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."

Lewis Carroll, *Through the Looking Glass*

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. In computing, we use language to describe procedures and use tools to turn descriptions of procedures in language that are easy for humans to read and write into executing processes. This chapter considers what a language is, how language works, and introduces the techniques we will use to define languages.

2.1 Surface Forms and Meanings

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 deliberate purpose (in particular, languages created for expressing procedures to be executed by computers).

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

[1] Thanks to Charles Yang for this definition.

describes a communication system between traffic lights and drivers:

Surface Form	Meaning
Green	Go
Yellow	Caution
Red	Stop

Rotary traffic signal

Communication systems involving humans are notoriously imprecise and subjective. A driver and a police officer may disagree on the actual meaning of the *Yellow* 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 know 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. It is impossible to express any *new* meaning using the communication system: all meanings must already be listed in the table!

Languages and Infinity

A real language must be able to express *infinitely* many different meanings. This means it must provide infinitely many surface forms; hence, there must be a system for generating new surface forms and a way of inferring the meaning of each generated 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:

$$1, 2, 3, \ldots$$

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. Thus, with only a few numbers and symbols we can describe a set containing infinitely many numbers.

The repeating pattern technique might be sufficient to describe some languages with infinitely many meanings. For example, this table defines an infinite language:

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

Although we can describe some infinite languages in this way, it is entirely 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 Construction

To define more expressive infinite languages, we need a richer system for constructing new surface forms and associated meanings. We need 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).

Components of 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".

[2] Languages that can be defined using simple repeating patterns in this way are known as *regular languages*.

For example, the verb *freeze* means to pass from a liquid state to a solid state; *antifreeze* is a substance designed to prevent freezing. English speakers who know the meaning of *freeze* and *anti-* could roughly guess the meaning of *antifreeze* even if they have never heard the word before.[3]

Note that the primitives are defined as the smallest units of *meaning*, not based on the surface forms. Both *anti* and *freeze* are morphemes; they cannot be broken into smaller parts with meaning. We can break *anti-* into two syllables, or four letters, but those sub-components do not have meanings that could be combined to produce 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 the word before. 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, by 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 sentence 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.

The next three sections introduce three different ways to define languages. The first system, *production systems*, is very powerful[4] but not widely used for defining languages today because it is too difficult (meaning it can be impossible) to determine if a given string is in the language. The next two, *recursive transition networks* and *replacement grammars* are less powerful than production systems, but more useful because they are simpler to reason about.

We focus on languages where the surface forms can easily be written down

[3] Guessing that it is a verb meaning to pass from the solid to liquid state would also be reasonable. This shows how imprecise and ambiguous natural languages are; for programming computers, we need the meanings of constructs to be clearly determined.

[4] In fact, it is exactly as powerful as a Turing machine (which we introduce in Chapter 6 and show is equivalent to a different model of computation in Chapter 17).

and interpreted linearly. This means the surface forms will be sequences of characters. A character is a symbol selected from a finite set of symbols known as an *alphabet*. A typical alphabet comprises the letters, numerals, and punctuation symbols used in English. 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. To define a language, we need to define a system that produces all strings in the language, and no other strings. 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. Its definition is, "truth that comes from the gut, not books". Identify the morphemes that are used to build *truthiness*, and explain, based on its composition, what *truthiness* should mean.

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. [⋆] Break *floccinaucinihilipilification* into its morphemes. Show that a speaker familiar with the morphemes could understand the word.

b. [⋆] Prove Guinness wrong by demonstrating a longer English word. An English speaker (familiar with the morphemes) should be able to deduce the meaning of your word.

Exploration 2.1: Wordsmithing

Embiggening your vocabulary with anticromulent words that ecdysiasts can grok.

a. [⋆] Invent a new English word by combining common morphemes.

b. [⋆] Get someone else to use the word you invented.

c. [⋆ ⋆ ⋆] Get Merriam-Webster to add your word to their dictionary.

