2

Language

Belittle! What an expression! It may be an elegant one in Virginia, and even perfectly intelligible; but for our part, all we can do is to guess at its meaning. For shame, Mr. Jefferson!
European Magazine and London Review, 1787
(reviewing Thomas Jefferson's *Notes on the State of Virginia*)

The most powerful tool we have for communication is language. This is true whether we are considering communication between two humans, between a human programmer and a computer, or between a network of computers. In computing, we use language to describe procedures and use machines to turn descriptions of procedures into executing processes. This chapter is about what language is, how language works, and ways to define languages.

2.1 Surface Forms and Meanings

A *language* is a set of surface forms and meanings, and a mapping between the surface forms and their associated meanings. In the earliest human languages, the surface forms were sounds but surface forms can be anything that can be perceived by the communicating parties such as drum beats, hand gestures, or pictures.

language

A *natural language* is a language spoken by humans, such as English or Swahili. Natural languages are very complex since they have evolved over many thousands years of individual and cultural interaction. We focus on *designed* languages that are created by humans for some a specific purpose such as for expressing procedures to be executed by computers.

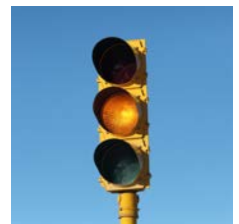
natural language

We focus on languages where the surface forms are text. In a textual language, the surface forms are linear sequences of characters. A *string* is a sequence of zero or more characters. Each character is a symbol drawn from a finite set known as an *alphabet*. For English, the alphabet is the set $\{a, b, c, \ldots, z\}$ (for the full language, capital letters, numerals, and punctuation symbols are also needed).

string

alphabet

A simple communication system can be described using 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	Go
Yellow	Caution
Red	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* 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

$$< \mathit{Symbol, Meaning} >$$

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 since all meanings must already be listed in the table!

Languages and Infinity. A useful language must be able to express *infinitely* many different meanings. Hence, there must be a way to generate new surface forms and guess their meanings (see Exercise 2.1). No finite representation, such as a printed table, can contain all the surface forms and meanings in an infinite language. One way to generate infinitely large sets is to use repeating patterns. For example, most humans would interpret 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. Thus, with only a few numbers and symbols we can describe a set containing infinitely many numbers. As discussed in Section 1.2.1, the language of the natural numbers is enough to encode all meanings in any countable set. But, finding a sensible mapping between most meanings and numbers is nearly impossible. The surface forms do not correspond closely enough to the ideas we want to express to be a useful language.

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 to describe languages that allow us to define an infinitely large set of surface forms and meanings with a compact notation. The approach we use is to define a language by defining a set of rules that produce exactly the set of surface forms in the language.

Components of Language. A language is composed of:

- *primitives* — the smallest units of meaning.
- *means of combination* — rules for building new language elements by combining simpler ones.

The primitives are the smallest meaningful units (in natural languages these are known as *morphemes*). A primitive cannot be broken into smaller parts whose meanings can be combined to produce the meaning of the unit. The means of combination are rules for building words from primitives, 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 can be inferred as "against the meaning of the original word". Rules like this one mean 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.

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.[1]

Primitives are the smallest units of *meaning*, not based on the surface forms. Both *anti* and *freeze* are primitive; 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 primitive.

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, there are a limited number of means of abstraction. English, in particular, has a very limited set of pronouns for abstracting people. It has *she* and *he* for abstracting a female or male person, respectively, but no gender-neutral pronouns for abstracting a person of either sex. The interpretation of what a pronoun abstract in natural languages is often confusing. For example, it is unclear what the *it* in this sentence refers to. Languages for programming computers need means of abstraction that are both powerful and unambiguous.

Exercise 2.1. 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". This word was reputedly invented by a non-hippopotomonstrosesquipedaliophobic student at Eton who combined four words in his Latin textbook. Prove Guinness wrong by identifying a longer English word. An English speaker (familiar with floccinaucinihilipilification and the morphemes you use) should be able to deduce the meaning of your word.

