MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

Problem Set 8

- Issued: Tuesday, March 31, 1998
- Tutorial preparation for: Week of April 6
- Written solutions due: Friday, April 10 in recitation
- Reading: Read section 3.4 before lecture on April 2. Read section 4.1 before lecture on April 9.
- Reminder: Quiz 2 will be held on Tuesday evening, April 14. More information will be distributed in lecture on Tuesday, April 7.

Streams and Speech Recognition

In lecture on March 31, we looked at streams and at some of the computational approaches that are associated with these data structures. We saw that the key concept behind streams was to decouple the apparent order of evaluation of expressions from the actual order evaluation in the computer. By doing so, we can execute processes without having to first generate all the stream elements. This is a useful variation on our earlier approaches to computation. There are many applications that are naturally viewed in terms of processing continuous sequences (or streams) of values (e.g. electrical signals, sonar, radar, sound, etc.).

Consider speech recognition. The input to a speech recognizer might be a stream of frequencies based on the interaction of sound waves generated by the speaker with some receiver (e.g. a taut drum that converts wave fronts to vibrations that can be converted to an electrical signal). The goal is to convert the electrical signal that represents the sound waves into words. In traditional approaches to speech recognition, there is an intermediate step in which the stream of vibrations is first converted into subunits of actual spoken words, called *phonemes*. These represent the units of sound that comprise spoken words.

For example, figure 1 shows a sample speech input.

Speech recognition would be simple if there was a clear transformation from vibrations to phonemes, and then to words. Unfortunately, in most normal situations, the speech signal is noisyand ambiguous. This means first that the interpretation of the vibration stream into phonemes is not clearcut—there may be several phonemes that could correspond to the detected vibrations. Second, the breaks between words may not be clear, so that one has to worry about where to separate the boundaries of the phonemes. Indeed, in the example shown in the figure, there is no obvious silence or gap between "he" and "ate". Because of these phenomena, building good speech-recognition systems is very difficult.

