."	Today, I run.
"I run the day after today."	One day after today, I run.
"I run the day after the day after today."	Two days after today, I run.
"I run the day after the day after the day after today."	Three days after today, run.
. . .	. . .

Although there are infinite languages we can describe in this way, it is wholly unsatisfactory.[2] The set of surface forms must be produced by simple repetition. Although we can express new meanings using this type of language (for example, we can always add one more "the day after" to the longest previously produced surface form), the new surface forms and associated meanings are very similar to previously known ones.

2.2 Language Components

A language is a set of surface forms and associated meanings, but the number of surface forms in any powerful language is infinite. Hence, we cannot define a language by listing all the surface forms and their meanings. Instead, we need to

[2]Languages that can be defined this way are known as *regular languages*. We will define this more precisely in Chapter **??**, and see that they are actually a bit more interesting than it seems here.

find ways of describing languages that allow us to describe an infinitely large set of surface forms and meanings with a compact notation. The approach we will use is to define a language by defining a set of rules that produce all strings in the language (and no strings that are not in the language).

A language is composed of:

- *primitives* — the smallest units of meaning. A primitive cannot be broken into smaller parts that have relevant meanings.
- *means of combination* — rules for building new language elements by combining simpler ones.

In English, the primitives are the smallest meaningful units, known as *morphemes*. The means of combination are rules for building words from morphemes, and for building phrases and sentences from words.

Since we have rules for producing new words not all words are primitives. For example, we can create a new word by adding *anti-* in front of an existing word. The meaning of the new word is (approximately) "against the meaning of the original word".

For example, *freeze* means to pass from a liquid state to a solid state; *antifreeze* is a substance designed to prevent freezing. An English speaker who knew the meaning of *freeze* and *anti-* could roughly guess the meaning of *antifreeze* even if the word is unfamiliar.

Note that the primitives are defined as the smallest units of *meaning*, not based on the surface forms. We can break *anti-* into two syllables, or four letters, but those sub-components do not have meanings related to the meaning of the morpheme.

This property of English means anyone can invent a new word, and use it in communication in ways that will probably be understood by listeners who have never heard this word. There can be no longest English word, since for whatever word you claim to be the longest, I can create a longer one (for example, but adding *anti-* to the beginning of your word).

Means of Abstraction. In addition to primitives and means of combination, powerful languages have an additional type of component that enables economic communication: *means of abstraction*. Means of abstraction allow us to give a

simple name to a complex entity. In English, the means of abstraction are *pronouns* like "she", "it", and "they". The meaning of a pronoun depends on the context in which it is used. It abstracts a complex meaning with a simple word. For example, the *it* in the previous sentence abstracts "the meaning of a pronoun", but the *it* in the setence before that one abstracts "a pronoun". In natural languages, means of abstraction tend to be awkward (English has *she* and *he*, but no gender-neutral pronoun for abstracting a person), and confusing (it is often unclear what a particular *it* is abstracting). Languages for programming computers need to have powerful and clear means of abstraction.

Next, we introduce two different ways of defining languages. We will limit our discussion to textual languages. This means the surface forms will be sequences of characters. We will refer to a sequence of zero or more characters as a *string*. Hence, our goal in defining the surface forms of a textual language is to define the set of strings in the language. The problem of associating meanings with those strings is much more difficult; we consider it in various ways in later chapters.

Exercise 2.1. Merriam-Webster's word for the year for 2006 was *truthiness*, a word invented and popularized by Stephen Colbert.

a. ($\diamond$) Invent a new English word by combining common morphemes.

b. ($\star$) Get someone else to use the word you invented.

c. ($\star\star\star\star$) Get Merriam-Webster to add your word to their dictionary.

Exercise 2.2. According to the Guinness Book of World Records, the longest word in the English language is *floccinaucinihilipilification*, meaning "The act or habit of describing or regarding something as worthless".

a. ($\diamond$) Break *floccinaucinihilipilification* into its morphemes. Show that a speaker familiar with the morphemes could understand the word.

b. ($\diamond$) Prove Guinness wrong by demonstrating a longer English word. An English speaker should be able to deduce the meaning of your word.

