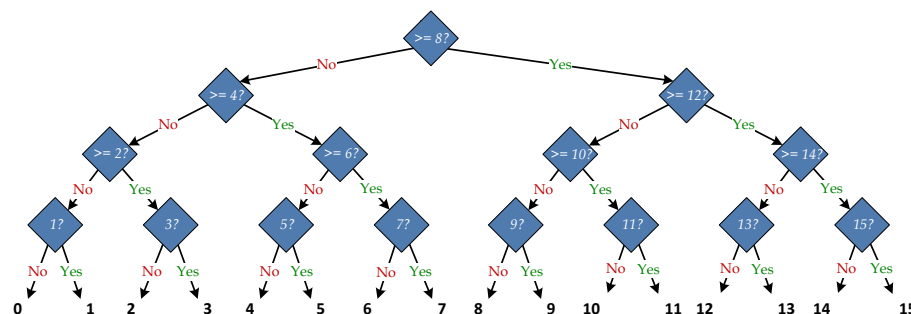


Computing

Exercise 1.1. Draw a binary tree with the minimum possible depth to:

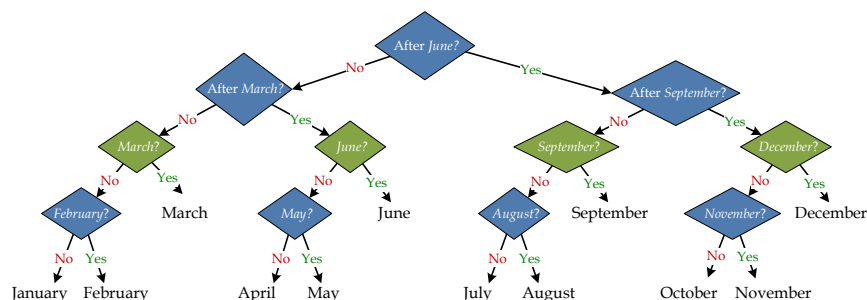
- a. Distinguish among the numbers $0, 1, 2, \dots, 15$.

Solution. There are sixteen (2^4) different numbers, so the minimum depth binary tree has four levels. There are many different trees that could work, so long as each decision divides the remaining possible numbers in two equal-sized sets. The easiest way to construct it is to start with the 3-bit tree in Figure 1.1 which distinguishes 0–7. We make two copies of this, but add 8 to each number in the second copy, and add an extra binary question at the top to decide which subtree to use.



- b. Distinguish among the 12 months of the year.

Solution. Since there are 12 months, we need $\log_2 12 \approx 3.58$ bits to distinguish them. This means we need a tree with 4 levels, but on some paths only three questions are needed. One solution would be to use 0–11 from the previous solution. A more natural solution might divide the year into quarters.



Note that the answers for the green boxes do not provide a full bit of information since the “No” answer leads to two leaf nodes (e.g., January and February), but the “Yes” answer only

leads to one leaf node (e.g., March). If each month is equally likely, the answer should be “No” two thirds of the time.

Exercise 1.2. How many bits are needed:

a. To uniquely identify any currently living human?

Solution. According to the U. S. Census Bureau, the world population (on July 4, 2011) is 6.95 Billion people. See <http://www.census.gov/main/www/popclock.html> for an updated count. To identify each person, we need

$$\lceil \log_2 6,950,000,000 \rceil = 33 \text{ bits}$$

(the notation $\lceil x \rceil$ means the “ceiling” of x which is the least integer larger than x). Since $2^{33} = 8,589,934,592$, 33 bits should be enough to uniquely identify every living person for at least the next decade.

b. To uniquely identify any human who ever lived?

Solution. This is much tougher, and requires defining a *human*. The best estimate I can find come from Carl Haub’s article for the Population Reference Bureau, originally written in 1995 and updated in 2002 (<http://www.prb.org/Articles/2002/HowManyPeopleHaveEverLivedonEarth.aspx>). He estimated that modern humans emerged about 50,000 years ago, and that the total number of humans born up to 2002 was 106 Billion. The world birth rate is approximately 130 million per year, so a current estimate would be perhaps 1-2 Billion more. So, 36 bits is certainly not enough ($2^{36} = 68,719,476,736$), but 37 bits should be enough for a long time ($2^{37} = 137,438,953,472$).

c. To identify any location on Earth within one square centimeter?

Solution. The total surface area of the Earth is $510,072,000 \text{ km}^2$. One kilometer is $100 * 1000 = 100,000$ centimeters, so one square kilometer is $(100 * 1000)^2 = 10,000,000,000$ square centimeters. So, the total surface area of the Earth is $5,100,720,000,000,000,000$ square centimeters. The number of bits needed to represent this is $\log_2 5,100,720,000,000,000,000 \approx 62.145$ so 63 bits is enough to uniquely identify every square centimeter of the Earth’s surface. (For comparison, Internet addresses are 128 bits. This is enough for each square centimeter of the Earth to have over 66 quadrillion IP addresses!)

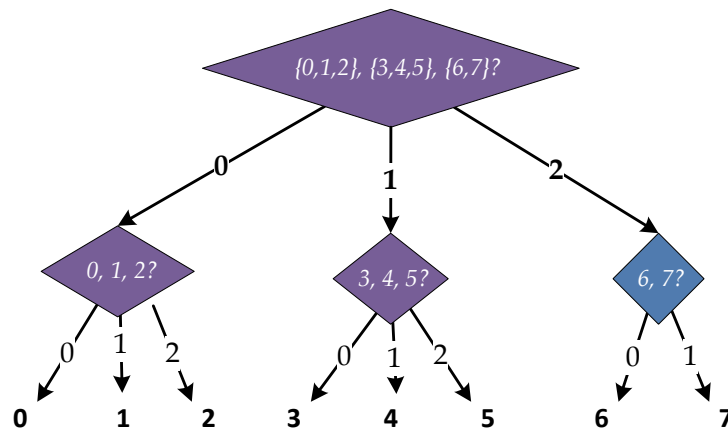
d. To uniquely identify any atom in the observable universe?

Solution. The number of atoms in the universe is estimated to be 10^{80} (see http://en.wikipedia.org/wiki/Observable_universe for a good explanation of how this number was derived). To determine the number of bits needed to uniquely identify every atom, we need to convert from decimal to binary. It requires $\log_2 10 \approx 3.32$ bits to represent each decimal digit, so $10^{80} \approx 2^{265.7}$. Hence, 266 bits is enough to uniquely identify every atom in the universe.

Exercise 1.3. The examples all use binary questions for which there are two possible answers. Suppose instead of basing our decisions on bits, we based it on *trits* where one trit can distinguish between three equally likely values. For each trit, we can ask a ternary question (a question with three possible answers).

a. How many trits are needed to distinguish among eight possible values? (A convincing answer would show a ternary tree with the questions and answers for each node, and argue why it is not possible to distinguish all the values with a tree of lesser depth.)

Solution. Two trits are required since $\log_3 8 \approx 1.893$. Here is one such tree:



All of the decision nodes except for the bottom rightmost one use three possible outputs. Hence, we could distinguish one additional value using 2 trits (which makes sense since $3^2 = 9$).

- b. [★] Devise a general formula for converting between bits and trits. How many trits does it require to describe b bits of information?

Solution. To convert between logarithm bases, we use the formula

$$\log_b x = \frac{\log_a x}{\log_a b}.$$

So, for a n -bit value x ,

$$\log_3 x = \frac{n}{\log_2 3} \approx 0.63093n$$

Thus, to represent an n -bit value requires up to $\lceil 0.63093n \rceil$ trits.

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-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.

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. *truthiness* Colbert, 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, Thomas odometershagbelittleauthentication-belittle*

Solution. Search the Oxford English Dictionary on-line (<http://www.oed.com>, only through uni-

versity subscriptions) for definitions and origins of each word.

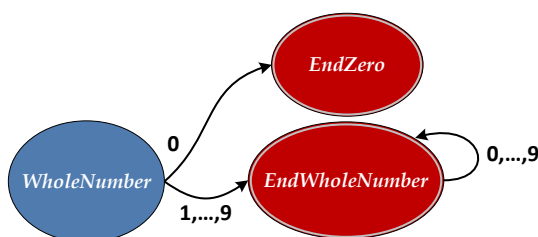
Exercise 2.4. Embiggening your vocabulary with anticromulent words ecdysiasts can grok.

- Invent a new English word by combining common morphemes.
- Get someone else to use the word you invented.
- ★★ Convince Merriam-Webster to add your word to their dictionary.

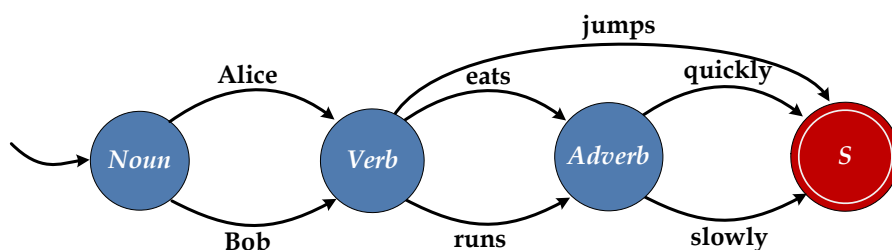
Solution. There's obviously no solution to this, but I should mention that *embiggen* and *cromulent* were coined by an episode of *The Simpsons*, and *ecdysiast* was invented by H. L. Mencken to mean “strip-tease artist” (adapting the Greek *ekdysis*).

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*.

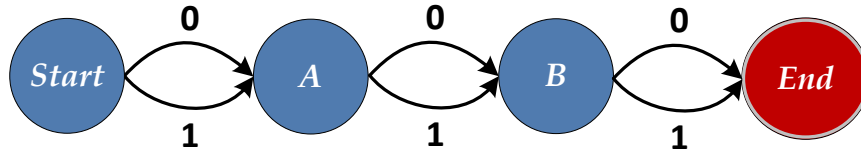
Exercise 2.7. Recursive transition networks.

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

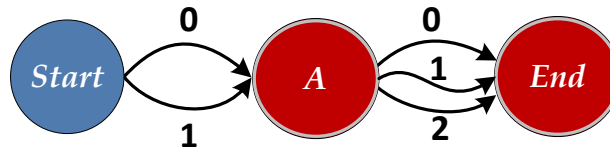
Solution. Only 2 nodes are needed to produce any number of strings! We can always have an arbitrary number of edges between the two nodes.

- 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.