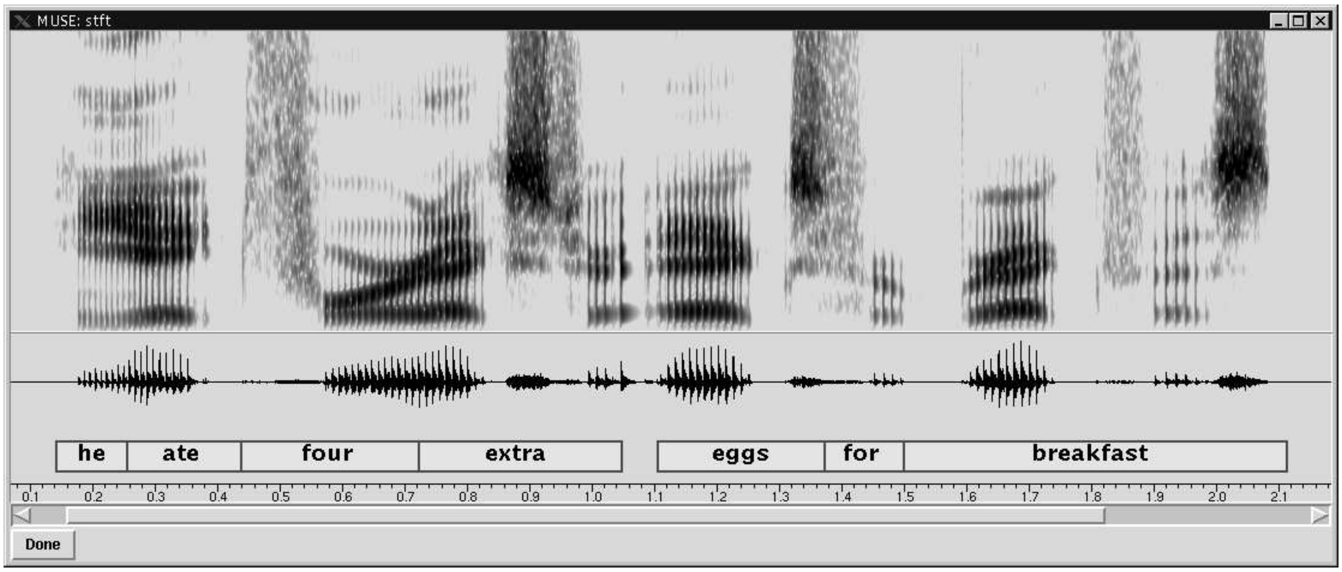

Figure 1: The picture illustrates three useful representations for human speech during computer speech recognition. The top display results from the short-time Fourier analysis of the speech signal and depicts the varying distribution of spectral energy across time and frequency (darker means more energy). The next display is the amplitude of the waveform, recorded and digitized at 16,000 samples per second. Finally, the transcription shows which words were actually spoken, and their corresponding alignments in time (the time axis is measured in seconds).

1. Background

In this problem set, you will experiment with an idealized system for speech-recognition, with the primary aim of seeing how stream processing can be a powerful tool for manipulating certain kinds of data. We are going to make several simplifying assumptions (after all, we don't really expect you to build a state-of-the-art speech recognition system in one week).

First, we'll not worry about converting actual sounds to phonemes. Instead, we'll use a program that simulates the operation of a "front end" to the speech processor by taking a stream of symbols that represent phonemes and putting out a stream of results. Since speech signals are noisy, our simulation will produce, for each input "phoneme" a list of possible phonemes this "sounds like," together with a probability that this is the actual phoneme that was "heard." For example, the front end might determine that a phoneme is either an "s" with probability .8, or an "sh" with probability .15, or a "c" with probability .05. Our simulator will include a parameter that reflects the amount of noise—phoneme identification becomes less reliable as the noise increases.

Our second simplification is that we'll assume that the ordinary English spelling of words can be used as their phonetic spelling (which is false). In the speech-recognition community, phonemes correspond to specific speech sounds, and they are notated in a system like the International Phonetic alphabet. To keep things simple, we will approximate this by using substrings of words as phonemes. Thus, the word "finish" might be represented by the sequence "f" "i" "n" "i" "sh". Overall, we'll use the following set of "phonemes":

`a b c ch d e ee f h i j k l ll m n o p qu r s sh ss t th u v w wh x y z`

The following table shows, for our simple model, which phonemes can get mistaken for other phonemes by our (simulated) front-end processor:

```
(define *confusion-list*
  '((a o u) (b p d t g) (c k ch t) (ch s sh) (d th) (e i o a) (ee i o a) (f v)
    (g j ch) (h wh) (i e a o) (j g) (k c) (l r) (ll r) (m n) (n m) (o a)
    (p b t d) (qu k) (r l) (s sh) (sh s ch) (ss sh ch) (t d b p) (th d b p)
    (u o) (v f) (w u) (wh h) (x z s) (y i) (z s x)))
```

The interpretation of this table is that an input "a" could be heard as an "a" or an "o" or a "u"; an input "b" as a "b", "p", "d", "t", or "g", and so on.

Our third and most important simplifying assumption is to assume that individual words in the input stream are clearly separated by silences, which we'll notate in the input stream by stars. Thus, the phrase "the cat looked at the fish" would appear as input to the font-end processor as

```
th e * c a t * l oo k e d * a t * th e * f i sh
```

In accordance with `*confusion-list*`, the front-end processor would report that the first phoneme is "th" or "d" or "b" or "p", the second phoneme is "e" or "i" or "o" or "a", and so on. Observe that these confusions are unrealistic: we'd do much better if we used real phonetic spellings.

In order to disambiguate the possible phoneme choices produced by the front end, our system will need to know which sequences of phonemes correspond to sounds. To do this, we'll use a dictionary that gives the "phonetic spellings" of common words. In our simplified approach, the phonetic spelling is the same as the real spelling, so to look up a sequence of phonemes, we'll concatenate the phonemes and see if this produces a word in the dictionary, which is a list of about a thousand common English words.

Our goal in this problem is to use these components to produce a system that can decode a noisy stream of phonemes into words and thus recognize what is being "said."

2. Tutorial Exercises

You should prepare these exercises for oral presentation in tutorial.

