Assignment 7

Due:	Tuesday, November 17th at noon (Waterloo time)
Language level:	Intermediate Student with lambda
Files to submit:	`super.rkt`, `guess.rkt`
Practice exercises:	

- **Make sure you read the OFFICIAL A07 post on Piazza** for the answers to frequently asked questions.
- Unless stated otherwise, all policies from Assignment 06 carry forward.
- This assignment covers material up to the end of Module 13.
- The **only** built-in functions and special forms you may use are listed below. If a built-in function or special form is not in the following list, you may not use it:

```
* + - ... / < <= = > >= abs add1 and append boolean? ceiling char=? char?
  check-error check-expect check-within cond cons cons? define define-struct
  eighth else empty? equal? error even? exp expt fifth filter first floor
  fourth integer? lambda length list list->string list? local log max member?
  min negative? not number->string number? odd? or positive? quotient
  remainder rest second seventh sixth sqr sqrt string->list string-append
  string-length string-upcase string<=? string<? string=? string>=? string>?
  string? sub1 substring symbol=? symbol? third zero?
```

- Remember that basic tests are meant as sanity checks only; by design, passing them should not be taken as any indication that your code is correct, only that it has the right form.
- Marks specified for each question include marks for style and testing. Extra style marks may be deducted for the unnecessary use of global definitions.
- You not need purposes, contracts, tests, or examples for locally defined helper functions.
- Module 14 introduces `lambda` expressions. For this assignment you may use `lambda` expressions if you wish, but they are not needed. You do not need purposes, contracts, tests, or examples for any `lambda` expressions you use.
- You may find this assignment longer than previous assignments. Since there's no assignment due on November 10, you may want to start this assignment before then. Although the parts of Question 2 appear to build on each other, many of them can be completed independently of the others. If you get stuck on one part, try another.

1. Place your solutions to the following in `super.rkt`. You may not **`define`** anything globally other than the functions you are specifically asked to write.

   (a) **[10%]** In module 13 you were introduced to the function `(filter pred? lst)` that produces a list with the elements of `lst` for which the predicate `pred?` produces a true value. Write a function `super-filter` that generalizes `filter` to work on nested lists, applying the predicate to filter non-list elements in each nested list. Remember, you may use `list?` in this assignment.

   ```
   ;; A nested list of X (nested-listof X) is one of:
   ;; * empty
   ;; * (cons (nested-listof X) (nested-listof X))
   ;; * (cons X (nested-listof X))

   (check-expect
    (super-filter
     odd?
     (list 1 (list 2 (list 2 3 4) 5 6 (list 7 8 9)) 10 11 12))
    (list 1 (list (list 3) 5 (list 7 9)) 11))
   ```

   (b) **[6%]** Using a predicate with `super-filter`, define a function `(ruthless lst)` that removes the symbol `'ruth` from a nested list of symbols.

   ```
   (check-expect
    (ruthless
     (list 'rabbit
           (list 'apple 'pluto
                 (list 'ruth 'blue) 'ruth) 'hello))
    (list 'rabbit
          (list 'apple 'pluto
                (list 'blue)) 'hello))
   ```