Exercise 2.2. 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.

[1] 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.

Exercise 2.3. According to the Oxford English Dictionary, Thomas Jefferson is the first person to use more than 60 words in the dictionary. Jeffersonian words include: (a) authentication, (b) belittle, (c) indecipherable, (d) inheritability, (e) odometer, (f) sanction, (g) vomit-grass, and (h) shag. For each Jeffersonian word, guess its derivation and explain whether or not its meaning could be inferred from its components.

Dictionaries are but the depositories of words already legitimated by usage. Society is the workshop in which new ones are elaborated. When an individual uses a new word, if ill formed, it is rejected; if well formed, adopted, and after due time, laid up in the depository of dictionaries.
Thomas Jefferson, letter to John Adams, 1820

Exercise 2.4. Embiggening your vocabulary with anticromulent words ecdysiasts can grok.

a. Invent a new English word by combining common morphemes.

b. Get someone else to use the word you invented.

c. [★★] Convince Merriam-Webster to add your word to their dictionary.

2.3 Recursive Transition Networks

This section describes a more powerful technique for defining languages. The surface forms of a textual language are a (typically infinite) set of strings. 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 more difficult; we consider it in later chapters.)

recursive transition network

A *recursive transition network* (RTN) is defined by a graph of nodes and edges. The edges are labeled with output symbols—these are the primitives in the language. The nodes and edge structure provides the means of combination.

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.

Figure 2.1 shows a simple RTN with three nodes and four edges that can produce four different sentences. Starting at the node marked *Noun*, there are two possible edges to follow. Each edge outputs a different symbol, and leads to the node marked *Verb*. From that node there are two output edges, each leading to the final node marked *S*. Since there are no edges out of *S*, this ends the string. Hence, the RTN 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”.

Recursive transition networks are more efficient than listing the strings in a language, since the number of possible strings increases with the number of possible paths through the graph. For example, adding one more edge from *Noun* to

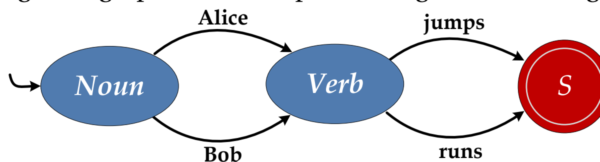


Figure 2.1. Simple recursive transition network.

Verb with label "Colleen" adds two new strings to the language.

The expressive power of recursive transition networks increases dramatically 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!

Consider what happens when we add the single "and" edge to the previous network to produce the network shown in Figure 2.2 below.

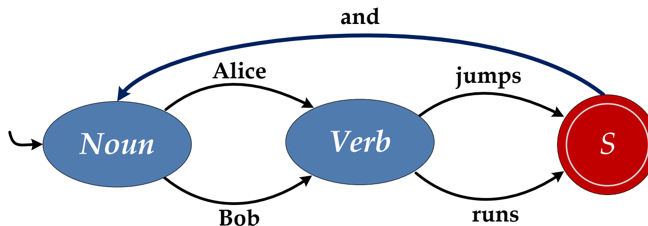
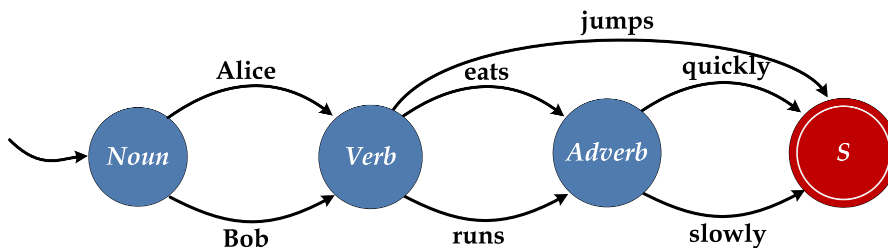


Figure 2.2. RTN with a cycle.