2.3 Production Systems

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 paraphrased from the description from *GEB*):

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 1938.

Rule I: $x\mathsf{I} ::\Rightarrow x\mathsf{IU}$
"If you possess a string whose last letter is I, you may produce the string with U added at the end."
Rule II: $\mathsf{M}x ::\Rightarrow \mathsf{M}xx$
"If you have Mx, you may produce Mxx."
Rule III: $x\mathsf{III}y ::\Rightarrow x\mathsf{U}y$
"If III occurs in one of the strings in your collection, you may produce a new string with U in place of III."
Rule IV: $x\mathsf{UU}y ::\Rightarrow xy$
"If UU occurs inside your string, you can produce a string with it removed."

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

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. *MIU-system* productions.

a. [⋆] Using the *MIU-system*, show how M can be derived starting from MI?

b. [⋆] Using the *MIU-system*, how many different strings can be derived starting from UMI?

c. [⋆] Using the *MIU-system*, how many different strings can be derived starting from MI?

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

Exercise 2.4. [⋆] Devise a Post production system that can produce all the surface forms in the { "I will run today.", "I will run the day after today.", "I will run the day after the day after today.", . . . } language.

2.4 Recursive Transition Networks

Although Post production systems are powerful enough to generate complex languages, they are more awkward to use than we would like. In particular, applying 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 whether or not a given string is in the language (see Exercise 3). *Recursive transition networks* (RTNs) are a less powerful[5] way of defining a language, but provide a clearer way of under-

Recursive transition networks

[5] We mean less powerful in a formal sense: there are languages that can be described by a Post production system that cannot be defined by any recursive transition network. Exploration 2.2 discusses the relative power of different language definition mechanisms.

standing the set of strings in a defined language.

A recursive transition network is defined by a graph of nodes and edges. The edges are labeled with output symbols. One of the nodes is designated the start node (indicated by an arrow pointing into that node). One or more of the nodes may be designated as final nodes (indicated by an inner circle). A string is in the language if there exists some path from the start node to a final node in the graph where the output symbols along the path edges produce the string.

For example, Figure 2.1 shows a simple recursive transition network with three nodes and four edges.

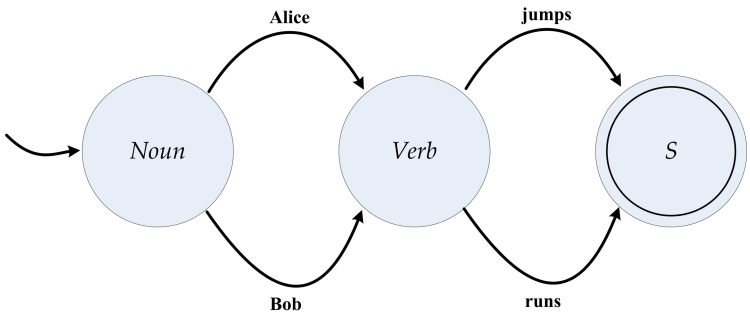


Figure 2.1. Simple recursive transition network.

This network can produce four different sentences. Starting in the node marked *Noun*, we have two possible edges to follow; each edge outputs a different symbol, and leads to the node marked *Verb*. From that node, we have two possible edges, each leading to the node marked *S*, which is a final node. Since there are no edges out of *S*, this ends the string. Hence, we can produce four strings corresponding to the four different paths from the start to final node: "Alice jumps", "Alice runs", "Bob jumps", and "Bob runs".

This way of defining a language is more efficient than just listing all strings in a table, since the number of possible strings increases with the number of possible paths in the graph. For example, adding one more edge from *Noun* to *Verb* with label "Colleen", would add two new strings to the language.