Tutorial exercise 1: The following procedure `all-choices` (which you will use in doing this problem set) takes as argument a list of n lists $L_1, L_2, \ldots, L_n$ and returns a list whose elements are all possible lists of the form $(x_1, x_2, \ldots x_n)$, where x_1 is chosen from L_1, x_2 is chosen from L_2, $\ldots$, x_n is chosen from L_n. (Mathematicians call this operation the *Cartesian product* of the input lists.) For example, `(all-choices '((1 2) (a b c)))` should return

```
((1 a) (1 b) (1 c) (2 a) (2 b) (2 c))
```

```
(define (all-choices possibilities-list)
  (if (null? possibilities-list)
      '(())
      (let ((rest (all-choices (cdr possibilities-list))))
        (flatten
         (map
          (lambda (choice-for-first)
            (map (lambda (follow-on)
                   (cons choice-for-first follow-on))
                 rest))
          (car possibilities-list))))))
```

Explain how this procedure works. In general, how many possible choices will there be as a function of the input list? How could you modify the procedure to produce a stream of choices rather than a list of choices. (Assume that the argument is still a list.)

Tutorial exercise 2: Using `cons-stream, stream-map, stream-filter`, create expressions that define an infinitely long stream of 1's and an infinitely long stream of the positive integers. Be prepared to explain why your expressions work.

Tutorial exercise 3: Using the same tools, create an expression that defines an infinitely long stream of the powers of 2 (i.e. 1, 2, 4, 8, 16, ...).

Tutorial exercise 4: Do exercise 3.54 from the text.

Tutorial exercise 5: Do exercise 3.55 from the text.

3. Programming Assignment

Begin by loading the code for problem set 8. The first thing to try is to see how Scheme deals with streams of inputs you type from the keyboard. The code for the problem set includes a procedure `input-stream` that generates a stream of the expressions you type, and also a procedure `stream-pp` that pretty-prints a stream one element at a time.

Try evaluating `(stream-pp (input-stream))`. If you are in the `*scheme*` buffer, you should see a prompt at the bottom of the screen requesting input. Type an expression followed by ENTER. Scheme will pretty-print the expression and request more input. This input stream is a potentially infinite stream: you can keep entering expressions for as long as you like. To terminate the stream type a period. Alternatively, you can break out the input by typing `C-g` (i.e., CTRL-`g`).

You'll be using `input-stream` to enter streams of phonemes. As an alternative to always entering items one at a time at the keyboard, you can also use the procedure `list->stream`, which converts a list of elements into a stream of elements. Try evaluating

```
(stream-pp (list->stream '(th e * c a t * l oo k e d * a t * th e * f i sh)))
```

and then produce the same printed response using `input-stream`.

Simulating the front-end processor

The problem set code includes a procedure `make-random-confusion` that takes as arguments a (supposedly spoken) phoneme and a noise level and returns a list of possible phonemes this might be, together with a weight (probability) to assign to that phoneme choice. For example

```
(make-random-confusion 'a .2)
;Value: ((u .54) (a .32) (o .14))
```

produces a list that says that the result is "u" with likelihood .54, "a" with likelihood .32, and "o" with likelihood .14. (Note that the actual input was "a".)

We'll call such a result a list of *weighted elements*. A weighted element consists of some data and an associated weight. The problem set code provides a constructor `make-weighted` and selectors `weighted-data` and `weighted-weight` for handling weighted elements.

`Make-random-confusion` is designed so that the weights of the elements it returns sum to 1 and the elements are listed in order from largest to smallest weight. The procedure uses randomness to simulate the noise, so calling it repeatedly will give different results.

The second argument to `make-random-confusion` represents a noise level, which should be between 0 and 1. With noise level 0, the "real" phoneme will (over many evaluations) tend to have a weight five times as large as that of the alternatives. With noise level 1 (very noisy) all the choices will tend to have the same weight—the phoneme that was actually input will not be preferred to any of the alternatives.