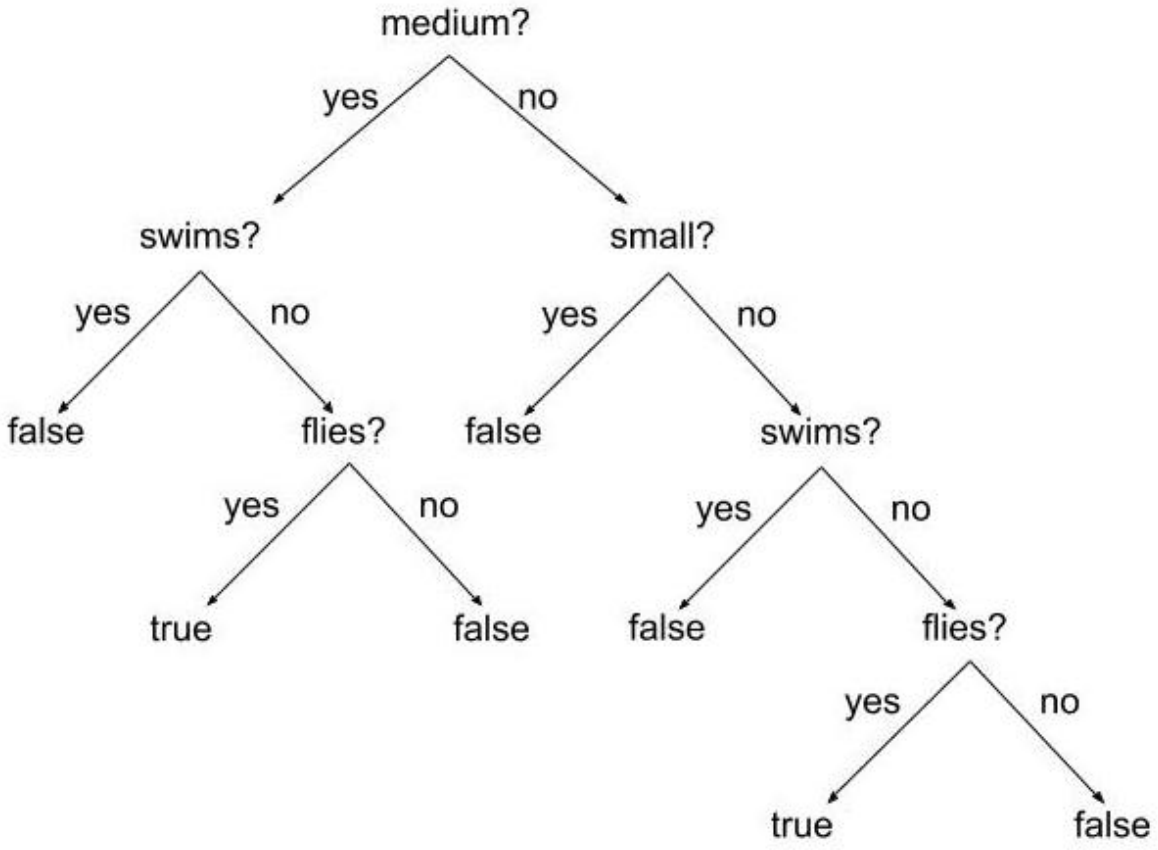
   (c) **[6%]** Using a predicate with `super-filter`, define a function `(supersize n lst)` that removes all numbers less than `n` from a nested list of natural numbers.

   ```
   (check-expect
    (supersize 4 (list 8 1 (list 2 6 3) 10 1))
    (list 8 (list 6) 10))
   ```

   (d) **[6%]** Using a predicate with `super-filter`, define `(super-keeper pred? lst)`, a function that produces a list with the elements of `lst` for which the predicate `pred?` produces a `false` value.

   ```
   (check-expect
    (super-keeper
     odd?
     (list 1 (list 2 (list 2 3 4) 5 6 (list 7 8 9)) 10 11 12))
    (list (list 2 (list 2 4) 6 (list 8)) 10 12))
   ```

2. A decision tree is a method for organizing a set of questions that lead to an answer. Each node in the tree asks a question, and the leaf nodes provide the answer. For example, here's a decision tree that might be used to decide if an animal is a `'crow`:



When we see an animal that might be a crow, we start at the root of the decision tree. If the animal is medium we follow the left subtree to make the decision, asking next if the animal can swim. If the animal isn't medium we follow the right subtree to make the decision, asking next if the animal is small. We continue down the tree in this way until we reach a leaf, which will be either be `true` or `false`, indicating if the animal is a crow. Once we have a decision tree, we can use it to create a predicate `crow?` which will tell you if a certain animal is a 'crow based on a given list of attributes

```
;; crow?: (listof Sym) -> Bool
(check-expect (crow? (list 'medium 'flies)) true)
```

Now, imagine you are walking around Waterloo Park writing down the attributes of the animals you see. You might come up with a list of your observations like this:

```
(define seen
  (list
    (list 'squirrel 'small 'angry)
    (list 'goose 'large 'swims 'flies 'angry)
```

```
    (list 'goose 'large 'swims 'flies 'angry)
    (list 'crow 'medium 'flies 'angry)))
```

Each element in the list starts with a label indicating the animal seen, followed by any number of attributes describing the animal.