The expressive power of recursive transition networks really increases once we add edges that form cycles in the graph. This is where the *recursive* in the name comes from. Once a graph has a cycle, there are *infinitely* many possible paths through the graph, since we can always go around the cycle one more time. Consider what happens when we add a single edge to the previous network:

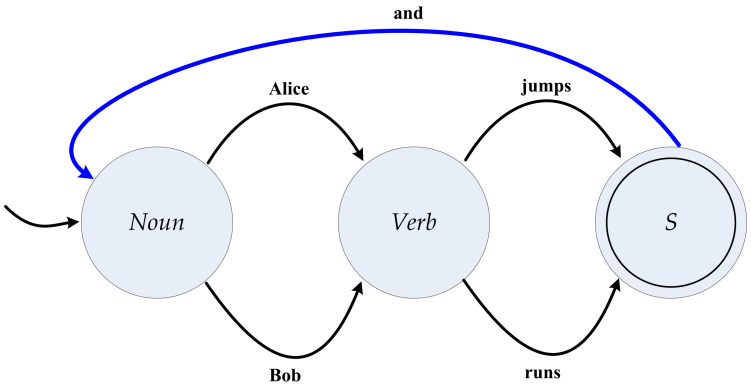


Figure 2.2. RTN with a cycle.

Now, we can produce infinitely many different strings! We can follow the "and" edge back to the *Noun* node, to produce strings like "Alice runs and Bob jumps and Alice jumps and Alice runs" with as many conjuncts as we want.

Exercise 2.5. [⋆] Draw a recursive transition network that defines the language of the whole numbers: 0, 1, 2, ...

Exercise 2.6. Recursive transition networks.

a. [⋆] What is the smallest number of edges needed for a recursive transition network that can produce exactly 8 strings?

b. [⋆] What is the smallest number of nodes needed for a recursive transition network that can produce exactly 8 strings?

c. [⋆] What is the smallest number of edges needed for a recursive transition network that can produce exactly n strings?

Exercise 2.7. [⋆⋆] Is it possible to define the language of the *MIU-system* using a recursive transition network? Either draw a network that matches the language produced by the *MIU-system*, or explain what it is impossible to do so.

2.4.1 Subnetworks

In the RTNs we have seen so far, the labels on the output edges are direct outputs known as *terminals*: following an edge just produces the symbol on that edge. We can make more expressive RTNs by allowing edge labels

to also name *subnetworks*. A subnetwork is identified by the name of its starting node. When an edge with a subnetwork label is followed, instead of outputting one symbol, the network traversal jumps to the subnetwork node. Then, it can follow any path from that node to a final node. Upon reaching a final node, the network traversal jumps back to complete the edge.

For example, consider the network shown in Figure 2.3. It describes the same language as the RTN in Figure 2.1, but uses subnetworks for *Noun* and *Verb*. To produce a string, we start in the *Sentence* node. The only edge out from *Sentence* is labeled *Noun*. To follow the edge, we jump to the *Noun* node, which is a separate subnetwork. Now, we can follow any path from *Noun* to a final node (in this cases, outputting either "Alice" or "Bob" on the path toward *EndNoun*.

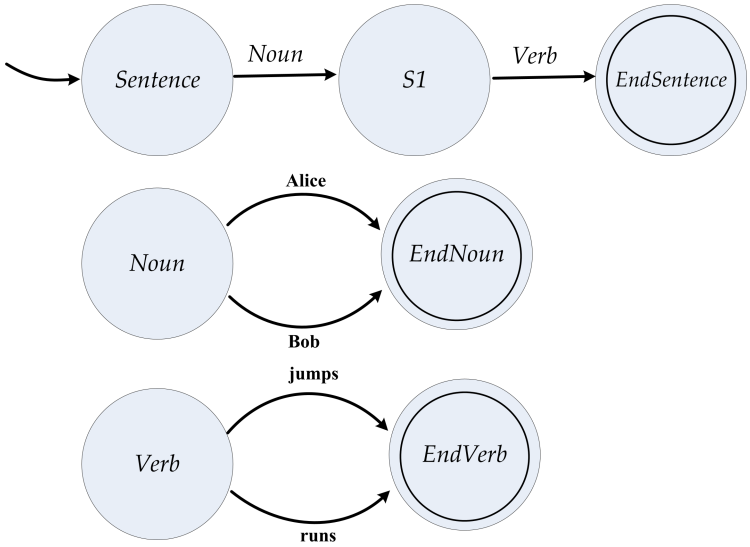


Figure 2.3. Recursive transition network with subnetworks.