Computer exercise 1: Write a procedure `simulate-front-end` that takes a stream of phonemes and a noise level as arguments and maps `make-random-confusion` along the stream to produce the stream of weighted alternatives. Using this, implement procedures `phoneme-input-stream` and `noisy-phoneme-input-stream` that simulate the operation of the front-end processor on a stream of phonemes you type at the keyboard. (Use noise levels of 0 and 1 for the two streams.) Try these out and observe the results. Also implement `phoneme-stream-from-list` and `noisy-phoneme-stream-from-list` that simulate the front-end processor operating on a given list of phonemes. Turn in listings of your definitions, and also the result of evaluating

```
(stream-pp (noisy-phoneme-stream-from-list '(p i g)))
```

Splitting the stream into "words"

Now that we can simulate streams of phoneme alternatives coming from the front-end processor, let's split these into words so we can try to identify them. The procedure `split-stream`, included in the problem set code, takes as arguments a stream and a predicate. It divides the stream into chunks, producing a stream of lists, where each list is a chunk of consecutive elements of the stream. Stream elements that satisfy the predicate are assumed to be markers that signal the beginning of a new chunk. These markers are not included in any of the chunks.

Computer exercise 2: Using `split-stream`, implement a procedure `split-stream-at-silences` that chunks the stream into "words" by splitting the stream at the stars. Demonstrate your procedure by using `stream-pp` to look at the result of splitting a `phoneme-input-stream` where you input a short phrase such as

```
a * p i g * l oo k e d * a t * th e * b i g * d o g
```

You should see your output printed a word at a time, so you won't see anything printed until you type a star. (Note: When `make-random-confusion` is given a "phoneme" symbol that is not in its table (e.g., a star) it returns a weighted object with that symbol as data and a weight of 1. So to test for a star, you should test whether the data part of the object is the symbol `*`.)

Finding real words

Now that we have the stream divided into chunks, we need to decipher each chunk into possible words. We'll start by generating all possible choices of phoneme sequences for each chunk, using the procedure `all-choices` which you reviewed for tutorial. We'll then filter these choices to find the ones that correspond to words in the dictionary.

To permit you to do the filtering, the problem set code includes a procedure called `word-in-dictionary?` that takes a list of phonemes and checks whether this is a word in the dictionary. (In our idealized implementation, this amounts to simply concatenating the phonemes and checking whether this produces a word in the dictionary.)

Computer exercise 3: Implement this plan. First, to provide some test data, create a "chunk" of phonemes corresponding to a single word:

```
(define one-chunk
     (stream-car
      (split-stream-at-silences
       (phoneme-stream-from-list '(c a t)))))
```

Note that the chunk here is a list—`split-stream-at-silences` produces a stream of lists, and you are taking the first element in that stream. Since `one-chunk` is a list you can look at it by pretty printing it with `pp` (not `stream-pp`). You should find that there are 4 choices for the first phoneme ("c" and three others), 3 choices for the second, and 4 choices for the third. Now generate the list of all choices. How many choices are there?

Next, implement a procedure called `sensible?` that takes a list of weighted phonemes and uses `word-in-dictionary?` to see if this is in the dictionary. (`Word-in-dictionary?` expects just the phoneme data, not the weights, so you'll have to extract these from the weighted phonemes.)

Having done this, you should be able to filter the list of choices to find the sensible ones, e.g.,

```
(pp (filter sensible? (all-choices one-chunk)))
```

You should find four possibilities that match words in our dictionary. What are these?

Computer exercise 4: Your result in exercise 3 should produce "words" that are lists of weighted phonemes. For further processing, it will be convenient to combine these weights to get an overall weight for the word rather than preserving the weights of the individual phonemes. Write a procedure `combine-weighted` that accomplishes this by taking a list of weighted elements and producing a single weighted element. The data part of the result should be a list of the data items of the individual weighted elements in the argument list, and the weight part of the result should be the product of the individual weights. Here's an example:

```
(combine-weighted '((c .6) (a .74) (t .5)))
;Value: ((c a t) .222)
```

By mapping `combine-weighted` down the list of results in exercise 3, you should get a list of possible words, each with a weight. You can now pass this result to a procedure called `normalize-weights`, which is supplied with the problem set code. `Normalize-weights` takes a list of weighted elements and normalizes the weights so that the sum of the weights is 1, and hence weights can be interpreted as probabilities. It also orders the list from largest to smallest weight.