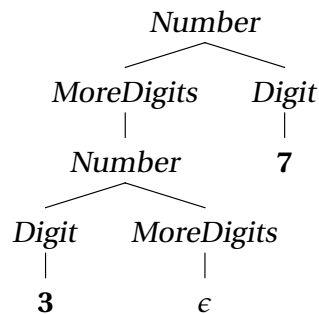
Solution. To allow continuing from a final node, we need to change step 3 to be: If the popped node, N , is a final node either return to step 2 or continue to step 4. Note that this procedure, as well as the original one, is *nondeterministic*. That means we cannot execute it by simply following the 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 ($Number ::= \Rightarrow Digit\ MoreDigits$) in the whole numbers grammar with: $Number ::= \Rightarrow MoreDigits\ Digit$.

- 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 $Number ::= \Rightarrow Digit\ MoreDigits$ generates the same strings as $Number ::= \Rightarrow MoreDigits\ Digit$ 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 produced 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.

$Number \quad ::= \Rightarrow 0$
 $Number \quad ::= \Rightarrow NonZeroDigit\ MoreDigits$
 $MoreDigits \quad ::= \Rightarrow$
 $MoreDigits \quad ::= \Rightarrow Digit\ MoreDigits$
 $Digit \quad ::= \Rightarrow 0$
 $Digit \quad ::= \Rightarrow NonZeroDigit$
 $NonZeroDigit ::= \Rightarrow 1 \mid 2 \mid \dots \mid 9$

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.

$Decimal ::= \Rightarrow Number\ OptMantissa$
 $OptMantissa ::= \Rightarrow \epsilon$
 $OptMantissa ::= \Rightarrow .\ Number$

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.

$Domain ::= \Rightarrow SubDomainList$
 $SubDomainList ::= \Rightarrow Label\ |\ SubDomainList\ .\ Label$
 $Label ::= \Rightarrow Letter\ MoreLetters$
 $MoreLetters ::= \Rightarrow LetterHyphens\ LetterDigit\ |\ \epsilon$
 $LetterHyphens ::= \Rightarrow LDHyphen\ |\ LDHyphen\ LetterHyphens\ |\ \epsilon$
 $LDHyphen ::= \Rightarrow LetterDigit\ |\ -$
 $LetterDigit ::= \Rightarrow Letter\ |\ Digit$
 $Letter ::= \Rightarrow A\ |\ B\ |\ \dots\ |\ Z\ |\ a\ |\ b\ |\ \dots\ |\ z$
 $Digit ::= \Rightarrow 0\ |\ 1\ |\ 2\ |\ 3\ |\ 4\ |\ 5\ |\ 6\ |\ 7\ |\ 8\ |\ 9$

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:

$Domain ::= \Rightarrow SubDomainList$
 $\quad ::= \Rightarrow SubDomainList\ .\ Label$
 $\quad ::= \Rightarrow SubDomainList\ .\ Letter\ MoreLetters$
 $\quad ::= \Rightarrow SubDomainList\ .\ e\ MoreLetters$
 $\quad ::= \Rightarrow SubDomainList\ .\ e\ LetterHyphens\ LetterDigit$
 $\quad ::= \Rightarrow SubDomainList\ .\ e\ LDHyphen\ LetterHyphens\ LetterDigit$
 $\quad ::= \Rightarrow SubDomainList\ .\ e\ LetterDigit\ LetterHyphens\ LetterDigit$
 $\quad ::= \Rightarrow SubDomainList\ .\ e\ Letter\ LetterHyphens\ LetterDigit$
 $\quad ::= \Rightarrow SubDomainList\ .\ e\ d\ LetterHyphens\ LetterDigit$
 $\quad ::= \Rightarrow SubDomainList\ .\ e\ d\ LetterDigit$ (using $LetterHyphens ::= \Rightarrow \epsilon$)
 $\quad ::= \Rightarrow SubDomainList\ .\ e\ d\ Letter$
 $\quad ::= \Rightarrow SubDomainList\ .\ e\ d\ u$
 $\quad ::= \Rightarrow^* SubDomainList\ .\ Label\ .\ e\ d\ u$ (we fast-forward the steps for $Label ::= \Rightarrow^* \text{virginia}$)
 $\quad ::= \Rightarrow SubDomainList\ .\ \text{virginia}\ .\ e\ d\ u$
 $\quad ::= \Rightarrow Label\ .\ \text{virginia}\ .\ e\ d\ u$
 $\quad ::= \Rightarrow \text{www}\ .\ \text{virginia}\ .\ e\ d\ u$ (fast-forwarding $Label ::= \Rightarrow^* \text{www}$)

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-**.

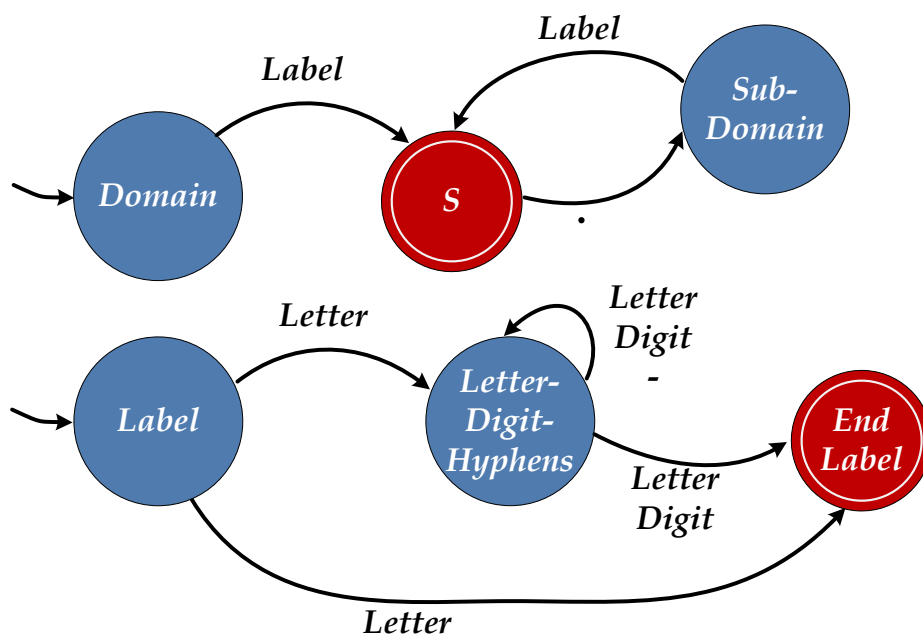
Solution.

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

- (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 not a grammatically valid domain name. The *Label* productions cannot produce a label that ends in a **-** since $MoreLetters ::= LetterHyphens\ LetterDigit \mid \epsilon$ 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.



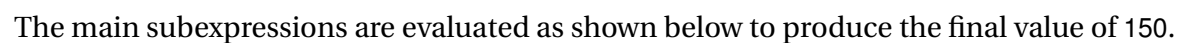
Note the need for the bottom edge from *Label* to *EndLabel* that is necessary to allow single-letter labels.

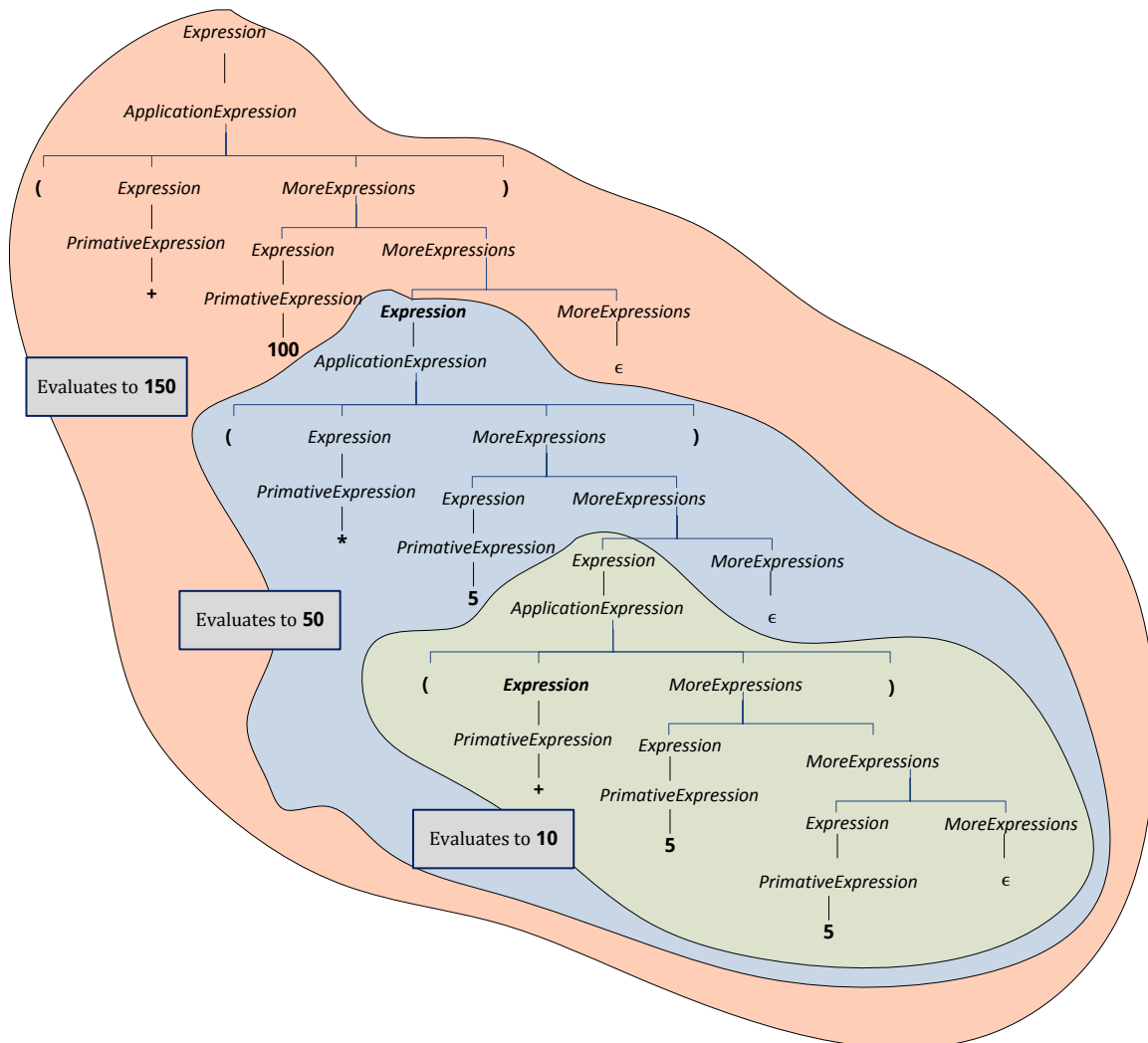
Exercise 2.16. [★] Prove that BNF grammars are as powerful as RTNs by devising a procedure that can construct a BNF grammar that defines the same language as any input RTN.