Suppose we replace the *Noun* subnetwork with the more interesting version shown in Figure 2.4.

The new subnetwork includes an edge from *Noun* to *N1* labeled with *Noun*. So, if we follow this edge, control jumps back into the *Noun* node, for another path through the *Noun* subnetwork. Starting from *Noun*, we can generate complex phrases like "Alice and Bob" or "Alice and Bob and Alice" (note there are two different paths that generate this phrase).

To keep track of paths through RTNs without subnetworks, a single marker suffices. We can start with the marker on the start node, and move it along the path through each node to the final node. To keep track of paths on

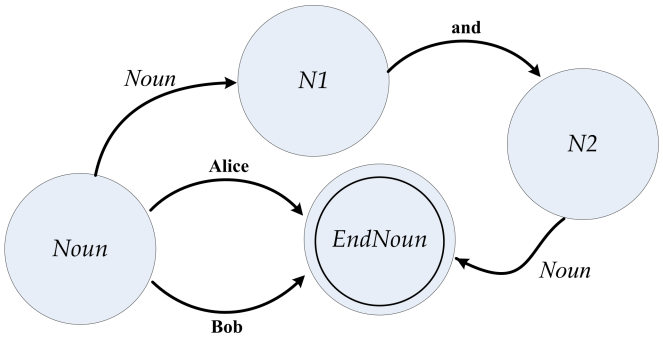


Figure 2.4. Alternate *Noun* subnetwork.

an RTN with subnetworks, though, is more complicated. We need to keep track of the where we are in the current network, but also where we need to jump to when a final node is reached. Since we can enter subnetworks within subnetworks, we need a way to keep track of arbitrarily many jump points.

A data structure for keeping track of this is known as a *stack*. We can think of a stack like a stack of trays in a cafeteria. At any point in time, only the *top* tray on the stack can be reached. We can take the top tray off the stack, after which the next tray is now on top. This operation is called *popping* the stack. We can push a new tray on top of the stack, which makes the old top of the stack now one below the new top. This operation is called *pushing*. *stack*

Using a stack, we can follow a path through an RTN using this procedure:[6]

1. Initially, push the starting node on the stack.
2. Pop a node, N, off the stack.
3. If N is a final node, check if the stack is empty. If the stack is empty, **stop**. Otherwise, go back to step 2.
4. Select an edge from the RTN that starts from node N. Use D to represent the destination of that edge, and s to output symbol on the edge.
5. Push D on the stack.
6. If s is a node (that is, the name of a subnetwork), push s on the stack. Otherwise, output s.
7. Go back to step 2.

For the example, we start by pushing *Sentence* on the stack. In step 2, we pop the stack, so the current node, N, is *Sentence*. Since it is not a final node, we do nothing for step 3. In step 4, we choose an edge starting from *Sentence*. There is only one edge to choose, and it leads to the node labeled

[6]For simplicity, this procedure assumes we always stop when a final node is reached. RTNs can have edges out of final nodes (as in Figure 2.2) where it is possible to either stop or continue from a final node.

S1. In step 5, we push *S1* on the stack. The label on the edge is *Noun*, which is a node, so we push *Noun* on the stack. The stack now contains two items: [*Noun*, *S1*]. As directed by step 7, we go back to step 2 and continue by popping the top node, *Noun*, off the stack. It is not a final node, so we continue to step 4, and select the edge from *Noun* to *N1*. Since *N1* is not a final node, we continue to step 5 and push *N1* on the stack. The label on the edge is *Noun*, which is a node, so we push *Noun* on the stack in step 6. At this point, the stack contains three nodes: [*Noun*, *N1*, *S1*].