Since you are a student in CS135 in Fall 2020, you already know how to recognize animals in Waterloo Park, but perhaps a newcomer to Waterloo won't know which animal is which. You decide to take your observations and create predicates that will help newcomers recognize animals in the park. To do this you will need to make many observations, perhaps thousands. To help you make these observations, the file `animals.rkt` contains a function `random-animals` that consumes a natural number and produces a random list of that many observations. To use `random-animals`, download `"animals.rkt"` to your working folder and put `(require "animals.rkt")` at the start of your solution file.

```
> (random-animals 1000)
(list
 (list 'crow 'large 'flies 'angry)
 (list 'sparrow 'small 'flies)
 ...
 (list 'goose 'large 'swims 'angry)
 (list 'squirrel 'small 'flies 'angry))
```

If you look at a list of random animals, you will see that there's at least a small chance that any animal will have any given attribute. For example, the list above contains a flying `'squirrel`. Nonetheless, a `'goose` is much more likely to fly than a `'squirrel`, and you should probably guess `'goose` rather than `'squirrel` if the animal flies. While `random-animals` won't produce an animal that has more than one size (e.g., a `'goose` that's both `'large` and `'small`) that restriction won't actually matter for this question.

Since you might make thousands of observations, you decide to write a function that creates predicates for you, based on a list of example observations:

```
; train-classifier: (listof Example) Sym -> ((listof Sym) -> Bool)
```

This function consumes a list of examples and a label, and produces a predicate to recognize items matching the label based on their attributes. In machine learning, this type of predicate is called a *classifier*. A label is just a symbol. An example has the following data definition:

```
;; An Example is a (cons Sym (listof Sym))
;; Requires: each attribute in the rest is unique
```

To implement `train-classifier` we will first build a decision tree from the list of examples and then create a classifier from the decision tree.

Suppose our list of examples comprises the observations we made in Waterloo Park, then we can create a classifier to recognize a `'goose` as follows:

```
> (define goose? (train-classifier (random-animals 1000) 'goose))
> (goose? (list 'angry 'swims 'flies 'large))
true
```

You realize that `train-classifier` need not be limited to animals. For example, imagine we have a function `random-pirates`:

```
> (random-pirates 3)
(list
  (list 'captain 'hook 'angry)
  (list 'sailor 'pegleg 'eyepatch)
  (list 'captain 'hat 'hook 'parrot))
```

We could build a decision tree classifier for pirate captains as follows:

```
> (define captain? (train-classifier (random-pirates 1000) 'captain))
> (captain? (list 'angry 'parrot 'hook))
true
```

We won't give you `random-pirates`. Instead, we are keeping it to ourselves to test the performance of your implementation of `train-classifier`.

The remainder of this question will help you through the process of implementing classifiers step by step. You will create decision trees using a machine learning algorithm called ID3, which you can read about in Wikipedia if you are interested. However, we will provide all the information you need to implement this algorithm as part of this assignment.

Although we recommend that you follow these steps, and use ID3, it is not required. You can use any machine learning method you wish, including the method mentioned in the enhancements or methods you might learn about through your own reading. If you choose to use other methods, your grade will be based entirely on the performance of your `train-classifier` function using the `performance` function described later. If you follow the step-by-step procedure, you will be able to receive marks for each step, as indicated.

Place your solution in `guess.rkt`. Apart from the functions you are specifically asked to write, if you **`define`** anything that is only used in one part of a question it **must** be defined locally. Helper functions and constants that are used in multiple parts of a question may be defined globally. If you choose to use a different machine learning method, **all** helper functions and constants must be defined locally.

(a) **[8%]** Given a list of examples we will want to collect all attributes an example might have. Write a function `(collect-attributes examples)` that consumes a list of examples and produces a list of attributes contained in the examples with no duplicates. Remember that the label at the start of each example is not an attribute.

```
> (collect-attributes seen)
(list 'medium 'flies 'swims 'large 'angry 'small)
```

The order of the attributes in the result doesn't matter, as long as there are no duplicates.

(b) **[8%]** Given a list of examples we will want to split the list into two lists, one list containing all examples with a particular attribute or label, and one list not containing that attribute or label. Note that we can split on either an attribute or a label, and for the purposes of this function we can view an example as just a list of symbols. Write a function `(split-examples examples symbol)` that splits the list of examples on the given symbol. The function should produce a list of two lists of examples, with the first containing the examples containing the symbol and the second containing the examples not containing the symbol.

```
> (split-examples seen 'goose) ; splitting on a label
(list
  (list
    (list 'goose 'large 'swims 'flies 'angry)
    (list 'goose 'large 'swims 'flies 'angry))
  (list
    (list 'crow 'medium 'flies 'angry)
    (list 'squirrel 'small 'angry)))
> (split-examples seen 'small) ; splitting on an attribute
(list
  (list
    (list 'squirrel 'small 'angry))
  (list
    (list 'crow 'medium 'flies 'angry)
    (list 'goose 'large 'swims 'flies 'angry)
    (list 'goose 'large 'swims 'flies 'angry)))
```