Solution.

Programming

Solution. The full parse tree for the expression is:





Exercise 3.2. Predict how each of the following Scheme expressions is evaluated. After making your prediction, try evaluating the expression in DrRacket. If the result is different from your prediction, explain why the Scheme interpreter evaluates the expression as it does.

a. 1120

Solution. 1120

b. (+ 1120)

Solution. 1120

c. (+ (+ 10 20) (* 2 0))

Solution. 30

d. (= (+ 10 20) (* 15 (+ 5 5)))

Solution. #f


The #f symbol represents the Boolean value *false*. Since the two operand expressions for the = have different values, the expression evaluates to #f.

e. +

Solution. #<procedure:+>

The symbol + is a primitive expression that is the built-in procedure for addition.

f. (+ + <)

Solution.  +: expects type <number> as 1st argument, given: #<procedure:+>; other arguments were: #<procedure:<>>

The interpreter produces an error since the + procedure is only defined for operands that are numbers. Since the value of the first argument is a procedure, there is a type error.

Exercise 3.3. For each question, construct a Scheme expression and evaluate it in DrRacket.

a. How many seconds are there in a year?

Solution. For non-leap years, there are 365 days in a year, 24 hours in a day, 60 minutes in an hour, and 60 seconds in a minute: (* 60 60 24 365)

b. For how many seconds have you been alive?

Solution. The first number (e.g., 20 here) is the number of years you have been alive: (* 20 60 60 24 365)

c. For what fraction of your life have you been in school?

Solution. Let's assume you are 20 years old and you've been in school for 15 years and in the United States it is typical to have 180 school days in a year, and that each school day is 7 hours long. Then, we can compute the fraction of your life spent in school by dividing the number of hours spent in school by the number of hours lived: (/ (* 15 180 7) (* 20 365 24)). This evaluates to 63/584. Note that math in Scheme is exact, and it is represented as a rational number. A more useful result is to convert it to a decimal number using the *exact->inexact* procedure:

```
> (exact->inexact (/ (* 15 180 7) (* 20 365 24)))
0.10787671232876712
```

Your answer, of course, will vary depending on how old you are, how many years you've spent in school, and the length of your school year and day.

Exercise 3.4. Construct a Scheme expression to calculate the distance in inches that light travels during the time it takes the processor in your computer to execute one cycle. (A meter is defined as the distance light travels in $1/299792458^{th}$ of a second in a vacuum. Hence, light travels at 299,792,458 meters per second. Your processor speed is probably given in *gigahertz* (GHz), which are 1,000,000,000 hertz. One hertz means once per second, so 1 GHz means the processor executes 1,000,000,000 cycles per second. On a Windows machine, you can find the speed of your processor by opening the Control Panel (select it from the Start menu) and selecting System. Note that Scheme performs calculations exactly, so the result will be displayed as a fraction. To see a more useful answer, use (*exact->inexact Expression*) to convert the value of the expression to a decimal representation.)

Solution. My computer is 2.33 GHz. Hence, one cycle takes (/ 1 2330000000) seconds. Light travels 299792458 meters per second. So, in the time the processor executes a single cycle, light travels

```
> (exact->inexact (* 299792458 (/ 1 2330000000)))
0.12866629098712445
```

meters. This is equivalent to

```
> (exact->inexact (/ (* 100 (* 299792458 (/ 1 2330000000))) 2.54))
5.065602007367104
```

inches. This should give you some sense of how close the processor speed is to reaching physical limits.

Exercise 3.5. Define a procedure, *cube*, that takes one number as input and produces as output the cube of that number.

Solution.

```
(define cube (lambda (x) (* x x x)))
```

or:

```
(define (cube x) (* x x x))
```

Exercise 3.6. Define a procedure, *compute-cost*, that takes as input two numbers, the first represents that price of an item, and the second represents the sales tax rate. The output should be the total cost, which is computed as the price of the item plus the sales tax on the item, which is its price times the sales tax rate. For example, (*compute-cost* 13 0.05) should evaluate to 13.65.

Solution.

```
(define compute-cost
  (lambda (price rate)
    (+ price (* price rate))))
```

Another approach:

```
(define (compute-cost price rate)
  (* price (+ 1 rate)))
```

Exercise 3.7. Follow the evaluation rules to evaluate the Scheme expression:

```
(bigger 3 4)
```

where *bigger* is the procedure defined above. (It is very tedious to follow all of the steps (that's why we normally rely on computers to do it!), but worth doing once to make sure you understand the evaluation rules.)

Exercise 3.8. Define a procedure, *xor*, that implements the logical exclusive-or operation. The *xor* function takes two inputs, and outputs true if exactly one of those outputs has a true value. Otherwise, it outputs false. For example, (*xor* true true) should evaluate to false and (*xor* (< 3 5) (= 8 8)) should evaluate to true.

Solution. There are many possible ways to define *xor*. Here are two possibilities:

```
(define (xor a b)
  (if a (not b) b))
```

```
(define (xor a b)
  (or (and a (not b)) (and (not a) b)))
```

Exercise 3.9. Define a procedure, *absvalue*, that takes a number as input and produces the absolute value of that number as its output. For example, (*absvalue* 3) should evaluate to 3 and (*absvalue* -150) should evaluate to 150.

Solution.

```
(define (absvalue v)
  (if (< v 0) (- v) v))
```

Note that Scheme provides a built-in function *abs* that implements the absolute value function. Hence, the easiest way to define *absvalue* would be,

```
(define absvalue abs)
```

Exercise 3.10. Define a procedure, *bigger-magnitude*, that takes two inputs, and outputs the value of the input with the greater magnitude (that is, absolute distance from zero). For example, (*bigger-magnitude* 5 -7) should evaluate to -7, and (*bigger-magnitude* 9 -3) should evaluate to 9.

Solution. We use the *absvalue* procedure defined in the previous exercise:

```
(define (bigger-magnitude a b)
  (if (> (absvalue a) (absvalue b)) a b))
```

Exercise 3.11. Define a procedure, *biggest*, that takes three inputs, and produces as output the maximum value of the three inputs. For example, (*biggest* 5 7 3) should evaluate to 7. Find at least two different ways to define *biggest*, one using *bigger*, and one without using it.

Solution. Our first solution uses the *bigger* procedure from Example 3.3:

```
(define (biggest a b c)
  (bigger (bigger a b) c))
```

Another approach is to define an if expression:

```
(define (biggest a b c)
  (if (> a b)
      (if (> a c) a c)
      (if (> b c) b c)))
```

We prefer the first version since it is shorter and easier to understand. This is a simple illustration of the advantages of building up more complex procedures by combining simple ones.

4

Problems and Procedures

Exercise 4.1. For each expression, give the value to which the expression evaluates. Assume *fcompose* and *inc* are defined as above.

a. $((fcompose\ square\ square)\ 3)$

Solution. The application expression $(fcompose\ square\ square)$ evaluates to a procedure that composes *square* with *square* (that is, it multiplies its input by itself four times). Hence, applying this procedure to 3 evaluates to 81.

b. $(fcompose\ (\lambda(x)\ (*\ x\ 2))\ (\lambda(x)\ (/ \ x\ 2)))$

Solution. This evaluates to an identity procedure for number inputs. It produces a procedure that takes a number as its input, and applies a procedure that multiplies by 2 to the result of a procedure that divides the input number by 2.

c. $((fcompose\ (\lambda(x)\ (*\ x\ 2))\ (\lambda(x)\ (/ \ x\ 2)))\ 1120)$

Solution. This applies the identity procedure from the previous part to 1120, so the result is 1120.

d. $((fcompose\ (fcompose\ inc\ inc)\ inc)\ 2)$

Solution. The inner application expression, $(fcompose\ inc\ inc)$, evaluates to a procedure that takes a number as its input and outputs the result of incrementing it twice (that is, adding 2). The next application expression, $(fcompose\ (fcompose\ inc\ inc)\ inc)$, composes this with another *inc* procedure, producing a procedure that adds 3 to its input. Applying this procedure to 2 results in the value 5.

Exercise 4.2. Suppose we define *self-compose* as a procedure that composes a procedure with itself:

(define (*self-compose* *f*) (*fcompose* *f* *f*))

Explain how $((fcompose\ self-compose\ self-compose)\ inc)\ 1)$ is evaluated.

Solution. The application expression, $(fcompose\ self-compose\ self-compose)$, produces a procedure that composes *self-compose* with itself. Using the substitution evaluation rules and the definition of *fcompose*, this expression evaluates to

$((\lambda(f\ g)\ (\lambda(x)\ (g\ (f\ x))))\ self-compose\ self-compose)$

which evaluates to $(\lambda(x)\ (self-compose\ (self-compose\ x)))$. Applying this to *inc* results in $(self-compose\ (self-compose\ inc))$. Substituting the definition of *self-compose*, we get:

(fcompose (fcompose inc inc) (fcompose inc inc))

Now, we can substitute the definition of *fcompose* for the outer application to get:

(lambda (x) ((fcompose inc inc) ((fcompose inc inc) x)))

This expression is applied to 1, producing ((fcompose inc inc) ((fcompose inc inc) 1)). Next, we substitute the definition of *fcompose* in the inner application to get:

((fcompose inc inc) ((blambda (f g) (blambda (x) (g (f x))) inc inc) 1))

Using the application rule, this simplifies to ((fcompose inc inc) (blambda (x) (inc (inc x))) 1)). Applying again, substituting 1 for x, we get:

((fcompose inc inc) (inc (inc 1)))

After performing the *inc* applications, this is ((fcompose inc inc) 3). The remaining application expressions are evaluated the same way, producing the final value of 5.

Exercise 4.3. Define a procedure *fcompose3* that takes three procedures as input, and produces as output a procedure that is the composition of the three input procedures. For example, ((fcompose3 *abs inc square*) -5) should evaluate to 36. Define *fcompose3* two different ways: once without using *fcompose*, and once using *fcompose*.

Solution. Without using *fcompose*:

(define (fcompose3 f1 f2 f3)
 (lambda (x) (f3 (f2 (f1 x)))))

Using *fcompose*:

(define (fcompose3 f1 f2 f3)
 (fcompose (fcompose f1 f2) f3))

Exercise 4.4. The *fcompose* procedure only works when both input procedures take one input. Define a *f2compose* procedure that composes two procedures where the first procedure takes two inputs, and the second procedure takes one input. For example, ((f2compose + *abs*) 3 -5) should evaluate to 2.

Solution.