We continue in the same manner, following the steps in the procedure as we keep track of a path through the network. Next, we pop *Noun*, and select the edge labeled "Alice", pushing *EndNoun* on the stack. Returning to step 2, we pop the *EndNoun*, which is a final node. Now, we are at the point to jump back to where we entered the subnetwork. This is the *N1* node, which is now on top of the stack. Continuing in step 2, we pop *N1*, and follow the edge labeled "and", continuing to node *N2*. This leads to another pass through the *Noun* subnetwork, after which we reach the *EndNoun* node. After continuing to step 3, the stack now contains just [*S1*]. In this manner, we can follow a path through the network, using the stack to keep track of the nodes to return to after finishing each subnetwork.

Exercise 2.8. Traversing RTNs.

a. [⋆] Show the sequence of stack values used in generating the string "Alice and Bob and Alice runs".

b. [⋆] Identify a string that cannot be produced with a stack that can hold no more than four elements.

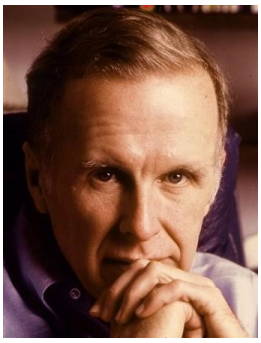
Exercise 2.9. [⋆] The procedure given for traversing RTNs assumes that a subnetwork path always stops when a final node is reached. Hence, it cannot follow all possible paths for an RTN where there are edges out of a final node. Describe a procedure that can follow all possible paths, even for RTNs that include edges from final nodes.

2.5 Replacement Grammars

Another way to define a language is to use a grammar.[7] This is the most common way languages are defined by computer scientists today, and the

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

way we will use for the rest of this book.



John Backus

A *grammar* is a set of rules for generating all strings in the language. The grammars we will use are a simple notation known as *Backus-Naur Form* (BNF). BNF was invented by John Backus in the late 1950s. Backus led efforts at IBM to define and implement Fortran, the first widely used high-level programming language. Fortran enabled computer programs to be written in a language more like familiar algebraic formulas than low-level machine instructions, enabling programs to be written more quickly and reliably (in the next chapter, we describe programming languages and how they are implemented). In defining the Fortran language, Backus and his team used ad hoc English descriptions to define the language. Backus developed the replacement grammar notation to precisely describe the language of a later programming language, Algol (1958). Peter Naur adapted the notation for the report describing the Algol language, and it was subsequently known as Backus-Naur Form at the suggestion of Donald Knuth to recognize both Backus' and Naur's contributions.

I flunked out every year. I never studied. I hated studying. I was just goofing around. It had the delightful consequence that every year I went to summer school in New Hampshire where I spent the summer sailing and having a nice time.
John Backus

Rules in a Backus-Naur Form grammar are of the form:

$$\textit{nonterminal} ::\Rightarrow \textit{replacement}$$

These rules are similar to Post production rules, except that the left side of a rule is always a single symbol, known as a *nonterminal* since it can never appear in the final generated string. Whenever we can match the nonterminal on the left side of a rule, we can replace it with what appears on the right side of the matching rule. This method of defining languages is exactly as powerful as recursive transition networks (the follow subsection considers why), but easier to write down.

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 a 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:

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

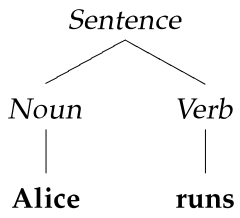
Starting from *Sentence*, we can generate four different sentences using the replacement rules: "Alice jumps", "Alice runs", "Bob jumps", and "Bob runs".

Derivation 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

Parse Tree 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 all strings 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 ("Colleen runs" and "Colleen jumps") 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 grammar is recursive if there is a way to start from a given nonterminal, and follow a sequence of one or more replacement rules to generate a production that contains the same nonterminal.

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 a language containing infinitely many strings.

