Lecture 14: Concise Code, Big-O, and Internal Representation of Lists
10:00 AM, Oct 5, 2018

Contents

1 Good Coding Practices
   1.1 More Concise Code ................................................. 1
   1.2 Quote .............................................................. 2

2 Pattern Matching .................................................. 2
   2.1 Example Problem: prefix? ........................................... 3
      2.1.1 Code ........................................................ 3
   2.2 Example Problem: sublist? ........................................... 5
      2.2.1 Code ........................................................ 5

3 Internal Representation of Lists ................................ 5
   3.1 Interpreting the contents of the first memory location in a cons cell . . . . . . . . . . . . 7
   3.2 car, cdr, and cons ................................................. 7

4 More Order of Growth ............................................. 8

1 Good Coding Practices

1.1 More Concise Code

We talked about the alternating harmonic series, represented as follows:

\[ 1 - \frac{1}{2} + \frac{1}{3} - \ldots \pm \frac{1}{n} \]  \hspace{1cm} (1)

Keeping in mind that the sign of the last term depends on whether \( n \) is even or odd, we defined the code which calculates the sum up to the \( n \)th term as:
(define a-harm
  (lambda (n)
    (cond
      ((zero? n) 0)
      ((even? n) (- (a-harm (- n 1)) (/ 1 n)))
      ((odd? n) (+ (a-harm (- n 1)) (/ 1 n))))))

However, you may notice that the last two cond clauses have very similar code. To make code more concise, we can modify our code to the following implementation:

(define a-harm
  (lambda (n)
    (if (zero? n) 0
     ((if (even? n) - +) (a-harm (- n 1)) (/ 1 n))))))

1.2 Quote

We’ve talked about the quote special form which prevents the evalpig from evaluating expressions in Racket. However, there is a sort of “shorthand” for the quote special form which allows you to use a single quote ‘ rather than using the full (quote ....) syntax. Examples of this are shown here:

- (quote CS17) ⇒ 'CS17
- (quote (1 2 3)) ⇒ '(1 2 3)

This shorthand for the quote special form is an example of *syntactic sugar*, according to which one syntactic form can be replaced with another so as to make the code more concise or more readable. Another example is the use of the keyword else in a cond expression as the condition of the last clause in place of #true.

2 Pattern Matching

In pattern matching, we have a *pattern* and a *text*. We are looking for occurrences of the pattern in the text. What constitutes an occurrence depends on the problem.
2.1 Example Problem: prefix?

**Input:** two lists of symbols, \textbf{pat} and \textbf{text}

**Output:** 
- \#true if the symbols in \textbf{pat} occur in \textbf{text}
  - in same order as in \textbf{pat}
  - consecutively
  - starting at beginning of \textbf{text}

**Example:**

\[(\text{prefix? '}(c b a) \ '}(c b a x y)) \Rightarrow \#true\]

2.1.1 Code

Here is our initial code for \texttt{prefix?}:

\[
\begin{align*}
\text{(define prefix?}
  \text{(lambda (pat text)}
  \text{(cond}
      \text{((empty? pat) #true)}
      \text{(true}
        \text{(if (empty? text)}
          \text{#false}
          \text{(if (equal? (car pat) (car text)}
            \text{(if (prefix? (cdr pat) (cdr text))}
              \text{#true}
              \text{#false)}
          \text{#false))))})
\end{align*}
\]

The value of the third if expression is the same as the value of that if expression’s condition, so we can replace that if expression with the condition:

\[
\begin{align*}
\text{(define prefix?}
  \text{(lambda (pat text)}
  \text{(cond}
      \text{((empty? pat) #true)}
      \text{(true}
        \text{(if (empty? text)}
          \text{#false}
          \text{(if (equal? (car pat) (car text)}
            \text{(prefix? (cdr pat) (cdr text))}
              \text{#true}
              \text{#false)})
          \text{#false))))})
\end{align*}
\]