(define (f2compose f g)
 (lambda (x y)
 (g (f x y)))

Exercise 4.5. How many different ways are there of choosing an unordered 5-card hand from a 52-card deck?

This is an instance of the “*n* choose *k*” problem (also known as the binomial coefficient): how many different ways are there to choose a set of *k* items from *n* items. There are *n* ways to choose the first item, *n* - 1 ways to choose the second, ..., and *n* - *k* + 1 ways to choose the *k*th item. But, since the order does not matter, some of these ways are equivalent. The number of possible ways to order the *k* items is *k*!, so we can compute the number of ways to choose *k* items from a

set of n items as:

$$\frac{n * (n - 1) * \cdots * (n - k + 1)}{k!} = \frac{n!}{(n - k)!k!}$$

- a. Define a procedure *choose* that takes two inputs, n (the size of the item set) and k (the number of items to choose), and outputs the number of possible ways to choose k items from n .

Solution.

```
(define (choose n k)
  (/ (factorial n) (* (factorial (- n k)) (factorial k))))
```

- b. Compute the number of possible 5-card hands that can be dealt from a 52-card deck.

Solution.

```
> (choose 52 5)
2598960
```

- c. [★] Compute the likelihood of being dealt a flush (5 cards all of the same suit). In a standard 52-card deck, there are 13 cards of each of the four suits. Hint: divide the number of possible flush hands by the number of possible hands.

Solution. The number of possible flushes for each suit is the number of ways to choose 5 cards from the 13 cards of each suit. So the total number of possible flushes is $(* 4$ (*choose* 13 5)). To compute the probability of being dealt a 5-card flush, we divide the number of ways to make a flush by the number of 5-card hands:

```
> (/ (* 4 (choose 13 5)) (choose 52 5))
33/16660
> (exact->inexact (/ (* 4 (choose 13 5)) (choose 52 5)))
0.0019807923169267707
```

So, you should expect to see a 5-card flush roughly once every 505 hands.

Exercise 4.6. Gauss, Karl Reputedly, when Karl Gauss was in elementary school his teacher assigned the class the task of summing the integers from 1 to 100 (e.g., $1 + 2 + 3 + \cdots + 100$) to keep them busy. Being the (future) “Prince of Mathematics”, Gauss developed the formula for calculating this sum, that is now known as the *Gauss sum*. Had he been a computer scientist, however, and had access to a Scheme interpreter in the late 1700s, he might have instead defined a recursive procedure to solve the problem. Define a recursive procedure, *gauss-sum*, that takes a number n as its input parameter, and evaluates to the sum of the integers from 1 to n as its output. For example, (*gauss-sum* 100) should evaluate to 5050.

Solution.

```
(define (gauss-sum n)
  (if (= n 1) 1
      (+ n (gauss-sum (- n 1)))))
```

Exercise 4.7. [★] *accumulate* Define a higher-order procedure, *accumulate*, that can be used to make both *gauss-sum* (from Exercise 4.6) and *factorial*. The *accumulate* procedure should take as its input the function used for accumulation (e.g., $*$ for *factorial*, $+$ for *gauss-sum*). With

your *accumulate* procedure, $((\text{accumulate } +) 100)$ should evaluate to 5050 and $((\text{accumulate } *) 3)$ should evaluate to 6. We assume the result of the base case is 1 (although a more general procedure could take that as a parameter).

Hint: since your procedure should produce a procedure as its output, it could start like this:

```
(define (accumulate f)
  (lambda (n)
    (if (= n 1) 1
        ...
```

Solution.

```
(define (accumulate f)
  (lambda (n)
    (if (= n 1) 1
        (f n ((accumulate f) (- n 1))))))
```

Here are a few examples:

```
> ((accumulate +) 100)
5050
> ((accumulate *) 100)
5050
> ((accumulate (lambda (x y) (- x y))) 10)
5
```

Exercise 4.8. To find the maximum of a function that takes a real number as its input, we need to evaluate at all numbers in the range, not just the integers. There are infinitely many numbers between any two numbers, however, so this is impossible. We can approximate this, however, by evaluating the function at many numbers in the range.

Define a procedure *find-maximum-epsilon* that takes as input a function *f*, a low range value *low*, a high range value *high*, and an increment *epsilon*, and produces as output the maximum value of *f* in the range between *low* and *high* at interval *epsilon*. As the value of *epsilon* decreases, *find-maximum-epsilon* should evaluate to a value that approaches the actual maximum value.

For example,

```
(find-maximum-epsilon (lambda (x) (* x (- 5.5 x))) 1 10 1)
```

evaluates to 7.5. And,

```
(find-maximum-epsilon (lambda (x) (* x (- 5.5 x))) 1 10 0.01)
```

evaluates to 7.5625.

Solution. We start from the *find-maximum* definition from the example, and add an extra parameter:

```
(define (find-maximum-epsilon f low high epsilon)
  (if (>= low high)
      (f low)
      (bigger (f low) (find-maximum-epsilon f (+ low epsilon) high epsilon))))
```

The most important change is replacing `=` in the if expression predicate with `>=`. Otherwise, it is possible the exact matching value is skipped and the procedure will continue to evaluate forever without every reaching the base case!

Exercise 4.9. [★] The *find-maximum* procedure we defined evaluates to the maximum value of the input function in the range, but does not provide the input value that produces that maximum output value. Define a procedure that finds the input in the range that produces the maximum output value. For example, *(find-maximum-input inc 1 10)* should evaluate to 10 and *(find-maximum-input (lambda (x) (* x (- 5.5 x))) 1 10)* should evaluate to 3.

Solution. `(define (find-maximum-input f low high) (define (bigger-input a b) (if (> (f a) (f b)) a b))`

`(if (= low high) low (bigger-input low (find-maximum-input f (+ low 1) high))))`

Exercise 4.10. [★] Define a *find-area* procedure that takes as input a function *f*, a low range value *low*, a high range value *high*, and an increment *epsilon*, and produces as output an estimate for the area under the curve produced by the function *f* between *low* and *high* using the *epsilon* value to determine how many regions to evaluate.

Solution. We estimate the area under the curve by summing the areas of each trapezoid formed by the points $(x, 0)$, $(x, f(x))$, $(x + \epsilon, 0)$, $(x + \epsilon, f(x + \epsilon))$. The area of a trapezoid is its length times its average height:

$$\epsilon \times \frac{(f(x + \epsilon) - f(x))}{2}.$$

```
(define (find-area f low high epsilon)
  (if (>= low high)
      0
      (+ (* (/ (+ (f low) (f (+ low epsilon))) 2) epsilon)
         (find-area f (+ low epsilon) high epsilon))))
```

Here are some examples:

```
> (find-area (lambda (x) 5) 0 5 0.001)
24.99999999999931
> (find-area (lambda (x) x) 0 5 0.001)
12.49999999999935
> (find-area (lambda (x) (sin x)) 0 (* 2 pi) 0.0001)
1.0372978290178584e-010
```

The answers are not exact for two reasons. The first is that *epsilon* is not infinitesimal (and never can be with a finite computation). The second is a minor bug in the code since it does not account for the situation where `(+ low epsilon)` exceeds *high*. Fixing this problem is left as (another) exercise for the reader.

Exercise 4.11. Show the structure of the *gcd-euclid* applications in evaluating *(gcd-euclid 6 9)*.

Solution. The first application is *(gcd-euclid 6 9)*. Since the predicate is false, the alternate clause is evaluated. It leads to the application *(gcd-euclid 9 6)*. In this application, the predicate

is still false, and the alternate clause is evaluated. Since $(\text{modulo } 9 \ 6)$ evaluates to 3, this results in the application $(\text{gcd-euclid } 6 \ 3)$. For this application, the predicate is $(= (\text{modulo } 6 \ 3) \ 0)$ which is *true*. Hence, the expression evaluates to the consequence clause, b which has the value 3.

Exercise 4.12. [★] Provide a convincing argument why the evaluation of $(\text{gcd-euclid } a \ b)$ will always finish when the inputs are both positive integers.

Solution.

Exercise 4.13. Provide an alternate definition of *factorial* that is tail recursive. To be tail recursive, the expression containing the recursive application cannot be part of another application expression. (Hint: define a *factorial-helper* procedure that takes an extra parameter, and then define *factorial* as $(\text{define } (\text{factorial } n) (\text{factorial-helper } n \ 1))$.)

Solution. Following the hint, we add an extra parameter to keep track of the working result:

```
(define (factorial n)
  (define (factorial-helper n v)
    (if (= n 1) v
        (factorial-helper (- n 1) (* v n))))
  (factorial-helper n 1))
```

Exercise 4.14. Provide a tail recursive definition of *find-maximum*.

Solution.

```
(define (find-maximum-tail f low high)
  (define (find-maximum-helper f low high best)
    (if (= low high)
        (bigger (f low) best)
        (find-maximum-helper f (+ low 1) high (bigger (f low) best))))
  (find-maximum-helper f low high (f low)))
```

Exercise 4.15. [★★] Provide a convincing argument why it is possible to transform any recursive procedure into an equivalent procedure that is tail recursive.

Solution.

Exercise 4.16. This exercise tests your understanding of the $(\text{factorial } 2)$ evaluation.

- a. In step 5, the second part of the application evaluation rule, Rule 3(b), is used. In which step does this evaluation rule complete?
- b. In step 11, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?
- c. In step 25, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?
- d. To evaluate $(\text{factorial } 3)$, how many times would Evaluation Rule 2 be used to evaluate the name *factorial*?
- e. [★] To evaluate $(\text{factorial } n)$ for any positive integer n , how many times would Evaluation Rule 2 be used to evaluate the name *factorial*?

Exercise 4.17. For which input values n will an evaluation of (*factorial* n) eventually reach a value? For values where the evaluation is guaranteed to finish, make a convincing argument why it must finish. For values where the evaluation would not finish, explain why.

5

Data

Exercise 5.1. Describe the type of each of these expressions.

a. 17

Solution. Number

b. $(\text{lambda } (a) (> a 0))$

Solution. A procedure of type: $\text{Number} \rightarrow \text{Boolean}$

c. $((\text{lambda } (a) (> a 0)) 3)$

Solution. Boolean

d. $(\text{lambda } (a) (\text{lambda } (b) (> a b)))$

Solution. A procedure of type: $\text{Number} \rightarrow (\text{Number} \rightarrow \text{Boolean})$. That is, the result of applying this procedure to a Number would be a procedure that takes a Number as input and outputs a Boolean.

e. $(\text{lambda } (a) a)$

Solution. $T \rightarrow T$. A procedure that takes any type as its input, and produces as its output a value of the same type as the input.