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" 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. How many different strings can be produced by the RTN below:



Exercise 2.7. Recursive transition networks.

a. How many nodes are needed for a recursive transition network that can produce exactly 8 strings?

b. How many edges are needed for a recursive transition network that can produce exactly 8 strings?

c. [⋆⋆] Given a whole number n, how many edges are needed for a recursive transition network that can produce exactly n strings?

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 labeled with a subnetwork is followed, 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*.

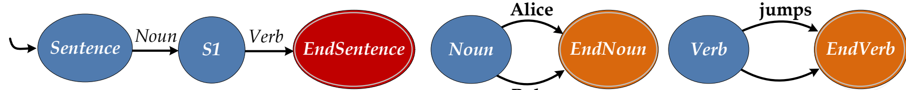


Figure 2.3. Recursive transition network with subnetworks.

Suppose we replace the *Noun* subnetwork with the more interesting version shown in Figure 2.4.This subnetwork includes an edge from *Noun* to *N1* labeled *Noun*. Following this edge involves following a path through the *Noun* subnetwork. Starting from *Noun*, we can generate complex phrases like "Alice and Bob" or "Alice and Bob and Alice" (find the two different paths through the network that generate this phrase).

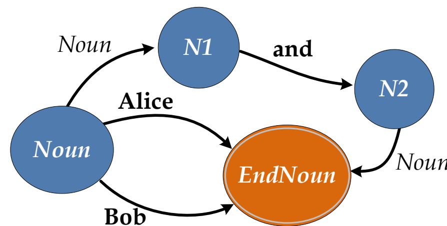


Figure 2.4. Alternate *Noun* subnetwork.

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

stack

A *stack* is a useful way to keep track of the subnetworks. 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 *pop* the top tray off the stack, after which the next tray is now on top. We can *push* a new tray on top of the stack, which makes the old top of the stack now one below the top.

We use a stack of nodes to keep track of the subnetworks as they are entered. The top of the stack represents the next node to process. At each step, we pop the node off the stack and follow a transition from that node.

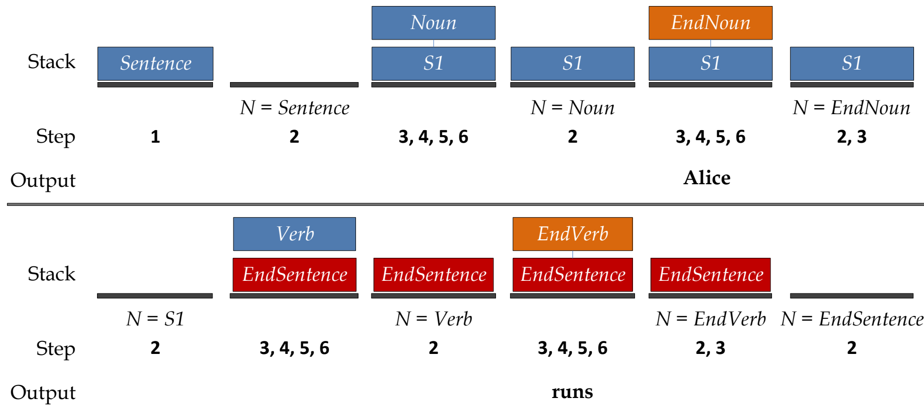


Figure 2.5. RTN generating "Alice runs".

Using a stack, we can derive a path through an RTN using this procedure:

1. Initially, push the starting node on the stack.
2. If the stack is empty, **stop**. Otherwise, pop a node, N, off the stack.
3. If the popped node, N, is a final node return to step 2.[2]
4. Select an edge from the RTN that starts from node N. Use D to denote the destination of that edge, and s to denote the output symbol on the edge.
5. Push D on the stack.
6. If s is a subnetwork, push the node s on the stack. Otherwise, output s, which is a terminal.
7. Go back to step 2.