Finally, define a procedure called `possible-words`, which performs all the operations that you did here and in exercise 3. Namely, `possible-words` should take as argument a chunk of output from the front-end processor, i.e., a list each of whose elements is a list of weighted phonemes. The procedure should generate all choices of phoneme sequences, filter these to find sequences in the dictionary, map `combine-weighted` down the resulting list, and finally normalize the weights.

Test your implementation of `possible-words` by applying it to `one-chunk` from exercise 3. Also try evaluating

```
(stream-pp
 (stream-map possible-words
             (split-stream-at-silences
              (phoneme-input-stream))))
```

and typing a stream of phonemes (with words separated by stars) to see the stream of possible words for each phoneme sequence you enter.

Computer exercise 5: Suppose you enter a phoneme sequence for which there are no possible words in the dictionary. This should appear in your output from exercise 4 as an empty list. To make this more uniform with the other lists of possible words, it would be good to replace this with a list containing an actual weighted element. Write such a procedure, `flag-unrecognized`, which simply returns its argument, unless its argument is the empty list, in which case it returns a list of one weighted word whose data is the symbol `*unrecognized*` and whose weight is 1. Try your test from the end of exercise 4 again, only this time also stream-mapping the result through `flag-unrecognized`, and see what happens if you type a mixture of words and gibberish.

Computer exercise 6: Now put all this processing together by defining a procedure called `recognize`, which takes a phoneme stream (with words delineated by stars) and returns the stream of weighted choices for words. Namely, `recognize` should split the stream at silences, stream map `possible-words` along the result, and stream map this through `flag-unrecognized`. Test your procedure by trying it on `phoneme-input-stream` and `noisy-phoneme-input-stream`. For experimentation, you can find the definition of the dictionary in the file `dict.scm`. Feel free to add more words. Turn in some sample results.

From words to phrases

With `recognize` from exercise 6 in hand, we're now in an analogous position to where we started with the results of the front-end processor. Rather than a stream of lists of weighted phoneme choices to assemble into words, we now have a stream of lists of weighted word choices. Can we perform the analogous operation—comb through the possible choices of words to produce phrases and sentences?

Let's start by simply generating all possible choices: Take a stream of word possibilities generated by `recognize`, transform this to a list (assume this is a finite stream), pass the result to `all-choices` and combine the weights and normalize them. As an example, try the following

```
(define (possible-phrases phoneme-stream)
  (normalize-weights
   (map
    combine-weighted
    (all-choices (stream->list (recognize phoneme-stream))))))
```

Computer exercise 7: Try the above approach. As test cases, use

```
(phoneme-stream-from-list '(a * b i g * p i g * r a n * b y * th e * d o g))
```

and also `phoneme-input-stream`. (Notice that you won't get any result printed here until you type a period to terminate the input stream.) Also try using `noisy-phoneme-stream-from-list` and `noisy-phoneme-input-stream`. Just as with the original problem of assembling phonemes into words, you should find that there are many potential sequences of words, only a few of which are sensible phrases.

Making the system smarter

We're not going to ask you to take the next step and produce meaningful phrases from the combinations of words. That would take us into the realm of ongoing research in speech recognition, which would be a little intense for a one-week problem set (even in 6.001!).

Computer exercise 8: Write a brief paragraph, describing how you think you might approach the problem of reducing the word choices to extract "meaning". Think back to the demonstration in lecture on March 31 of the work of the Lab for Computer Science's Spoken Language Systems Group and say how one might begin building a system of that type.

Turn in answers to the following questions along with your answers to the questions in the problem set:

1. About how much time did you spend on this homework assignment? (Reading and preparing the assignment plus computer work.)
2. Which scheme system(s) did you use to do this assignment (for example: 6.001 lab, your own NT machine, your own Win95 machine, your own Linux machine)?
3. We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).
   - If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the text (such as some of the archives), please indicate your consultants' names and your references. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.
   - Otherwise, write "I worked alone using only the reference materials," and sign your statement.