Exercise 5.2. Define or identify a procedure that has the given type.

a. $\text{Number} \times \text{Number} \rightarrow \text{Boolean}$

Solution. $>$ (the built-in greater-than procedure takes two Numbers as inputs and outputs a Boolean)

b. $\text{Number} \rightarrow \text{Number}$

Solution. $-$ (the built-in negation procedure takes one Number input and outputs a Number)

c. $(\text{Number} \rightarrow \text{Number}) \times (\text{Number} \rightarrow \text{Number}) \rightarrow (\text{Number} \rightarrow \text{Number})$

Solution. The *fcompose* function from Section 4.2.1 takes as inputs two functions from Number to Number, and outputs a function that takes a Number as input and outputs a Number.

$(\text{lambda } (f\ g) (\text{lambda } (x) (g\ (f\ x))))$

d. $\text{Number} \rightarrow (\text{Number} \rightarrow (\text{Number} \rightarrow \text{Number}))$

Solution.

$(\text{lambda } (a) (\text{lambda } (b) (\text{lambda } (c) (+ a\ b\ c))))$

Exercise 5.3. Suppose the following definition has been executed:

```
(define tpair
  (cons (cons (cons 1 2) (cons 3 4))
        5))
```

Draw the structure defined by *tpair*, and give the value of each of the following expressions.

a. *(cdr tpair)*

Solution.

b. *(car (car (car tpair)))*

Solution.

c. *(cdr (cdr (car tpair)))*

Solution.

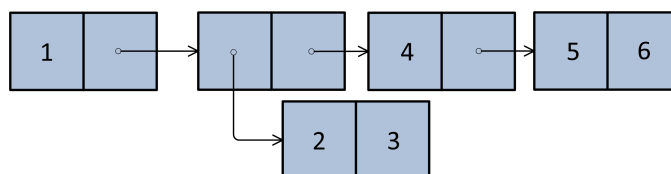
d. *(car (cdr (cdr tpair)))*

Solution.

Exercise 5.4. Write expressions that extract each of the four elements from *fstruct* defined by *(define fstruct (cons 1 (cons 2 (cons 3 4))))*.

Solution.

Exercise 5.5. Give an expression that produces the structure shown below.



Solution.

Exercise 5.6. Convince yourself the definitions of *scons*, *scar*, and *scdr* above work as expected by following the evaluation rules to evaluate

```
(scar (scons 1 2))
```

Solution.

Exercise 5.7. Show the corresponding definitions of *tcar* and *tcd* that provide the pair selection behavior for a pair created using *tcons* defined as:

```
(define (tcons a b) (lambda (w) (if w b a)))
```

Solution.

Exercise 5.8. Define a procedure that constructs a quintuple and procedures for selecting the five elements of a quintuple.

Solution.

Exercise 5.9. Another way of thinking of a triple is as a Pair where the first cell is a Pair and the second cell is a scalar. Provide definitions of *make-triple*, *triple-first*, *triple-second*, and *triple-third* for this construct.

Solution.

Exercise 5.10. For each of the following expressions, explain whether or not the expression evaluates to a List. Check your answers with a Scheme interpreter by using the *list?* procedure.

- a. *null*
- b. *(cons 1 2)*
- c. *(cons null null)*
- d. *(cons (cons (cons 1 2) 3) null)*
- e. *(cdr (cons 1 (cons 2 (cons null null))))*
- f. *(cons (list 1 2 3) 4)*

Solution.

Exercise 5.11. Define a procedure *is-list?* that takes one input and outputs true if the input is a List, and false otherwise. Your procedure should behave identically to the built-in *list?* procedure, but you should not use *list?* in your definition.

Solution.

Exercise 5.12. Define a procedure *list-max* that takes a List of non-negative numbers as its input and produces as its result the value of the greatest element in the List (or 0 if there are no elements in the input List). For example, *(list-max (list 1 1 2 0))* should evaluate to 2.

Solution.

Exercise 5.13. Use *list-accumulate* to define *list-max* (from Exercise 5.12).

Solution.

Exercise 5.14. [★] Use *list-accumulate* to define *is-list?* (from Exercise 5.11).

Solution.

Exercise 5.15. Define a procedure *list-last-element* that takes as input a List and outputs the last element of the input List. If the input List is empty, *list-last-element* should produce an error.

Solution.

Exercise 5.16. Define a procedure *list-ordered?* that takes two inputs, a test procedure and a List. It outputs true if all the elements of the List are ordered according to the test procedure. For example, *(list-ordered? < (list 1 2 3))* evaluates to true, and *(list-ordered? < (list 1 2 3 2))* evaluates to false. Hint: think about what the output should be for the empty list.

Solution.

Exercise 5.17. Define a procedure *list-increment* that takes as input a List of numbers, and produces as output a List containing each element in the input List incremented by one. For example, (*list-increment* 1 2 3) evaluates to (2 3 4).

Solution.

Exercise 5.18. Use *list-map* and *list-sum* to define *list-length*:

```
(define (list-length p) (list-sum (list-map ____ p)))
```

Solution.

Exercise 5.19. Define a procedure *list-filter-even* that takes as input a List of numbers and produces as output a List consisting of all the even elements of the input List.

Solution.

Exercise 5.20. Define a procedure *list-remove* that takes two inputs: a test procedure and a List. As output, it produces a List that is a copy of the input List with all of the elements for which the test procedure evaluates to true removed. For example, (*list-remove* (**lambda** (x) (= x 0)) (*list* 0 1 2 3)) should evaluate to the List (1 2 3).

Solution.

Exercise 5.21. [★★] Define a procedure *list-unique-elements* that takes as input a List and produces as output a List containing the unique elements of the input List. The output List should contain the elements in the same order as the input List, but should only contain the first appearance of each value in the input List.

Solution.

Exercise 5.22. Define the *list-reverse* procedure using *list-accumulate*.

Exercise 5.23. Define *factorial* using *intsto.factorial*

Solution.

Exercise 5.24. [★] Define a procedure *deep-list-map* that behaves similarly to *list-map* but on deeply nested lists. It should take two parameters, a mapping procedure, and a List (that may contain deeply nested Lists as elements), and output a List with the same structure as the input List with each value mapped using the mapping procedure.

Solution.

Exercise 5.25. [★] Define a procedure *deep-list-filter* that behaves similarly to *list-filter* but on deeply nested lists.

Solution.

6

Machines

Solutions for Chapter 6 provided by Ervin Varga.

Exercise 6.1. Babbage's design for the Analytical Engine called for a store holding 1000 variables, each of which is a 50-digit (decimal) number. How many bits could the store of Babbage's Analytical Engine hold?

Solution. With 50 decimal digits we can differentiate 10^{50} numbers covering the range $[0, 10^{50} - 1]$. Thus, we need $\lceil \log_2 10^{50} \rceil = 167$ bits per variable. Thus, The machine could store 167,000 bits in total, which is 20,875 bytes (or approximately 20KB).

Exercise 6.2. Define a Scheme procedure, *logical-or*, that takes two inputs and outputs the logical or of those inputs.

Solution.

(define (logical-or a b) (if a true b))

Exercise 6.3. What is the meaning of composing *not* with itself? For example, $(not (not A))$.

Solution. It is an identity function: for any A , $(not(not A)) = A$.

Exercise 6.4. Define the *xor* function using only *nand* functions.

Solution. $(xor A B) \equiv (and (or A B) (nand A B))$. The book already describes how to express *and*, *not*, and *or* in terms of *nand*, so we can plug in those definition to obtain a full definition of *xor* using only *nand*.

Exercise 6.5. [★] Our definition of $(not A)$ as $(nand A A)$ assumes there is a way to produce two copies of a given input. Design a component for our wine machine that can do this. It should take one input, and produce two outputs, both with the same value as the input. (Hint: when the input is *true*, we need to produce two full bottles as outputs, so there must be a source similarly to the *not* component.)

Solution. It is very similar to the *not* machine. The difference is that the two nozzles are at the bottom of the basin (below the float) and fill two bottles of wine. The source's outlet is also at the bottom blocked by the float when the input is empty. If the input is a full bottle, then the float would unblock the source and fill the two bottles.

Exercise 6.6. [★] The digital abstraction works fine as long as actual values stay close to the

value they represent. But, if we continue to compute with the outputs of functions, the actual values will get increasingly fuzzy. For example, if the inputs to the *and3* function in Figure 6.3 are initially all $\frac{3}{4}$ full bottles (which should be interpreted as *true*), the basin for the first *and* function will fill to $1\frac{1}{2}$, so only $\frac{1}{2}$ bottle will be output from the first *and*. When combined with the third input, the second basin will contain $1\frac{1}{4}$ bottles, so only $\frac{1}{4}$ will spill into the output bottle. Thus, the output will represent *false*, even though all three inputs represent *true*. The solution to this problem is to use an *amplifier* to restore values to their full representations. Design a wine machine amplifier that takes one input and produces a strong representation of that input as its output. If that input represents *true* (any value that is half full or more), the amplifier should output *true*, but with a strong, full bottle representation. If that input represents *false* (any value that is less than half full), the amplifier should output a strong *false* value (completely empty).

Solution. This is a use case for the identify function from Exercise 6.3: two *not* machines should be combined to produce the desired amplifier.

Exercise 6.7. Adding logically.

a. What is the logical formula for r_3 ?

Solution.

$$\begin{aligned}
 r_2 &= (\text{xor } (\text{xor } a_2 \ b_2) \ c_2) \\
 &= (\text{xor } (\text{xor } a_2 \ b_2) \\
 &\quad (\text{or } (\text{and } a_1 \ b_1) (\text{and } a_1 (\text{and } a_0 \ b_0)) (\text{and } b_1 (\text{and } a_0 \ b_0)))) \\
 c_3 &= (\text{or } (\text{and } a_2 \ b_2) (\text{and } a_2 \ c_2) (\text{and } b_2 \ c_2)) \\
 &= (\text{or } (\text{and } a_2 \ b_2) \\
 &\quad (\text{and } a_2 (\text{or } (\text{and } a_1 \ b_1) (\text{and } a_1 (\text{and } a_0 \ b_0)) (\text{and } b_1 (\text{and } a_0 \ b_0)))) \\
 &\quad (\text{and } b_2 (\text{or } (\text{and } a_1 \ b_1) (\text{and } a_1 (\text{and } a_0 \ b_0)) (\text{and } b_1 (\text{and } a_0 \ b_0)))))) \\
 r_3 &= (\text{xor } (\text{xor } a_3 \ b_3) \ c_3) \\
 &= (\text{xor } (\text{xor } a_3 \ b_3) \\
 &\quad (\text{or } (\text{and } a_2 \ b_2) \\
 &\quad (\text{and } a_2 (\text{or } (\text{and } a_1 \ b_1) (\text{and } a_1 (\text{and } a_0 \ b_0)) (\text{and } b_1 (\text{and } a_0 \ b_0)))) \\
 &\quad (\text{and } b_2 (\text{or } (\text{and } a_1 \ b_1) (\text{and } a_1 (\text{and } a_0 \ b_0)) (\text{and } b_1 (\text{and } a_0 \ b_0)))))) \\
 c_4 &= (\text{or } (\text{and } a_3 \ b_3) (\text{and } a_3 \ c_3) (\text{and } b_3 \ c_3)) \\
 r_4 &= (\text{xor } (\text{xor } a_4 \ b_4) \ c_4)
 \end{aligned}$$

b. Without simplification, how many functions will be composed to compute the addition result bit r_4 ?