Consider generating the string "Alice runs" using the RTN in Figure 2.3. We start following step 1 by pushing *Sentence* on the stack. In step 2, we pop the stack, so the current node, N, is *Sentence*. Since *Sentence* is not a final node, we do nothing for step 3. In step 4, we follow an edge starting from *Sentence*. There is only one edge to choose and it leads to the node labeled *S1*. In step 5, we push *S1* on the stack. The edge we followed is labeled with the node *Noun*, so we push *Noun* on the stack. The stack now contains two items: [*Noun, S1*]. Since *Noun* is on top, this means we will first traverse the *Noun* subnetwork, and then continue from *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 labeled "Alice" from *Noun* to *EndNoun*. We push *EndNoun* on the stack, which now contains: [*EndNoun, S1*]. The label on the edge is the terminal, "Alice", so we output "Alice" following step 6. We continue in the same manner, following the steps in the procedure as we keep track of a path through the network. The full processing steps are shown in Figure 2.5.

Exercise 2.8. Show the sequence of stacks used in generating the string "Alice and Bob and Alice runs" using the network in Figure 2.3 with the alternate *Noun* subnetwork from Figure 2.4.

[2]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.

Exercise 2.9. Identify a string that cannot be produced using the RTN from Figure 2.3 with the alternate *Noun* subnetwork from Figure 2.4 without the stack growing to contain five elements.

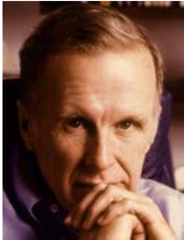
Exercise 2.10. The procedure given for traversing RTNs assumes that a subnetwork path always stops when a final node is reached. Hence, it cannot follow all possible paths for an RTN where there are edges out of a final node. Describe a procedure that can follow all possible paths, even for RTNs that include edges from final nodes.

2.4 Replacement Grammars

Another way to define a language is to use a grammar. 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.

grammar

A *grammar* is a set of rules for generating all strings in the language. We use the *Backus-Naur Form* (BNF) notation to define a grammar. BNF grammars are exactly as powerful as recursive transition networks (Exploration 2.1 explains what this means and why it is the case), but easier to write down.



John Backus

BNF was invented by John Backus in the late 1950s. Backus led efforts at IBM to define and implement Fortran, the first widely used 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. In defining the Fortran language, Backus and his team used ad hoc English descriptions to define the language. These ad hoc descriptions were often misinterpreted, motivating the need for a more precise way of defining a language.

Rules in a Backus-Naur Form grammar have the form:

$$nonterminal ::\Rightarrow \ replacement$$

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

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. 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 output 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. The terminals are the primitives in the language; the grammar rules are its means of combination.

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. Wherever we find a nonterminal on the left side of a rule, we can replace it with what appears on the right side of any rule where that nonterminal matches the left side. A string is generated once there are no nonterminals remaining.

Here is an example BNF grammar (that describes the same language as the RTN

in Figure 2.1):

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

Starting from *Sentence*, the grammar can generate four sentences: "Alice jumps", "Alice runs", "Bob jumps", and "Bob runs".

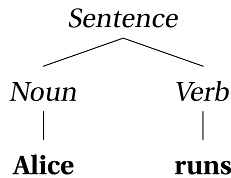
A *derivation* shows how a grammar generates a given string. Here is the derivation of "Alice runs":

derivation

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":

parse tree



BNF grammars can be more compact than just listing strings in the language since a grammar can have many replacements for each nonterminal. For example, adding the rule, *Noun* ::⇒ **Colleen**, to the grammar adds 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 the grammar contains a nonterminal that can produce a production that contains itself.