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:

$$\textit{MoreDigits} ::\Rightarrow \epsilon$$

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* $::\Rightarrow \epsilon$ rule, the only rule we would have with *MoreDigits* on the left side is the third rule:

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

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 the *MoreDigits* $::\Rightarrow \epsilon$ rule we can never stop.

Circular vs. Recursive Definitions

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 generate a production that does not contain *MoreDigits* (or a nonterminal that eventually must produce *MoreDigits*). With the *MoreDigits* $::\Rightarrow \epsilon$ 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 2.5 shows a parse tree for the derivation of **150** from *Number*.

It is common 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:

$$\textit{Digit} ::\Rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

Exercise 2.10. [⋆] 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.

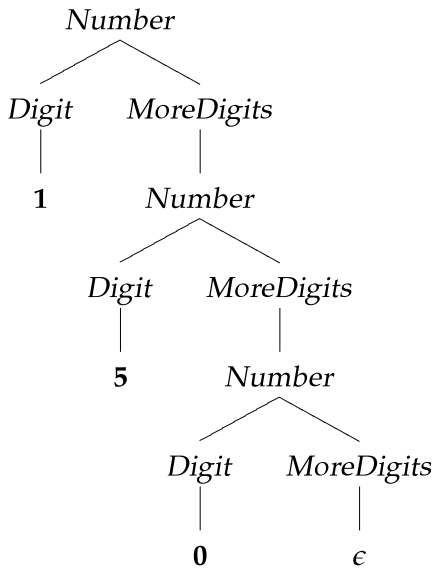


Figure 2.5. Deriviation of 150 from *Number*.

Exercise 2.11. Suppose we replaced the first rule (*Number* ::⇒ *Digit MoreDigits*) in the whole numbers grammar with this rule:

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

a. [⋆] 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. [⋆] Does this change the language? Either show some string that is in the 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 2.12. [⋆] The grammar for whole numbers we defined allows strings with non-standard leading zeros such as "000" and "00005". Devise a grammar that produces all whole numbers (including "0"), but no strings with unnecessary leading zeros.

Exercise 2.13. [⋆⋆] Devise a grammar that defines the language of valid dates (e.g., "December 7, 1941"). Your language should include all valid dates, but no invalid dates (that is, "September 29, 2007" and "February 29, 2008" are in the language, but "February 29, 2009" is not).

Exploration 2.2: Power of Language Systems

We claimed that recursive transition networks and BNF replacement grammars are *equally* powerful. Here, we explain more precisely what that means and prove that the two systems are, in fact, equivalent in power.

First, what does it mean to say two systems are equally powerful? The purpose of a language description mechanism is to define a set of strings comprising a language. Hence, the power of a language description mechanism is determined by the set of languages (that is, a set of sets of strings) it can define.

One approach to consider is counting the number of languages that can be defined. Even the simplest mechanisms can define infinitely many languages, however, so just counting the number of languages does not distinguish well between the different language description mechanisms. For example, even with the table listing all surface forms in the language (as introduced in Section 2.1) we can define infinitely many different languages. There is no limit on the number of entries in the table, so we can always add one more entry containing a new surface form to define a new language. Similarly, we can argue that both RTNs and BNFs can describe infinitely many different languages. We can always add a new edge to an RTN to increase the number of strings in the language, or add a new replacement rule to a BNF that replaces a nonterminal with a new terminal symbol.

Instead, we need to consider the particular languages that each mechanism can define. A system A is more powerful that another system B if we can use A to define every language that can be defined by B, and there is some language L that can be defined using A that cannot be defined using B. This matches our intuitive interpretation of *more powerful* — A is more powerful than B if it can do everything B can do and more. The set diagrams in Figure 2.6 depict three possible scenarios.