Solution. The number of functions (without simplification) for c_k is $4 + 2c_{k-1}$ for $k > 1$, and $c_1 = 1$. Thus, $c_2 = 6$, $c_3 = 16$, and $c_4 = 36$.

The number of functions (without simplification) for r_k is $2 + c_k$ for $k > 0$. Thus, $r_1 = 3$, $r_2 = 8$, $r_3 = 18$, and $r_4 = 38$.

c. [★] Is it possible to compute r_4 with fewer logical functions?

Solution. Yes — we can considerably simplify the expression of r_4 by observing that we can get:

$$\begin{aligned}
 c_2 &= (\text{or } (\text{and } a_1 \ b_1) (\text{and } c_1 (\text{or } a_1 \ b_1))) \\
 &= (\text{or } (\text{and } a_1 \ b_1) (\text{and } (\text{and } a_0 \ b_0) (\text{or } a_1 \ b_1)))
 \end{aligned}$$

Or more generally:

$$ck = (\text{or } (\text{and } ak \ bk) \ (\text{and } c[k-1] \ (\text{or } ak \ bk)))$$

We may simplify both c_3 and c_4 via the previous template. We'll leave proving the minimal number of operations as a [**] exercise.

Exercise 6.8. Show how to compute the result bits for binary multiplication of two 2-bit inputs using only logical functions.

Solution. The multiplication of two 2-bits numbers is comprised from one-bit multiplications and additions. Suppose we multiply a_1a_0 with b_1b_0 . We get the following result bits (the carry bits are presented for better clarity):

$$r0 = (\text{and } a0 \ b0)$$

$$c0 = 0$$

$$r1 = (\text{xor } (\text{and } a0 \ b1) \ (\text{and } a1 \ b0))$$

$$c1 = (\text{and } (\text{and } a0 \ b1) \ (\text{and } a1 \ b0))$$

$$r2 = (\text{xor } (\text{and } a1 \ b1) \ c1) = (\text{xor } (\text{and } a1 \ b1) \ (\text{and } (\text{and } a0 \ b1) \ (\text{and } a1 \ b0)))$$

$$c2 = (\text{and } (\text{and } a1 \ b1) \ c1) = (\text{and } (\text{and } a1 \ b1) \ (\text{and } (\text{and } a0 \ b1) \ (\text{and } a1 \ b0)))$$

$$r3 = c2 = (\text{and } (\text{and } a1 \ b1) \ c1) = (\text{and } (\text{and } a1 \ b1) \ (\text{and } (\text{and } a0 \ b1) \ (\text{and } a1 \ b0)))$$

Exercise 6.9. [★] Show how to compute the result bits for binary multiplication of two inputs of any length using only logical functions.

Solution.

The number of logical functions needed is quite large even for a tiny example of multiplying 2-bit by 2-bit numbers. In a general case, we cannot use such naïve approach. A classical two n -bit multiplier applies a shifter and accumulator to sum up partial products (this is known as shift-and-add multiplication). Modern architectures are much faster and employ quite advanced algorithms and hardware. You may want to look up Wallace Trees and Dadda Multipliers to get some ideas how to do this.

Exercise 6.10. Follow the rules to simulate the checking parentheses Turing Machine on each input (assume the beginning and end of the input are marked with a #):

a.)

Solution.

The descriptions below reference the rules from Figure 6.5. Assume the rules are numbered from top to bottom starting with 1. In all use cases below, the machine starts in *LookForClosing* state, the head is above the start symbol #, and it moves to the right one square.

To process) we use rules 1 (*Found closing*) and 8 (*Reached beginning, Halt*).

b. ()

Solution. 2, 1, 6, 3, 4, 11, 11, 12.

c. *empty input*

Solution. 4, 12.

d. $((()((())))())$

Solution. 2, 2, 1, 6, 3, 2, 2, 1, 6, 3, 1, 7, 7, 6, 3, 3, 3, 1, 7, 7, 7, 7, 7, 7, 6, 3, 3, 3, 3, 3, 3, 2, 1, 6, 3, 4, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12.

e. $((()))((()((())))())$

Solution. 2, 2, 1, 6, 3, 1, 7, 7, 6, 3, 3, 3, 1, 7, 7, 7, 7, 8.

Exercise 6.11. [★] Design a Turing Machine for adding two arbitrary-length binary numbers. The input is of the form $a_{n-1} \dots a_1 a_0 + b_{m-1} \dots b_1 b_0$ (with # markers at both ends) where each a_k and b_k is either 0 or 1. The output tape should contain bits that represent the sum of the two inputs.

Solution.

There are many strategies that can work for this, but its best to break the problem into a few sub-tasks like finding the least significant remaining digit, adding two bits (with a stored carry in the state), moving to the next position for writing the output, etc. For a problem this complicated, there are many on-line Turing Machine simulators to use to develop and test your solution.

7

Cost

Solutions for Chapter 7 provided by Ervin Varga.

Exercise 7.1. Suppose you are defining a procedure that needs to append two lists, one short list, *short* and one very long list, *long*, but the order of elements in the resulting list does not matter. Is it better to use *(list-append short long)* or *(list-append long short)*? (A good answer will involve both experimental results and an analytical explanation.)

Solution. It is faster to evaluate *(list-append short long)*, since the procedure iterates over the first argument and appends the second one once the first has been exhausted.

Here are some timings to support this statement:

```
> (define long-list (intsto 30000))
> (define short-list (intsto 100))
> (time (car (list-append long-list short-list)))
cpu time: 30 real time: 30 gc time: 21
> (time (car (list-append short-list long-list)))
cpu time: 1 real time: 0 gc time: 0
```

Exercise 7.2. For each of the g functions below, answer whether or not g is in the set $O(n)$. Your answer should include a proof. If g is in $O(n)$ you should identify values of c and n_0 that can be selected to make the necessary inequality hold. If g is not in $O(n)$ you should argue convincingly that no matter what values are chosen for c and n_0 there are values of $n \geq n_0$ such the inequality in the definition of O does not hold.

Solution. In all of these cases we need to analyze whether $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$, for arbitrary constants c and n_0 .

a. $g(n) = n + 5$

Solution. $g(n)$ is in $O(n)$: choose $c = 5$ and $n_0 = 2$. Then, $n + 5 \leq 5n$ for all $n \geq 2$.

b. $g(n) = .01n$

Solution. $g(n)$ is in $O(n)$: Choose $c = 1$ and $n_0 = 1$. Then, $0.01n \leq n$ for all $n \geq 1$.

c. $g(n) = 150n + \sqrt{n}$

Solution. $g(n)$ is in $O(n)$: choose $c = 151$ and $n_0 = 1$. Then, $150n + \sqrt{n} \leq 151n$ for all $n \geq 1$.

d. $g(n) = n^{1.5}$

Solution. $g(n)$ is **not** in $O(n)$. For any value of c , we can make $n^{1.5} = n \cdot \sqrt{n} > cn$ by choosing $n > c^2$.

e. $g(n) = n!$

Solution. $g(n)$ is **not** in $O(n)$. For any value of c , we can make $n! = n \cdot (n-1) \dots 2 \cdot 1 \Rightarrow cn$ by choosing $n > c + 1$. Notice that in this case $n - 1 > c$ (second term above).

Exercise 7.3. [★] Given f is some function in $O(h)$, and g is some function not in $O(h)$, which of the following must always be true:

Solution. In all of these cases we know that the following is true for some c_1, c_2, n_0 , and m_0 : $f(n) \leq c_1 h(n)$ when $n \geq n_{0f}$, and $h(n) \leq c_2 g(n)$ when $n \geq n_{0g}$. Saying that g is not in $O(h)$ implies that h is in $O(g)$. Again, it doesn't matter whether $O(g)$ is a reasonable upper bound, i.e., we can say that $n + 1$ is in $O(n!)$, although such an upper bound is not useful.

a. For all positive integers m , $f(m) \leq g(m)$.

Solution. Not always true. To prove this, it is enough to find a single counterexample. Suppose $f = m + 10$, $h = m$, and $g = 0.1m^2$. Obviously, $m + 10$ is in $O(m)$ and $0.1m^2$ is not in $O(m)$. Nevertheless, $m + 10 > 0.1m^2$ for $m = 1$, hence the original statement doesn't apply for all positive integers m .

b. For some positive integer m , $f(m) < g(m)$.

Solution. Always true. Recall that g is not in $O(h)$, so at some value m , $g(m)$ will become strictly larger than $ch(m)$ irrespective of the chosen constant c . On the other hand, we know that $f(m)$ is bounded by $h(m)$ for some constant c and for all $n > n_0$ for some constant n_0 . Consequently, $f(m)$ must be smaller than $g(m)$ for some input m .

c. For some positive integer m_0 , and all positive integers $m > m_0$,

$$f(m) < g(m).$$

Solution. Always true. Intuitively, since $g(m) \notin O(h)$ it grows faster than h , while since $f(m) \in O(h)$ it grows no faster than h , so eventually $g(m)$ must exceed $f(m)$. More formally, it is enough to choose $m_0 = \max(m_{0f}, m_{0g})$ from the two following expressions: $f(m) \leq c_1 h(m)$, for some c_1 when $m \geq m_{0f}$ since $f \in O(h)$; and $g(m) \leq c_1 h(m)$, when $m \geq m_{0g}$ since $g \notin O(h)$. Note that the second inequality must hold for some m_{0g} since $g \notin O(h)$. Eventually, $g(m)$ must become larger than $c_1 h(m)$ for any constant c_1 .

Exercise 7.4. Repeat Exercise 7.2 using Ω instead of O .