Suppose we add the rule,

Sentence ::⇒ *Sentence* **and** *Sentence*

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

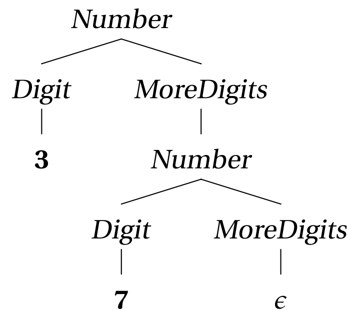
Infinitely many! This grammar describes the same language as the RTN in Figure 2.2. It can generate "Alice runs and Bob jumps" and "Alice runs and Bob jumps and Alice runs" and sentences with any number of repetitions of "Alice runs". This is very powerful: by using recursive rules a compact grammar can be used to define a language containing infinitely many strings.

Example 2.1: Whole Numbers

This grammar defines the language of the whole numbers (0, 1, ...) with leading zeros allowed:

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

Here is the parse tree for a derivation of **37** from *Number*:



Circular vs. Recursive Definitions. The second rule means 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 would 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 follow this rule, we replace *MoreDigits* with *Digit MoreDigits*. We can produce as many *Digit*s as we want, but without the *MoreDigits* ::⇒ ϵ 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 generate a production that does not contain *MoreDigits* (or a nonterminal that eventually must produce *MoreDigits*). 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.

base case

Condensed Notation. 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:

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

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

a. How does this change the parse tree for the derivation of **37**? 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 generate the same 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. Define a BNF grammar that describes the language of decimal numbers (the language should include 3.14159, 0.423, and 1120 but not 1.2.3).

Exercise 2.14. The BNF grammar below (extracted from Paul Mockapetris, *Domain Names - Implementation and Specification*, IETF RFC 1035) describes the language of domain names on the Internet.

Domain	::⇒	*SubDomainList*
SubDomainList	::⇒	*Label* | *SubDomainList* **.** *Label*
Label	::⇒	*Letter MoreLetters*
MoreLetters	::⇒	*LetterHyphens LetterDigit* | ϵ
LetterHyphens	::⇒	*LDHyphen* | *LDHyphen LetterHyphens* | ϵ
LDHyphen	::⇒	*LetterDigit* | **-**
LetterDigit	::⇒	*Letter* | *Digit*
Letter	::⇒	**A** | **B** | ... | **Z** | **a** | **b** | ... | **z**
Digit	::⇒	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

a. Show a derivation for **www.virginia.edu** in the grammar.

b. According to the grammar, which of the following are valid domain names: (1) **tj**, (2) **a.-b.c**, (3) **a-a.b-b.c-c**, (4) **a.g.r.e.a.t.d.o.m.a.i.n-**.

Exploration 2.1: Power of Language Systems

Section 2.4 claimed that recursive transition networks and BNF grammars are equally powerful. What does it mean to say two systems are equally powerful?

A language description mechanism is used to define a set of strings comprising a language. Hence, the power of a language description mechanism is determined by the set of languages it can define.

One approach to measure the power of language description mechanism would be to count the number of languages that it can define. Even the simplest mech-

anisms can define infinitely many languages, however, so just counting the number of languages does not distinguish well between the different language description mechanisms. 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 set of 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 diagrams in Figure 2.6 show 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 more powerful; A can do some things B cannot do, and B can do some things A cannot do.

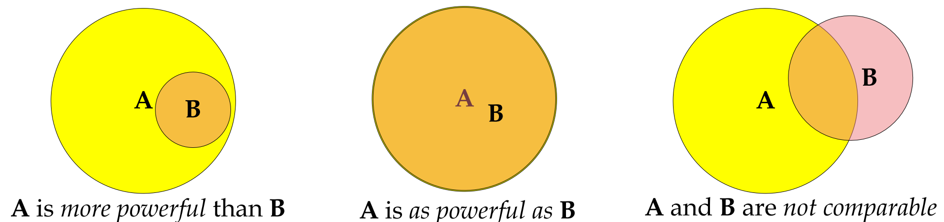


Figure 2.6. System power relationships.