(c) **[8%]** Given a list of examples, we will want to compute some statistics about the examples. More specifically, we will need to compute a *histogram* indicating how many times each attribute appears in the examples. Write a function `(histogram examples)` that consumes a list of examples and produces a list of attribute/count pairs, with each pair indicating how many times that attribute appears in the examples. The pairs in a histogram can be in any order.

```
;; A Histogram is a (listof (list Sym Nat))
;; Requires: A symbol can appear in only one pair.

> (histogram seen)
(list
  (list 'small 1) (list 'angry 4) (list 'large 2)
  (list 'swims 2) (list 'flies 3) (list 'medium 1))
> (histogram (random-animals 1000))
(list
  (list 'small 517) (list 'swims 504) (list 'flies 762)
  (list 'large 260) (list 'angry 632) (list 'medium 223))
```

(d) **[8%]** This next step follows up from the previous step, augmenting a histogram with some additional information that will simplify the creation of a decision tree. First, since we will be splitting the examples on a label to build the tree, and both splits may not contain all attributes, we want to add any missing attributes to the histogram with

counts of zero. Second, we want to add to each element of the histogram the number of examples *not* containing the attribute. To do this, we simply take the difference between the value in the histogram and the total number of examples. Write a function `(augment-histogram histogram attributes total)` that augments a histogram as described above. Along with the histogram, the function consumes a list of all attributes and a total for the number of examples. This total will be greater than or equal to any value in the histogram. The function produces an augmented histogram (AH), which is a list of `(list Sym Nat Nat)` triples, with the following data definition:

```
;; An Augmented Histogram (AH) is a (listof (list Sym Nat Nat))
;; Requires: A symbol can appear in only one triple.

(check-expect
 (augment-histogram
  (list (list 'a 100) (list 'c 50))
  (list 'a 'b 'c)
  200)
 (list (list 'a 100 100) (list 'b 0 200) (list 'c 50 150)))
(check-expect
  (augment-histogram empty (list 'x 'y) 10)
  (list (list 'x 0 10) (list 'y 0 10)))
```

Notice that examples and test cases can be written independently of the previous steps.

(e) **[8%]** This next step computes a measure called *entropy*, which we will use to select the attribute to place at the root of the decision tree. Entropy measures the amount of uncertainty associated with an attribute. For the root of the decision tree we want to pick the attribute with the minimum entropy, which provides the most certainty in distinguishing one thing from another. To compute entropy, we start with a *contingency table*, which compares the counts in histograms for examples with a target label vs. those examples without the target label.

	label	$\overline{\text{label}}$	total
attribute	a	b	$a+b$
$\overline{\text{attribute}}$	c	d	$c+d$
total	$a+c$	$b+d$	$a+b+c+d$

Where the notation $\overline{\text{label}}$ indicates examples that do not have the label and $\overline{\text{attribute}}$ indicates examples that do not have the attribute. Consider the following random animals:

```
> (define examples (random-animals 1000))
> (define attributes (collect-attributes examples))
> (define split (split-examples examples 'goose))
> (define goose (histogram (first split)))
> (augment-histogram goose attributes (length (first split)))
(list
  (list 'large 126 59) (list 'angry 161 24)
  (list 'small 17 168) (list 'flies 170 15)
```

```
  (list 'swims 162 23) (list 'medium 42 143))
> (define not-goose (histogram (second split)))
> (augment-histogram not-goose attributes (length (second split)))
(list
  (list 'large 146 669) (list 'angry 469 346)
  (list 'small 454 361) (list 'flies 615 200)
  (list 'swims 365 450) (list 'medium 215 600))
```

This example would give us the following contingency table for large geese.

	goose	$\overline{\text{goose}}$	total
large	126	146	272
$\overline{\text{large}}$	59	669	728
total	185	815.	1000

Notice that unless an animal is `'large`, it's very unlikely to be a `'goose`. Knowing if an animal is `'large` or not, reduces our uncertainty about whether or not an animal is a `''goose`. Entropy quantifies this uncertainty, so that we pick the attribute with the minimum uncertainty for the root of our decision tree. To compute the entropy for a 2×2 contingency table, we first define $P(n,m)$, which estimates a probability from a pair of counts, $n \geq 0$ and $m \geq 0$:

$$P(n,m) = \begin{cases} \frac{n}{n+m}, & \text{if } n+m > 0, \textit{and} \\ 0.5, & \text{if } n+m = 0. \end{cases}$$