In all of these cases we need to analyze whether $g(n) \geq cf(n)$ for all $n \geq n_0$ for any constants c and n_0 .

a. $g(n) = n + 5$

Solution. $g(n)$ is in $\Omega(n)$: choose $c = 1$ and $n_0 = 1$. Then, $n + 5 \geq n$ for all $n \geq 1$.

b. $g(n) = .01n$

Solution. $g(n)$ is in $\Omega(n)$: Choose $c = 0.01$ and $n_0 = 1$. Then, $0.01n \leq 0.01n$ for all $n \geq 1$.

c. $g(n) = 150n + \sqrt{n}$

Solution. $g(n)$ is in $\Omega(n)$: choose $c = 1$ and $n_0 = 1$. Then, $150n + \sqrt{n} \geq n$ for all $n \geq 1$.

d. $g(n) = n^{1.5}$

Solution. $g(n)$ is in $\Omega(n)$: choose $c = 1$ and $n_0 = 1$, then $n^{1.5} = n\sqrt{n} \geq n$ for all $n \geq 1$.

e. $g(n) = n!$

Solution. $g(n)$ is in $\Omega(n)$: choose $c = 1$ and $n_0 = 1$, then $n! \geq n$ for all $n \geq 1$.

Exercise 7.5. For each part, identify a function g that satisfies the property.

a. g is in $O(n^2)$ but not in $\Omega(n^2)$.

Solution. $g(n) = n$ is in $O(n^2)$ but not in $\Omega(n^2)$.

b. g is not in $O(n^2)$ but is in $\Omega(n^2)$.

Solution. $g(n) = n^3$ is not in $O(n^2)$ but is in $\Omega(n^2)$.

c. g is in both $O(n^2)$ and $\Omega(n^2)$.

Solution. $g(n) = n^2$

Exercise 7.6. Repeat Exercise 7.2 using Θ instead of O .

Solution. This follows from our previous results — to be in $\Theta(n)$, $g(n)$ must be in both $O(n)$ and $\Omega(n)$. So, (a) $g(n) = n + 5$, (b) $g(n) = 0.01n$, and (c) $g(n) = 150n + \sqrt{n}$ are all in $\Theta(n)$, but (d) $g(n) = n^{1.5}$ and (e) $g(n) = n!$ are not since they are not in $O(n)$.

Exercise 7.7. Show that $\Theta(n^2 - n)$ is equivalent to $\Theta(n^2)$.

Exercise 7.8. [★] Is $\Theta(n^2)$ equivalent to $\Theta(n^{2.1})$? Either prove they are identical, or prove they are different.

Exercise 7.9. [★] Is $\Theta(2^n)$ equivalent to $\Theta(3^n)$? Either prove they are identical, or prove they are different.

Exercise 7.10. Explain why the *list-map* procedure from Section 5.4.1 has running time that is linear in the size of its List input. Assume the procedure input has constant running time.

Solution.

Exercise 7.11. Consider the *list-sum* procedure (from Example 5.2):

(define (list-sum p) (if (null? p) 0 (+ (car p) (list-sum (cdr p))))

What assumptions are needed about the elements in the list for the running time to be linear in the number of elements in the input list?

Solution.

Exercise 7.12. For the decimal six-digit odometer (shown in the picture on page 145), we measure the amount of work to add one as the total number of wheel digit turns required. For example, going from 000000 to 000001 requires one work unit, but going from 000099 to 000100 requires three work units.

- a. What are the worst case inputs?

Solution.

- b. What are the best case inputs?

Solution.

- c. [★] On average, how many work units are required for each mile? Assume over the lifetime of the odometer, the car travels 1,000,000 miles.

Solution.

- d. Lever voting machines were used by the majority of American voters in the 1960s, although they are not widely used today. Most lever machines used a three-digit odometer to tally votes. Explain why candidates ended up with 99 votes on a machine far more often than 98 or 100 on these machines.

Solution.

Exercise 7.13. [★] The *list-get-element* argued by comparison to $+$, that the $-$ procedure has constant running time when one of the inputs is a constant. Develop a more convincing argument why this is true by analyzing the worst case and average case inputs for $-$.

Solution.

Exercise 7.14. [★] Our analysis of the work required to add one to a number argued that it could be done in constant time. Test experimentally if the DrRacket $+$ procedure actually satisfies this property. Note that one $+$ application is too quick to measure well using the *time* procedure, so you will need to design a procedure that applies $+$ many times without doing much other work.

Solution.

Exercise 7.15. [★] Analyze the running time of the elementary school long division algorithm.

Solution.

Exercise 7.16. [★] Define a Scheme procedure that multiplies two multi-digit numbers (without using the built-in $*$ procedure except to multiply single-digit numbers). Strive for your procedure to have running time in $\Theta(n)$ where n is the total number of digits in the input numbers.

Solution.

Exercise 7.17. [★★★★] Devise an asymptotically faster general multiplication algorithm than Fürer's, or prove that no faster algorithm exists.

Solution.

Exercise 7.18. Analyze the asymptotic running time of the *list-sum* procedure (from Example 5.2):

```
(define (list-sum p)
  (if (null? p)
      0
      (+ (car p) (list-sum (cdr p)))))
```

You may assume all of the elements in the list have values below some constant (but explain why this assumption is useful in your analysis).

Solution.

Exercise 7.19. Analyze the asymptotic running time of the *factorial* procedure (from Example 4.1):

```
(define (factorial n) (if (= n 0) 1 (* n (factorial (- n 1)))))
```

Be careful to describe the running time in terms of the *size* (not the magnitude) of the input.

Solution.

Exercise 7.20. Consider the *intsto* problem (from Example 5.8).

a. [★] Analyze the asymptotic running time of this *intsto* procedure:

```
(define (revintsto n)
  (if (= n 0)
      null
      (cons n (revintsto (- n 1)))))
(define (intsto n) (list-reverse (revintsto n)))
```

b. [★] Analyze the asymptotic running time of this *instto* procedure:

```
(define (intsto n)
  (if (= n 0) null (list-append (intsto (- n 1)) (list n))))
```

c. Which version is better?

d. [★★] Is there an asymptotically faster *intsto* procedure?

Solution.

Exercise 7.21. Analyze the running time of the *board-replace-peg* procedure (from Exploration 5.2):peg-board puzzle

```
(define (row-replace-peg pegs col val)
  (if (= col 1) (cons val (cdr pegs))
      (cons (car pegs) (row-replace-peg (cdr pegs) (- col 1) val))))
(define (board-replace-peg board row col val)
  (if (= row 1) (cons (row-replace-peg (car board) col val) (cdr board))
      (cons (car board) (board-replace-peg (cdr board) (- row 1) col val))))
```

Solution.

Exercise 7.22. Analyze the running time of the *deep-list-flatten* procedure from Section 5.5:

```
(define (deep-list-flatten p)
  (if (null? p) null
      (list-append (if (list? (car p))
                        (deep-list-flatten (car p))
                        (list (car p)))
                    (deep-list-flatten (cdr p)))))
```

Solution.

Exercise 7.23. [★] Find and correct at least one error in the *Orders of Growth* section of the Wikipedia page on *Analysis of Algorithms* (http://en.wikipedia.org/wiki/Analysis_of_algorithms). This is rated as [★] now (July 2011), since the current entry contains many fairly obvious errors. Hopefully it will soon become a [★★★] challenge, and perhaps, eventually will become impossible!

Solution.

8

Sorting and Searching

Exercise 8.1. What is the best case input for *list-sort-best-first*? What is its asymptotic running time on the best case input?

Solution.

Exercise 8.2. Use the *time* special form (Section 7.1) to experimentally measure the evaluation times for the *list-sort-best-first-let* procedure. Do the results match the expected running times based on the $\Theta(n^2)$ asymptotic running time?

You may find it helpful to define a procedure that constructs a list containing n random elements. To generate the random elements use the built-in procedure *random* that takes one number as input and evaluates to a pseudorandom number between 0 and one less than the value of the input number. Be careful in your time measurements that you do not include the time required to generate the input list.random

Solution.

Exercise 8.3. Define the *list-find-best* procedure using the *list-accumulate* procedure from Section 5.4.2 and evaluate its asymptotic running time.

Solution.

Exercise 8.4. [★] Define and analyze a *list-sort-worst-last* procedure that sorts by finding the worst element first and putting it at the end of the list.

Solution.

Exercise 8.5. We analyzed the worst case running time of *list-sort-insert* above. Analyze the best case running time. Your analysis should identify the inputs for which *list-sort-insert* runs fastest, and describe the asymptotic running time for the best case input.

Solution.

Exercise 8.6. Both the *list-sort-best-first-sort* and *list-sort-insert* procedures have asymptotic running times in $\Theta(n^2)$. This tells us how their worst case running times grow with the size of the input, but isn't enough to know which procedure is faster for a particular input. For the questions below, use both analytical and empirical analysis to provide a convincing answer.

- a. How do the actual running times of *list-sort-best-first-sort* and *list-sort-insert* on typical inputs compare?

Solution.

- b.** Are there any inputs for which *list-sort-best-first* is faster than *list-sort-insert*?

Solution.

- c.** For sorting a long list of n random elements, how long does each procedure take? (See Exercise 8.2 for how to create a list of random elements.)

Solution.

Exercise 8.7. Define a procedure *binary-tree-size* that takes as input a binary tree and outputs the number of elements in the tree. Analyze the running time of your procedure.

Solution.

Exercise 8.8. [★] Define a procedure *binary-tree-depth* that takes as input a binary tree and outputs the depth of the tree. The running time of your procedure should not grow faster than linearly with the number of nodes in the tree.

Solution.

Exercise 8.9. [★★] Define a procedure *binary-tree-balance* that takes as input a sorted binary tree and the comparison function, and outputs a sorted binary tree containing the same elements as the input tree but in a well-balanced tree. The depth of the output tree should be no higher than $\log_2 n + 1$ where n is the number of elements in the input tree.

Solution.

Exercise 8.10. Estimate the time it would take to sort a list of one million elements using *list-quicksort*.

Solution.

Exercise 8.11. Both the *list-quicksort* and *list-sort-tree-insert* procedures have expected running times in $\Theta(n \log n)$. Experimentally compare their actual running times.

Solution.

Exercise 8.12. What is the best case input for *list-quicksort*? Analyze the asymptotic running time for *list-quicksort* on best case inputs.

Solution.

Exercise 8.13. [★] Instead of using binary trees, we could use ternary trees. A node in a ternary tree has two elements, a left element and a right element, where the left element must be before the right element according to the comparison function. Each node has three subtrees: *left*, containing elements before the left element; *middle*, containing elements between the left and right elements; and *right*, containing elements after the right element. Is it possible to sort faster using ternary trees?

Solution.

Exercise 8.14. Define a procedure for finding the longest word in a document. Analyze the running time of your procedure.

Solution.

Exercise 8.15. Produce a list of the words in Shakespeare's plays sorted by their length.

Solution.