To determine the relationship between RTNs and BNFs we need to understand if there are languages that can be defined by a BNF that cannot be defined by a RTN and if there are languages that can be defined by a RTN that cannot be defined by an BNF. We will show only the first part of the proof here, and leave the second part as an exercise.

For the first part, we prove that there are no languages that can be defined by a BNF that cannot be defined by an RTN. Equivalently, *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 cannot prove this by enumerating each language and showing its corresponding RTN. Instead, we use a proof technique commonly used in computer science: *proof by construction*. We show an algorithm that given any BNF grammar constructs an RTN that defines the same language as the input BNF grammar.

proof by construction

Our 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.

Before presenting the general construction algorithm, we illustrate the approach with the example BNF grammar from Example 2.1:

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

The grammar has three nonterminals: *Number*, *Digit*, and *MoreDigits*. For each nonterminal, we construct a subnetwork by first creating two nodes corresponding to the start and end of the subnetwork for the nonterminal. We make *StartNumber* the start node for the RTN since *Number* is the starting nonterminal for the grammar.

Next, we need to add edges to the RTN corresponding to the production rules in the grammar. The first rule indicates that *Number* can be replaced by *Digit MoreDigits*. To make the corresponding RTN, we need to introduce an intermediate node since each RTN edge can only contain one label. We need to traverse two edges, with labels *StartDigit* and *StartMoreDigits* between the *StartNumber* and *EndNumber* nodes. The resulting partial RTN is shown in Figure 2.7.

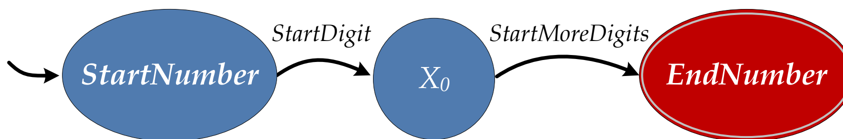


Figure 2.7. Converting the *Number* productions to an RTN.

For the *MoreDigits* nonterminal there are two productions. The first means *MoreDigits* can be replaced with nothing. In an RTN, we cannot have edges with unlabeled outputs. So, the equivalent of outputting nothing is to turn *StartMoreDigits* into a final node. The second production replaces *MoreDigits* with *Number*. We do this in the RTN by adding an edge between *StartMoreDigits* and *EndMoreDigits* labeled with *Number*, as shown in Figure 2.8.

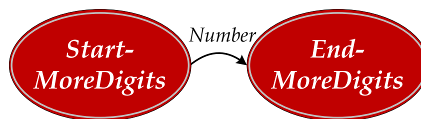


Figure 2.8. Converting the *MoreDigits* productions to an RTN.

Finally, we convert the ten *Digit* productions. For each rule, we add an edge between *StartDigit* and *EndDigit* labeled with the digit terminal, as shown in Figure 2.9.

This example illustrates that it is possible to convert a particular grammar to an RTN. For a general proof, we present a general an algorithm that can be used to do the same conversion for any BNF:

1. For each nonterminal *X* in the grammar, construct two nodes, *StartX* and

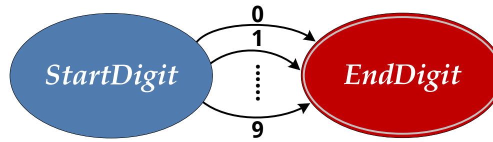


Figure 2.9. Converting the *Digit* productions to an RTN.

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 in reverse: 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.

Exercise 2.15. Produce an RTN that defines the same languages as the BNF grammar from Exercise 2.14.

Exercise 2.16. [⋆] 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.

2.5 Summary

Languages define a set of surface forms and associated meanings. Since useful language must be able to express infinitely many things, we need tools for

defining infinite sets of surface forms using compact and precise notations. The tool we will use for the remainder of this book is the BNF replacement grammar which precisely defines a language using replacement rules. This system can describe infinite languages with small representations because of the power of recursive rules. In the next chapter, we introduce the Scheme programming language that we will use to describe procedures.