2.3 Post Production Systems

As for any claims I might make perhaps the best I can say is that I would have *proved* Gödel's Theorem in 1921 — had I been Gödel.
Emil Post, from postcard to Gödel, October 29, 1938. (Quoted in Liesbeth De Mol, *Closing the Circle: An Analysis of Emil Post's Early Work*, Association for Symbolic Logic, 2006.)

Production systems were invented by Emil Post, an American logician in the 1920's. A *Post production system* consists of a set of production rules. Each production rule consists of a pattern to match on the left side, and a replacement on the right side. From an initial string, rules that match are applied to produce new strings. For example, Douglas Hofstadter describes the following Post production system known as the *MIU-system* in *Gödel, Escher, Bach* (each rule is described first formally, and the quoted text below is the description from *GEB*):

Rule I: $x\mathsf{I} ::\Rightarrow x\mathsf{IU}$
"If you possess a string whose last letter is I, you can add on a U at the end."

Rule II: $\mathsf{M}x ::\Rightarrow \mathsf{M}xx$
"Suppose you have Mx. Then you may add Mxx to your collection."

Rule III: $x\mathsf{III}y ::\Rightarrow x\mathsf{U}y$
"If III occurs in one of the strings in your collection, you may make a new string with U in place of III."

Rule IV: $x\mathsf{UU}y ::\Rightarrow xy$
"If UU occurs inside one of your strings, you can drop it."

The rules use the variables x and y to match any sequence of symbols. On the left side of a rule, x means match a sequence of zero or more symbols. On the right side of a rule, x means produce whatever x matched on the left side of the rule. We refer to this process as *binding*. The variable x is initially unbound — it may match any sequence of symbols. Once it is matched, though, it is *bound* and refers to the sequence of symbols that were matched.

For example, consider applying Rule II to MUM. To match Rule II, the first M matches the M at the beginning of the left side of the rule. After that the rule uses x, which is currently unbound. We can bind x to UM to match the rule. The right side of the rule produces Mxx. Since x is bound to UM, the result is MUMUM.

Given these four rules, we can start from a given string and apply the rules to produce new strings. For example, starting from MI we can apply the rules to produce MUIUIU:

1. MI Initial String

2.	MII	Apply Rule II with x bound to I
3.	MIIII	Apply Rule II with x bound to II
4.	MIIIIIIII	Apply Rule II with x bound to IIII
5.	MUIIIII	Apply Rule III with x bound to M and y bound to IIIII
6.	MUIIIIIU	Apply Rule I with x bound to MUIIIII
7.	MUIUIU	Apply Rule III with x bound to MUI and y bound to IU

Note that at some steps we have many choices about which rule to apply, and what bindings to use when we apply the rule. For example, at step 5 we could have instead bound x to MUII and y to the empty string to produce MUIIU.

Exercise 2.3.

a. ($\diamond$) Using the *MIU-system*, show how M can be derived starting from MI?

b. ($\diamond$) Using the *MIU-system*, how many different strings can be derived starting from UMI?

c. ($\diamond$) Using the *MIU-system*, how many different strings can be derived starting from MI?

d. ($\star$) (Based on GEB) Using the *MIU-system*, is it possible to produce MU starting from MI?

Exercise 2.4. ($\star$) Devise a Post production system that can produce all the surface forms in the { "I run today.", "I run the day after today.", "I run the day after the day after today.", . . . } language.

2.4 Replacement Grammars

Although Post production systems are powerful enough to generate complex languages[3], they are more awkward to use than we would like. In particular, applying

[3]In Chapter **??**, we will consider more carefully the set of languages that can be defined by different systems.

a rule requires making decisions about binding variables in the left side of a rule. This makes it hard to reason about the strings in a language, and hard to determine if a given string is in the language or not (see Exercise 2.3).