Exercise 8.16. [★] Analyze the running time required to build the index.

- a. Analyze the running time of the *text-to-word-positions* procedure. Use n to represent the number of characters in the input string, and w to represent the number of distinct words. Be careful to clearly state all assumptions on which your analysis relies.

Solution.

- b. Analyze the running time of the *insert-into-index* procedure.

Solution.

- c. Analyze the running time of the *index-document* procedure.

Solution.

- d. Analyze the running time of the *merge-indexes* procedure.

Solution.

- e. Analyze the overall running time of the *index-pages* procedure. Your result should describe how the running time is impacted by the number of documents to index, the size of each document, and the number of distinct words.

Solution.

Exercise 8.17. [★] The *search-in-index* procedure does not actually have the expected running time in $\Theta(\log w)$ (where w is the number of distinct words in the index) for the Shakespeare index because of the way it is built using *merge-indexes*. The problem has to do with the running time of the binary tree on pathological inputs. Explain why the input to *list-to-sorted-tree* in the *merge-indexes* procedure leads to a binary tree where the running time for searching is in $\Theta(w)$. Modify the *merge-indexes* definition to avoid this problem and ensure that searches on the resulting index run in $\Theta(\log w)$.

Solution.

Exercise 8.18. [★★] The site <http://www.speechwars.com> provides an interesting way to view political speeches by looking at how the frequency of the use of different words changes over time. Use the *index-histogram* procedure to build a historical histogram program that takes as input a list of indexes ordered by time, and a target word, and output a list showing the number of occurrences of the target word in each of the indexes. You could use your program to analyze how Shakespeare's word use is different in tragedies and comedies or to compare Shakespeare's vocabulary to Jefferson's.

Solution.

9

Mutation

Exercise 9.1. Devise a Scheme expression that could have four possible values, depending on the order in which application subexpressions are evaluated.

Solution.

Exercise 9.2. Draw the environment that results after evaluating:

```
> (define alpha 0)
> (define beta 1)
> (define update-beta! (lambda () (set! beta (+ alpha 1))))
> (set! alpha 3)
> (update-beta!)
> (set! alpha 4)
```

Solution.

Exercise 9.3. Draw the environment that results after evaluating the following expressions, and explain what the value of the final expression is. (Hint: first, rewrite the let expression as an application.)

```
> (define (make-applier proc) (lambda (x) (proc x)))
> (define p (make-applier (lambda (x) (let ((x 2)) x))))
> (p 4)
```

Solution.

Exercise 9.4. What is the value of (*mlist-length pair*) for the pair shown in Figure 9.5?

Solution.

Exercise 9.5. [★] Define a *mpair-circular?* procedure that takes a MutablePair as its input and outputs true when the input contains a cycle and false otherwise.

Solution.

Exercise 9.6. Define an imperative-style procedure, *mlist-inc!* that takes as input a MutableList of Numbers and modifies the list by adding one to the value of each element in the list.

Solution.

Exercise 9.7. [★] Define a procedure *mlist-truncate!* that takes as input a MutableList and modifies the list by removing the last element in the list. Specify carefully the requirements for the input list to your procedure.

Solution.

Exercise 9.8. [★] Define a procedure *mlist-make-circular!* that takes as input a MutableList and modifies the list to be a circular list containing all the elements in the original list. For example, (*mlist-make-circular!* (*mlist* 3)) should produce the same structure as the circular pair shown in Figure 9.5.

Solution.

Exercise 9.9. [★] Define an imperative-style procedure, *mlist-reverse!*, that reverses the elements of a list. Is it possible to implement a *mlist-reverse!* procedure that is asymptotically faster than the *list-reverse* procedure from Example 5.4?

Solution.

Exercise 9.10. [★★] Define a procedure *mlist-aliases?* that takes as input two mutable lists and outputs true if and only if there are any mcons cells shared between the two lists.

Solution.

Exercise 9.11. Define the *mlist-map!* example from the previous section using *while*.

Solution.

Exercise 9.12. Another common imperative programming structure is a repeat-until—textbfrepeat-until loop. Define a *repeat-until* procedure that takes two inputs, a body procedure and a test procedure. The procedure should evaluate the body procedure repeatedly, until the test procedure evaluates to a true value. For example, using *repeat-until* we could define *factorial* as:

```
(define (factorial n)
  (let ((fact 1))
    (repeat-until
      (lambda () (set! fact (* fact n)) (set! n (- n 1)))
      (lambda () (< n 1)))
    fact))
```

Solution.

Exercise 9.13. [★★] Improve the efficiency of the indexing procedures from Section 8.2.3 by using mutation. Start by defining a mutable binary tree abstraction, and then use this and the MutableList data type to implement an imperative-style *insert-into-index!* procedure that mutates the input index by adding a new word-position pair to it. Then, define an efficient *merge-index!* procedure that takes two mutable indexes as its inputs and modifies the first index to incorporate all word occurrences in the second index. Analyze the impact of your changes on the running time of indexing a collection of documents.

Solution.

10

Objects

Exercise 10.1. Modify the *make-counter* definition to add a *previous!* method that decrements the counter value by one.

Solution.

Exercise 10.2. [★] Define a *variable-counter* object that provides these methods:

make-variable-counter: Number \rightarrow VariableCounter

Creates a *variable-counter* object with an initial counter value of 0 and an initial increment value given by the parameter.

set-increment!: Number \rightarrow Void

Sets the increment amount for this counter to the input value.

next!: Void \rightarrow Void

Adds the increment amount to the value of the counter.

get-count: Void \rightarrow Number

Outputs the current value of the counter.

Here are some sample interactions using a *variable-counter* object:

```
> (define vcounter (make-variable-counter 1))
> (ask vcounter 'next!)
> (ask vcounter 'set-increment! 2)
> (ask vcounter 'next!)
> (ask vcounter 'get-count)
3
```

Solution.

Exercise 10.3. Define a *countdown* class that simulates a rocket launch countdown: it starts at some initial value, and counts down to zero, at which point the rocket is launched. Can you implement *countdown* as a subclass of *counter*?

Solution.

Exercise 10.4. Define the *variable-counter* object from Exercise 10.2 as a subclass of *counter*.

Solution.

Exercise 10.5. Define a new subclass of *parameterizable-counter* where the increment for each

next! method application is a parameter to the constructor procedure. For example, (*make-parameterizable-counter* 0.1) would produce a counter object whose counter has value 0.1 after one invocation of the *next!* method.

Solution.

11

Interpreters

Exercise 11.1. Draw the parse tree for each of the following Python expressions and provide the value of each expression.

a. $1 + 2 + 3 * 4$

Solution.

b. $3 > 2 + 2$

Solution.

c. $3 * 6 >= 15 == 12$

Solution.

d. $(3 * 6 >= 15) == \text{True}$

Solution.

Exercise 11.2. Do comparison expressions have higher or lower precedence than addition expressions? Explain why using the grammar rules.

Solution.

Exercise 11.3. Define the sequence of factorials as an infinite list using delayed evaluation.

Solution.

Exercise 11.4. Describe the infinite list defined by each of the following definitions. (Check your answers by evaluating the expressions in LazyCharme.)

a. `(define p (cons 1 (merge-lists p p +)))`

Solution.

b. `(define t (cons 1 (merge-lists t (merge-lists t t +) +)))`

Solution.

c. `(define twos (cons 2 twos))`

Solution.

d. `(define doubles (merge-lists (ints-from 1) twos *))`

Solution.



Eratosthenes

Exercise 11.5. [★★] A simple procedure known as the *Sieve of Eratosthenes* for finding prime numbers was created by Eratosthenes, an ancient Greek mathematician and astronomer. The procedure imagines starting with an (infinite) list of all the integers starting from 2. Then, it repeats the following two steps forever:

1. Circle the first number that is not crossed off; it is prime.
2. Cross off all numbers that are multiples of the circled number.

To carry out the procedure in practice, of course, the initial list of numbers must be finite, otherwise it would take forever to cross off all the multiples of 2. But, with delayed evaluation, we can implement the Sieve procedure on an effectively infinite list.

Implement the sieve procedure using lists with lazy evaluation. You may find the *list-filter* and *merge-lists* procedures useful, but will probably find it necessary to define some additional procedures.

Solution.

12

Computability

Exercise 12.1. Is the Launches-Missiles Problem described below computable? Provide a convincing argument supporting your answer.

Launches-Missiles

Input: A specification of a procedure.

Output: If an application of the procedure would lead to the missiles being launched, outputs True. Otherwise, outputs False.

You may assume that the only thing that causes the missiles to be launched is an application of the *launchMissiles* procedure.

Solution.

Exercise 12.2. Is the Same-Result Problem described below computable? Provide a convincing argument supporting your answer.

Same-Result

Input: Specifications of two procedures, P and Q .

Output: If an application of P terminates and produces the same value as applying Q , outputs True. If an application of P does not terminate, and an application of Q also does not terminate, outputs True. Otherwise, outputs False.

Solution.

Exercise 12.3. Is the Check-Proof Problem described below computable? Provide a convincing argument supporting your answer.

Check-Proof

Input: A specification of an axiomatic system, a statement (the theorem), and a proof (a sequence of steps, each identifying the axiom that is applied).

Output: Outputs True if the proof is a valid proof of the theorem in the system, or False if it is not a valid proof.

Solution.

Exercise 12.4. Is the Find-Finite-Proof Problem described below computable? Provide a convincing argument supporting your answer.

Find-Finite-Proof

Input: A specification of an axiomatic system, a statement (the theorem), and a maximum number of steps (max-steps).

Output: If there is a proof in the axiomatic system of the theorem that uses max-steps or fewer steps, outputs True. Otherwise, outputs False.

Solution.

Exercise 12.5. [★] Is the Find-Proof Problem described below computable? Provide a convincing argument why it is or why it is not computable.

Find-Proof

Input: A specification of an axiomatic system, and a statement (the theorem).

Output: If there is a proof in the axiomatic system of the theorem, outputs True. Otherwise, outputs False.

Solution.

Exercise 12.6. Confirm that the machine showing in Figure 12.3 runs for 6 steps before halting.

Solution.

Exercise 12.7. Prove the Beaver Bound problem described below is also noncomputable:

Beaver-Bound

Input: A positive integer, n .

Output: A number that is greater than the maximum number of steps a Turing Machine with n states and a two-symbol tape alphabet can run starting on an empty tape before halting.

A valid solution to the Beaver-Bound problem can produce any result for n as long as it is greater than the Busy Beaver value for n .

Solution.

Exercise 12.8. [★★★] Find a 5-state Turing Machine that runs for more than 47,176,870 steps, or prove that no such machine exists.

Solution.