In the leftmost picture, the set of languages that can be defined by B is a proper subset of the set of languages that can be defined by A. Hence, A is more powerful than B. In the center picture, the sets are equal. This means every language that can be defined by A can also be defined by B, and every language that can be defined by B can also be defined by A, and the systems are equally powerful. In the rightmost picture, there are some elements of A that are not elements of B, but there are also some elements of B that are not elements of A. This means we cannot say either one is

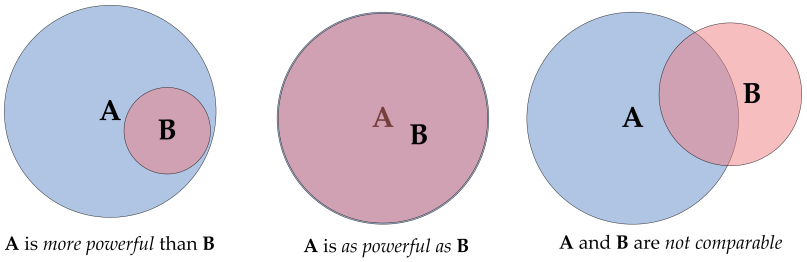


Figure 2.6. System power relationships.

more powerful; A can do some things B cannot do, and B can do some things A cannot do.

So, to determine the relationship between RTNs and BNFs, we need to understand if there are languages that can be defined by an RTN that cannot be defined by a BNF and if there are languages that can be defined by a BNF that cannot be defined by an RTN.

First, we will prove that there are no languages that can be defined by a BNF that cannot be defined by an RTN. This is equivalent to showing that *every* language that can be defined by a BNF grammar has a corresponding RTN. Since there are infinitely many languages that can be defined by BNF grammars, we obviously cannot prove this by enumerating each language and showing the corresponding RTN. Instead, we use a proof technique commonly used in computer science: *proof by construction*. We show that given any BNF grammar we can construct a corresponding RTN. That is, define an algorithm that takes as input a BNF grammar, and produces as output an RTN that defines the same language as the input BNF grammar.

Our general strategy is to construct a subnetwork corresponding to each nonterminal. For each rule where the nonterminal is on the left side, the right hand side is converted to a path through that node's subnetwork. Here is our algorithm for converting a BNF grammar to an equivalent RTN:

1. For each nonterminal X in the grammar, construct two nodes, *StartX* and *EndX*, where *EndX* is a final node. Make the node *StartS* the start node of the RTN, where S is the start nonterminal of the grammar.
2. For each rule in the grammar, add a corresponding path through the RTN. All BNF rules have the form X ::⇒ *replacement* where X is a nonterminal in the grammar and *replacement* is a sequence of zero or more terminals and nonterminals: $[R_0, R_1, \ldots, R_n]$.
   (a) If the replacement is empty, make *StartX* a final node.
   (b) If the replacement has just one element, R_0, add an edge from *StartX* to *EndX* with edge label R_0.

(c) Otherwise:

i. Add an edge from *StartX* to a new node labeled $X_{i,0}$ (where i identifies the grammar rule), with edge label R_0.

ii. For each remaining element R_j in the replacement add an edge from $X_{i,j-1}$ to a new node labeled $X_{i,j}$ with edge label R_j. (For example, for element R_1, a new node $X_{i,1}$ is added, and an edge from $X_{i,0}$ to $X_{i,1}$ with edge label R_1.)

iii. Add an edge from $X_{i,n-1}$ to *EndX* with edge label R_n.

Following this procedure, we can convert any BNF grammar into an RTN that defines the same language. Hence, we have proved that RTNs are at least as powerful as BNF grammars.

To complete the proof that BNF grammars and RTNs are equally powerful ways of defining languages, we also need to show that a BNF can define every language that can be defined using an RTN. This part of the proof can be done using a similar strategy: by showing a procedure that can be used to construct a BNF equivalent to any input RTN. We leave the details as an exercise for especially ambitious readers.

a. [⋆] Prove that BNF grammars are not more powerful than Post production systems.

b. [⋆] Prove that BNF grammars are as powerful as RTNs by devising a procedure that can construct a BNF grammar that defines the same language as any input RTN.