2
Language

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-hippopotomonstrose-squipedaliophobic 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.

Solution. One option would be to add the suffix *-able* to make the adjective *floccinaucinihilipilificationable*, which would mean something like, "regarding the act or habit of describing or regarding something as worthless", although this is quite a floccinaucinihilipilificationable 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.truthinessColbert, Stephen

Solution. The morphemes are "truth" + "-y" + "-ness". "Truth" has many meanings, but the one used here is "state of being the case (fact)". The "-y" suffix makes a noun and adjective, meaning the "like that of", so "truthy" would be interpreted as "like the truth". The word "truthy" does appear in the dictionary. Its traditional definition is "Truthful, or seeming to be true" (http://en.wiktionary.org/wiki/truthy). The "y" transforms into an "i" in the spelling because of spelling rules that transform mid-word "y"s into "i"s. The suffix "-ness" means "the state of being something" (e.g., "dryness" is the state of being dry). So, "truthiness" should mean "the state of being like the truth", which is somewhat different from Colbert's definition. Of course, the real meaning of words is all about how people interpret them, and Colbert's definition has been widespread enough that most English speakers would interpret it the way he wants now.

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.Jefferson, Thomasodometershagbelittleauthentication-belittle

Solution. Search the Oxford English Dictionary on-line (http://www.oed.com, only through university subscriptions) for definitions and origins of each word.

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

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

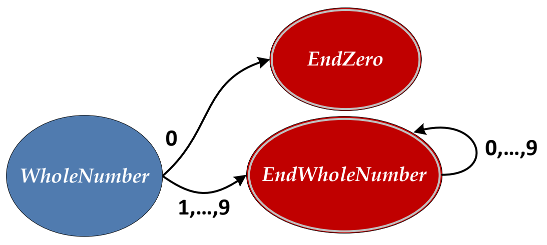
b. Get someone else to use the word you invented.

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

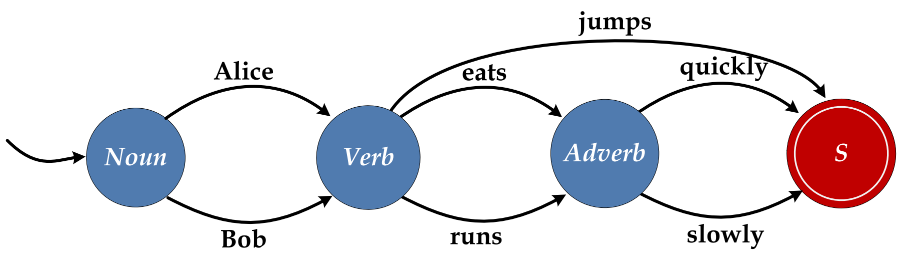
Solution. There's obviously no solution to this, but I should mention that *embiggen* and *cromulent* were coined by an episode of *The Simpsons*, and *ecdysiast* was invented by H. L. Mencken to mean "strip-tease artist" (adapting the Greek *ekdysis*).

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

Solution. Since we expect a whole number to have at least one digit (the empty string is not allowed), and cannot have leading zeros (e.g., 003 is not a valid whole number), we need a special state to handle 0.



Exercise 2.6. How many different strings can be produced by the RTN below:



Solution. There are 10 total strings, corresponding to each path through the RTN to the final *S* state: *Alice jumps, Alice eats slowly, Alice eats quickly, Alice runs slowly, Alice runs quickly, Bob jumps, Bob eats slowly, Bob eats quickly, Bob runs slowly Bob runs quickly.*

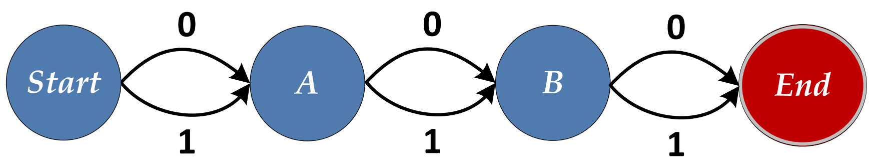
Exercise 2.7. Recursive transition networks.

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

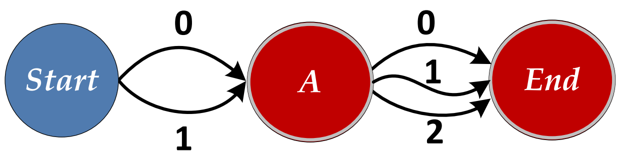
Solution. Only 2 nodes are needed to produce any number of strings! We can always have an arbitrary number of edges between the two nodes.

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

Solution. If there is only one final state allowed, the minimum number of edges is 6. To produce 8 total strings, we need two choices three times ($2 \times 2 \times 2 = 8$).



If there can be more than one final state, though, it is possible to use only five edges!



I believe there is no RTN with fewer than five edges that can produce exactly 8 strings, but a convincing proof that this is the case is worth a gold star.

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

Solution. Unknown (at least to me)! This is quite tricky, hence the [⋆⋆] .

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.

Solution.

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.