In this section, we will describe a similar, but simpler, way of defining a language known as a grammar[4]. This is the most common way languages are defined by computer scientists today, and the way we will use for the rest of this book.

A *grammar* is set of rules for generating all strings in the language. The grammars we will use are a simple replacement grammar known as *Backus-Naur Form* (BNF). Rules are of the form:

$$\textit{symbol} \quad ::\Rightarrow \quad \textit{replacement}$$

These rules are similar to Post production rules, except that the left side of a rule is always a single symbol. Whenever we can match the left side of a rule, we can replace it with what appears on the right side of the matching rule.

We call the symbol on the left side of a rule a *nonterminal*, since it cannot appear in the final string. The right side of a rule contains one or more symbols. These symbols may include nonterminals, which will be replaced using replacement rules before generating the final string. They may also be *terminals*, which are symbols that never appear as the left side of a rule. When we describe grammars, we use *italics* to represent nonterminal symbols, and **bold** to represent terminal symbols. Once a terminal is reached, no more replacements can be done on it.

We can generate string in the language described by a replacement grammar by starting from a designated start symbol (e.g., *sentence*), and at each step selecting a nonterminal in the working string, and replacing it with the right side of a replacement rule whose left side matches the nonterminal. Unlike Post production systems, there are no variables to bind in BNF grammar rules. We simply look for a nonterminal that matches the left side of a rule.

Here is an example BNF grammar:

[4]You are probably already somewhat familiar with grammars from your time in what was previously known as "grammar school"!

1.	*Sentence*	::⇒	*Noun Verb*
2.	*Noun*	::⇒	**Alice**
3.	*Noun*	::⇒	**Bob**
4.	*Verb*	::⇒	**jumps**
5.	*Verb*	::⇒	**runs**

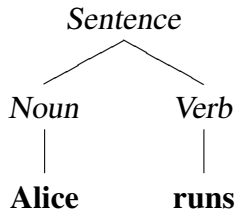
Starting from *Sentence*, we can generate four different sentences using the replacement rules: "Alice jumps", "Bob jumps", "Alice runs", and "Alice jumps".

A derivation shows how a grammar generates a given string. Here is the derivation of "Alice runs":

Sentence ::⇒*Noun Verb*	using Rule 1
::⇒**Alice** *Verb*	replacing *Noun* using Rule 2
::⇒**Alice runs**	replacing *Verb* using Rule 5

We can represent a grammar derivation as a tree, where the root of the tree is the starting nonterminal (*sentence* in this case), and the leaves of the tree are the terminals that form the derived sentence. Such a tree is known as a *parse tree*.

Here is the parse tree for the derivation of "Alice runs":



From this example, we can see that BNF notation offers some compression over just listing the string in the language, since a grammar can have multiple replacement rules for each nonterminal. Adding another rule like,

6.	*Noun*	::⇒	**Colleen**

to the grammar would add two new strings to the language.

Recursive Grammars. The real power of BNF as a compact notation for describing languages, though, comes once we start adding recursive rules to our grammar. A recursive grammar includes a rule where the nonterminal on the left side of the rule can be generated from its own replacement on the right side of the rule. Suppose we add the rule,

7.	*Sentence*	::⇒	*Sentence* **and** *Sentence*

to our example grammar. Now, how many sentences can we generate?

Infinitely many! For example, we can generate "Alice runs and Bob jumps" and "Alice runs and Bob jumps and Colleen runs". We can also generate "Alice runs and Alice runs and Alice runs and Alice runs", with as many repetitions of "Alice runs" as we want. This is very powerful: it means a compact grammar can be used to define an infinitely large language.