Notice that $P(x,y) = 1 - P(y,x)$. We then define for a probability, $0 \leq p \leq 1$:

$$e(p) = \begin{cases} -p\log_2(p), & \text{if } 0 < p \leq 1, \textit{and} \\ 0, & \text{if } p = 0. \end{cases}$$

Finally, the entropy of a 2×2 contingency table is computed as:

$$P(a+b,c+d)(e(P(a,b)) + e(P(b,a))) + P(c+d,a+b)(e(P(c,d)) + e(P(d,c)))$$

Write a function `(entropy positive-counts negative-counts)` that consumes two elements from augmented histograms and produces their entropy. You can assume that the elements are for the same attribute and are taken from the augmented histogram for each split.

```
> (entropy (list 'large 126 59) (list 'large 146 669))
#i0.5663948489858
> (entropy (list 'small 17 168) (list 'small 454 361))
#i0.5825593868115
> (entropy (list 'a 0 100) (list 'b 100 0))
0.0
```

Since entropy can be difficult to compute by hand, you can base your testing on these three examples. You do not need to include any other examples or tests. Use `check-within` to three decimal places (i.e., within 0.001).

(f) **[8%]** Next, we determine the entropy for each attribute across pairs of augmented histograms. Write a function `(entropy-attributes positive negative)` that consumes two augmented histograms and computes the entropy of each attribute, producing a list of attribute/entropy pairs. You can assume that exactly the same attributes will be included in both augmented histograms and that the attributes will appear in the same order. The function produces an entropy association list (EAL) with the following data definition:

```
;; An Entropy Association List (EAL) is a (listof (list Sym Num))
;; Requires: A symbol can appear in only one pair.
```

Since entropy can be difficult to difficult to compute by hand, you can base your testing on the following example:

```
> (entropy-attributes
    (list
     (list 'large 126 59) (list 'angry 161 24)
     (list 'small 17 168) (list 'flies 170 15)
     (list 'swims 162 23) (list 'medium 42 143))
    (list
     (list 'large 146 669) (list 'angry 469 346)
     (list 'small 454 361) (list 'flies 615 200)
     (list 'swims 365 450) (list 'medium 215 600)))
(list
 (list 'large #i0.5663948489858) (list 'angry #i0.6447688190492)
 (list 'small #i0.5825593868115) (list 'flies #i0.6702490498564)
 (list 'swims #i0.6017998773730) (list 'medium #i0.6901071708677))
```

You do not need to include any other examples or tests. Use `check-within` to three decimal places.

(g) **[8%]** Now we determine the attribute with the minimum entropy, which will form the basis for the question at the root of the decision tree. Write a function `(best-attribute entropies)` that consumes a non-empty list of attribute/entropy pairs, such as those generated by `entropy-attributes`. The function should produce the attribute with the minimum entropy, or any such attribute if somehow there's a tie.

```
> (best-attribute
   (list
    (list 'large #i0.5663948489858) (list 'angry #i0.6447688190492)
    (list 'small #i0.5825593868115) (list 'flies #i0.6702490498564)
    (list 'swims #i0.6017998773730) (list 'medium #i0.6901071708677)))
'large
```

Notice that this step is really just finding the key with the minimum value in an association list. There's nothing about the function that needs to be specific to attribute/entropy pairs, and you can write test cases that show it works on any association list from symbols to numbers.

(h) **[8%]** Now we are ready to build decision trees. We use a list representation for the trees, with a data definition as follows:

```
;; A Decision Tree (DT) is one of:
;; * Bool
;; * (list Sym DT DT)
```

The first element of the list represents an attribute. If the attribute appears in the example, follow the first DT to decide. If the attribute does not appear in the example, follow the second DT. If the DT is a Bool, that's the decision.

Using this data definition, write a function `(build-dt examples label)`. Here's what it should do to implement the ID3 algorithm mentioned in the introduction to this question. In implementing these steps, **`local`** is your friend. Don't be afraid to nest **`local`** definitions.