Solution. For each **and**, the stack needs to grow to store either the *N1* or *N2* node (and one more node for the original *Noun*). So, an example of a sentence that requires a stack depth of five is **Alice and Alice and Alice and Alice and Alice runs**.

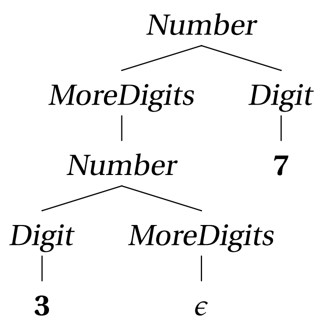
Exercise 2.10. The procedure given for traversing RTNs assumes that a subnetwork path always stops when a final node is reached. Hence, it cannot follow all possible paths for an RTN where there are edges out of a final node. Describe a procedure that can follow all possible paths, even for RTNs that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

Solution. To allow continuing from a final node, we need to change step 3 to be: If the popped node, N, is a final node either return to step 2 or continue to step 4. Note that this procedure, as well as the original one, is *nondeterministic*. That means we cannot executed it by simply following the steps mechanically, but instead must make choices. (In the original procedure, the choice was hidden in the *Select* at the beginning of step 3 — the procedure does not specify which edge to select when there are several choices.) One way to mechanically execute a non-deterministic procedure is to systematically try all possible choices until one is found that leads to the desired solution (in this case, that is ending with an empty stack and the desired output).

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.

Solution.



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.

Solution. Although this changes the way numbers are parsed, it does not change the language. The production *Number* ::⇒ *Digit MoreDigits* generates the same strings as *Number* ::⇒ *MoreDigits Digit* since *MoreDigits* is the same in both, and it generates a sequence of zero or more digits. The difference is whether the single digit produced by *Digit* is before or after the sequence of zero or more digits produces by *MoreDigits*. Since all digits are interchangeable, though, this produces the same set 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.

Solution. To eliminate the leading zeros, we need to add a new nonterminal for *NonZeroDigit*, and a special rule for **0**.

Number	::⇒	**0**
Number	::⇒	*NonZeroDigit MoreDigits*
MoreDigits	::⇒	
MoreDigits	::⇒	*Number*
Digit	::⇒	**0**
Digit	::⇒	*NonZeroDigit*
NonZeroDigit	::⇒	**1** | **2** | ··· | **9**

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).

Solution. We assume the *Number* definition from Example 2.1.

$$\begin{array}{lll}
\mathit{Decimal} & ::\Rightarrow & \mathit{Number\ OptMantissa} \\
\mathit{OptMantissa} & ::\Rightarrow & \epsilon \\
\mathit{OptMantissa} & ::\Rightarrow & \textbf{.}\ \mathit{Number}
\end{array}$$

This does allow numbers like **003.200**. A stricter definition of decimal numbers that disallows leading zeros would use the *Number* definition from the previous exercise, but would need to define a *FullNumber* nonterminal also to allow leading zeros to the right of the decimal point.

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.

$$\begin{array}{lll}
\mathit{Domain} & ::\Rightarrow & \mathit{SubDomainList} \\
\mathit{SubDomainList} & ::\Rightarrow & \mathit{Label} \mid \mathit{SubDomainList}\ \textbf{.}\ \mathit{Label} \\
\mathit{Label} & ::\Rightarrow & \mathit{Letter\ MoreLetters} \\
\mathit{MoreLetters} & ::\Rightarrow & \mathit{LetterHyphens\ LetterDigit} \mid \epsilon \\
\mathit{LetterHyphens} & ::\Rightarrow & \mathit{LDHyphen} \mid \mathit{LDHyphen\ LetterHyphens} \mid \epsilon \\
\mathit{LDHyphen} & ::\Rightarrow & \mathit{LetterDigit} \mid \textbf{-} \\
\mathit{LetterDigit} & ::\Rightarrow & \mathit{Letter} \mid \mathit{Digit} \\
\mathit{Letter} & ::\Rightarrow & \mathbf{A} \mid \mathbf{B} \mid \ldots \mid \mathbf{Z} \mid \mathbf{a} \mid \mathbf{b} \mid \ldots \mid \mathbf{z} \\
\mathit{Digit} & ::\Rightarrow & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}
\end{array}$$

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

Solution. Note that the grammar is ambiguous, so there are many other ways to produce the same string. Here is one possible derivation:

$$\begin{array}{rll}
\mathit{Domain} & ::\Rightarrow & \underline{\mathit{SubDomainList}} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \underline{\mathit{Label}} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \underline{\mathit{Letter}}\ \mathit{MoreLetters} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \mathbf{e}\ \underline{\mathit{MoreLetters}} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \mathbf{e}\ \underline{\mathit{LetterHyphens}}\ \mathit{LetterDigit} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \mathbf{e}\ \underline{\mathit{LDHyphen}}\ \mathit{LetterHyphens\ LetterDigit} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \mathbf{e}\ \underline{\mathit{LetterDigit}}\ \mathit{LetterHyphens\ LetterDigit} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \mathbf{e}\ \underline{\mathit{Letter}}\ \mathit{LetterHyphens\ LetterDigit} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \mathbf{e\ d}\ \underline{\mathit{LetterHyphens}}\ \mathit{LetterDigit} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \mathbf{e\ d}\ \underline{\mathit{LetterDigit}} \qquad \text{(using } \mathit{LetterHyphens} ::\Rightarrow \epsilon\text{)} \\
 & ::\Rightarrow & \mathit{SubDomainList}\ \textbf{.}\ \mathbf{e\ d}\ \underline{\mathit{Letter}} \\
 & ::\Rightarrow & \underline{\mathit{SubDomainList}}\ \textbf{.}\ \mathbf{e\ d\ u} \\
 & ::\Rightarrow^{*} & \mathit{SubDomainList}\ \textbf{.}\ \underline{\mathit{Label}}\ \textbf{.}\ \mathbf{e\ d\ u} \qquad \text{(we fast-forward the steps for } \mathit{Label} ::\Rightarrow^{*} \mathbf{virginia}\text{)} \\
 & ::\Rightarrow & \underline{\mathit{SubDomainList}}\ \textbf{.}\ \mathbf{v\ i\ r\ g\ i\ n\ i\ a}\ \textbf{.}\ \mathbf{e\ d\ u} \\
 & ::\Rightarrow & \underline{\mathit{Label}}\ \textbf{.}\ \mathbf{v\ i\ r\ g\ i\ n\ i\ a}\ \textbf{.}\ \mathbf{e\ d\ u} \\
 & ::\Rightarrow & \mathbf{w\ w\ w}\ \textbf{.}\ \mathbf{v\ i\ r\ g\ i\ n\ i\ a}\ \textbf{.}\ \mathbf{e\ d\ u} \qquad \text{(fast-forwarding } \mathit{Label} ::\Rightarrow^{*} \mathbf{w\ w\ w}\text{)}
\end{array}$$

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-**.

Solution.

(1) **tj** is a grammatically valid domain name: *Domain* $::\Rightarrow$ *SubDomainList* $::\Rightarrow$ *Label* $::\Rightarrow^{*}$ **tj**.

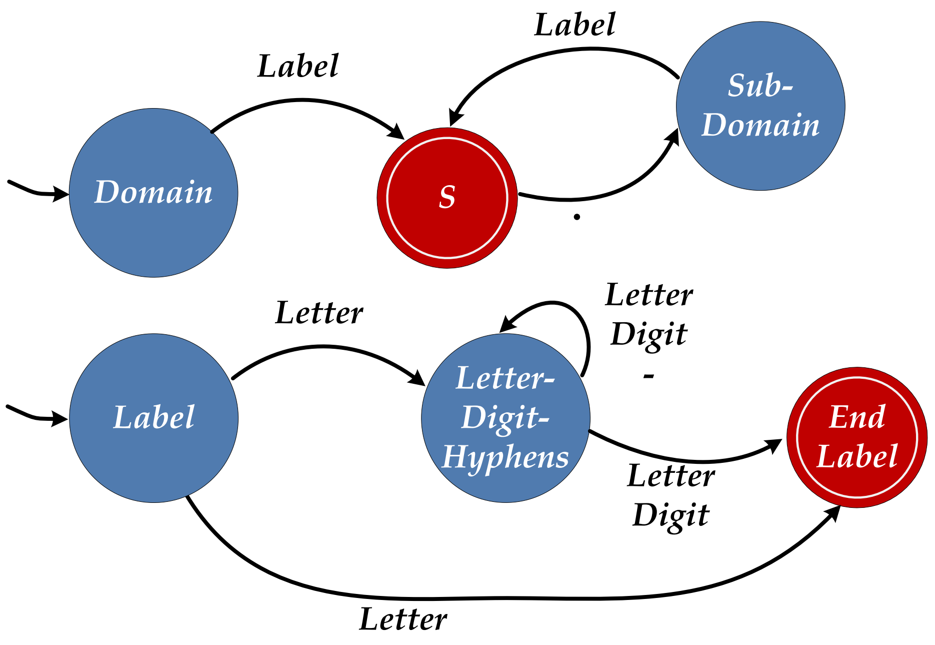
(2) **a.-b.c** is not a grammatically valid domain name. The *Label* cannot produce the string **-b** since the only production for *Label* is *Label* ::⇒ *Letter MoreLetters* and *Letter* cannot produce **-**.

(3) **a-a.b-b.c-c** is a grammatically valid domain name: *Domain* ::⇒ *SubDomainList* ::⇒* *Label* **.** *Label* **.** *Label* ::⇒* **a-a.b-b.c-c**.

(4) **a.g.r.e.a.t.d.o.m.a.i.n-** is a not grammatically valid domain name. The *Label* productions cannot produce a label that ends in a **-** since *MoreLetters* ::⇒ *LetterHyphens LetterDigit* | ϵ a label must end in a letter or digit.

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

Solution.



Note the need for the bottom edge from *Label* to *EndLabel* that is necessary to allow single-letter labels.

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.

Solution.