Example 2.1: Whole Numbers. Here is a grammar that defines the language of the whole numbers (0, 1, . . .):

Number	::⇒	*Digit MoreDigits*
MoreDigits	::⇒	
MoreDigits	::⇒	*Number*
Digit	::⇒	**0**
Digit	::⇒	**1**
Digit	::⇒	**2**
Digit	::⇒	**3**
Digit	::⇒	**4**
Digit	::⇒	**5**
Digit	::⇒	**6**
Digit	::⇒	**7**
Digit	::⇒	**8**
Digit	::⇒	**9**

Note that the second rule says we can replace *MoreDigits* with nothing. This is sometimes written as ϵ to make it clear that the replacement is empty:

MoreDigits ::⇒ ϵ

This is a very important rule in the grammar—without it *no* strings could be generated; with it *infinitely* many strings can be generated. The key is that we can only produce a string when all nonterminals in the string have been replaced with terminals. Without the *moredigits* ::⇒ ϵ rule, the only rule we have with *MoreDigits* on the left side is the third rule: *MoreDigits* ::⇒ *Number*. The only rule we have with *Number* on the left side is the first rule, which replaces *Number* with *Digit MoreDigits*. Every time we go through this replacement cycle, we replace *MoreDigits* with *Digit MoreDigits*. We can produce as many *Digit*s as we want, but without another rule we can never stop.

This is the difference between a *circular* definition, and a *recursive* definition. Without the stopping rule, *MoreDigits* would be defined in a circular way. There is no way to start with *MoreDigits* and produce something that does not contain *MoreDigits* (or a nonterminal that eventually must produce *MoreDigits*) in it. With the *MoreDigits* ::⇒ ϵ rule, however, we have a way to produce something terminal from *MoreDigits*. This is known as a *base case* — a rule that turns an otherwise circular definition into a meaningful, recursive definition.

Figure 2.1 shows a parse tree for the derivation of **150** from *Number*.

Condensed Notation. It is commonly useful to have many grammar rules with the same left side nonterminal. For example, the whole numbers grammar has ten rules with *Digit* on the left side to produce the ten terminal digits. Each of these is an alternative rule that can be used when the production string contains the nonterminal *Digit*. A compact notation for these types of rules is to use the vertical bar (|) to separate alternative replacements. For example, we could write the ten *Digit* rules compactly as:

Digit ::⇒ **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Exercise 2.5. (⋄) The grammar for whole numbers is complicated because we do not want to include the empty string in our language. Devise a simpler grammar that defines the language of the whole numbers including the empty string.

Exercise 2.6. Suppose we replaced the first rule (*Number* ::⇒ *Digit MoreDigits*)

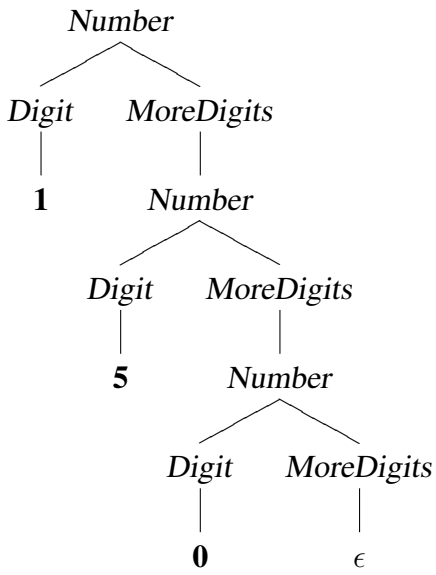


Figure 2.1: Deriviation of **150** from *Number*.

in the whole numbers grammar with this rule:

$$\textit{Number} \quad ::\Rightarrow \quad \textit{MoreDigits Digit}$$

a. ($\diamond$) How does this change the parse tree for the derivation of **150** from *Number*? Draw the parse tree that results from the new grammar.

b. ($\star$) Does this change the language? Either show some string that is in language defined by the modified grammar but not in the original language (or vice versa), or argue that both grammars can generate exactly the same sets of strings.

Exercise 2.7. ($\star$) Devise a grammar that defines the language of dates (e.g., "December 7, 1941"). Is it possible for your language to only include valid dates (that is, "January 31, 2007 is in the language, but "February 29, 2007" is not)?