- Collect the attributes from the examples using `collect-attributes`.
- Split the examples into two lists, one with the label (called the *positive examples*) and one without the label (called the *negative examples*). Use `split-examples`.
- If the list of positive examples is empty, the decision is `false`.
- If the list of negative examples is empty, the decision is `true`.
- If the list of attributes is empty, then there are two possibilities: If the list of positive examples is longer than the list of negative examples the decision is `true`. Otherwise, the decision is `false`.
- Use `histogram`, `augment-histogram`, and `best-attribute` to select the attribute with minimum entropy. We will call this attribute the *root attribute*.
- Split the examples into two lists, one with the root attribute and one without the root attribute. Use `split-examples`. Remove the root attribute from the first list of examples.
- Call `build-dt` recursively on each list to create two subtrees.
- If the subtrees are the same (compare them using `equal?`) just produce it. Otherwise, produce a list containing the root attribute and the two subtrees.

```
> (build-dt (random-animals 1000) 'goose)
(list 'large (list 'swims (list 'angry true false) false) false)
> (build-dt (random-animals 1000) 'crow)
(list 'swims
      false
      (list 'flies
            (list 'angry
                  (list 'medium
                        true
                        (list 'large
                              true
                              (list 'small true false)))
                  false)
            false))
```

```
> (build-dt (random-animals 1000) 'emu)
false
```

Like any machine learning algorithm, it's hard to write test cases for `build-dt`. For this part only, you need not include test cases, although you should check the trees for various animals to see that they reach reasonable looking decisions. You should typically test with 1000 or more random examples. For symbols that do not appear in the examples, such as `'emu` your DT should always be `false`. To test the performance of your decision tree classifier over many examples (including `random-pirates`) we will use *sensitivity* and *specificity* as described in the bonus question.

(i) **[8%]** Finally, we create a classifier using our decision tree algorithm. Write a function `(train-classifier examples label)` as described in the introduction. It should generate a decision tree from the examples, and then produce a predicate that consumes a list of attributes and produces a decision.

```
(define goose? (train-classifier (random-animals 1000) 'goose))
(check-expect (goose? (list 'large 'angry 'flies 'swims)) true)
(check-expect (goose? (list 'small 'angry)) false)
(define squirrel? (train-classifier (random-animals 1000) 'squirrel))
(check-expect (squirrel? (list 'large 'angry 'flies 'swims)) false)
(check-expect (squirrel? (list 'small  'angry)) true)
(define crow? (train-classifier (random-animals 1000) 'crow))
(check-expect (crow? (list 'angry 'flies 'medium)) true)
```

Using at least 1000 examples generated by `random-animals` we expect that these test cases will always pass. We will test your implementation using the procedure outlined in the bonus question. You don't need to implement the bonus. We will provide our own version.

Bonus Question [10%]

Classifiers like the ones created by `train-classifier` can be measured in terms of their *sensitivity* and *specificity*, where:

- sensitivity is the percentage of positive examples that are correctly classified, and
- specificity is the percentage of negative examples that are correctly classified.

Medical tests are often characterized in terms of their sensitivity and specificity. For example, according to Public Health Ontario tests for detecting SARS-CoV-2 can have "a sensitivity of "70% to 90%" and "a specificity of over >99%".

Write a function `(performance classifier? examples label)` that takes a classifier, examples, and a label and computes the sensitivity and specificity for the classifier on the examples. You

can assume that the label is the same as the label used to train the classifier. To keep the classifier from "cheating" by looking at the labels, you should remove the labels from each example before you apply the classifier to it. The function should produce a list with three elements: the label, the sensitivity as a percentage rounded to 1% and the specificity rounded to 1%. You can use Racket's `round` function for this purpose.

```
> (performance goose? (random-animals 1000) 'goose)
(list 'goose 59 94)
> (performance squirrel? (random-animals 1000) 'squirrel)
(list 'squirrel 79 97)
> (performance captain? (random-pirate 1000) 'captain)
(list 'captain 93 98)
```

To test your implementation of `train-classifier` we will use our own implementation of `performance`. A sensitivity of 50% or more and a specificity of 90% or more on our test cases will receive full correctness marks, which ID3 should be able to achieve on our test cases. We will always use at least 1000 examples for both `train-classifier` and `performance`.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

Decision trees were a key machine learning technique until the last five years or so, when neural networks became dominant. The C4.5 algorithm was the immediate successor to the ID3 algorithm and was widely used in practice for simple classification tasks. C4.5 improved on ID3 in a number of ways, including the ability to handle attributes that have numeric values, such as the weight of an animal. C4.5 can also handle missing attributes. For example, we may not know if an animal flies or not. If you are interested, you can try to extend your classifier to handle numeric attributes and missing attributes using the C4.5 algorithm for inspiration.