The value of the second if expression is true if the condition \((\text{equal? (car pat) (car text)})\) is true and the value of the if-true expression \((\text{prefix? (cdr pat) (cdr text)})\) is true, and is false otherwise. Therefore this if expression can be replaced with an and expression, which more directly (and succinctly) says the same thing:

\[
\text{(and (equal? (car pat) (car text)} (prefix? (cdr pat) (cdr text))))}
\]
The result of the replacement is this:

```scheme
(define prefix?
  (lambda (pat text)
    (cond
      ((empty? pat) #true)
      (#true
        (if (empty? text)
            #false
            (and (equal? (car pat) (car text))
                 (prefix? (cdr pat) (cdr text)))))))
```

The purpose of the remaining `if` expression is to dispose of the case when `text` is the empty list; the `and` expression requires that it is not empty. A cleaner way to achieve the same end is to consider `(cons? text)` to be an additional condition required for the match to be true: we replace the whole `if` expression by the `and` expression with that additional condition:

```scheme
(and (cons? text)
     (equal? (car pat) (car text))
     (prefix? (cdr pat) (cdr text)))
```

Note that because of what we call the *short-circuiting* behavior of the `and` special form, if the first condition `(cons? text)` evaluates to false, the remaining conditions will not be evaluated.

Using this simplification, we obtain the following code:

```scheme
(define prefix?
  (cond
    ((empty? pat) #true)
    (#true
      (and (cons? text)
           (equal? (car pat) (car text))
           (prefix? (cdr pat) (cdr text))))))
```

Take one more look at the code. The occurrences of `#true` should cue you to suspect that understanding the logic can lead to a further simplification/clarification. The logic of the code is that the output should be `#true` if the pattern is empty or if the value of the `and` expression is true, and should be `#false` otherwise. We therefore replace the `cond` expression with an `or` expression:

```scheme
(define prefix?
  (or (empty? pat)
      (and (cons? text)
           (and (equal? (car pat) (car text))
                (prefix? (cdr pat) (cdr text))))))
```
2.2 Example Problem: sublist?

Input: two lists of symbols, pat and text
Output: #true if the symbols in pat occur in text consecutively

2.2.1 Code

Using prefix?, we can implement sublist? as follows:

\[
\text{(define sublist?}
\begin{align*}
\text{(lambda (pat text) } & \\
\text{ (or (prefix? pat text) (sublist? pat (cdr text))))})
\end{align*}
\]

3 Internal Representation of Lists

A list as we imagine it is a mathematical idea. It exists only in our minds. There must be some way for a list to be represented physically inside a computer.

Here we will outline the traditional Lisp representation of lists. Understanding this representation will help you understand why car, cdr, and cons take constant time whereas length takes linear time. It will also explain why Lisp does not provide a convenient way to “cons” an item onto the end of a list.

A computer’s memory is a long sequence of memory locations. At each location a number can be stored. The locations are consecutively numbered. (For simplicity, we consider the numbering to start at 1.) The number of a memory location is called its address, in analogy to the addresses of buildings on a street.

<table>
<thead>
<tr>
<th>address</th>
<th>number stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>13</td>
</tr>
</tbody>
</table>
A pointer is a number interpreted as the address of a memory location. Let’s see how pointers can be used to make a list.

Lisp interprets the memory as being divided into pairs of consecutive memory locations. Each pair forms a cons cell. Memory locations 1 and 2 form a cons cell, memory locations 3 and 4 form a cons cell, and so on. The first (odd-numbered) memory locations store items in a list. The second (even-numbered) memory locations store addresses.

Start by looking at the cons cell consisting of memory locations 1 and 2.

Memory location 1 stores a 7, which we will interpret as the number 7. Memory location 2 stores a 9, which we will interpret as a pointer to the memory location whose address is 9.

To further explore the list being represented, let’s “follow the pointer”: look at the cons cell consisting of memory locations 9 and 10. Memory location 9 stores a 10, which we will interpret as the number 10. Memory location 10 stores a 15, which we will interpret as a pointer to the memory location whose address is 15.

Let’s look at the cons cell consisting of memory locations 15 and 16. Memory location 15 stores a 5, which we will interpret as the number 5. Memory location 16 stores a 13, which we will interpret as a pointer to the memory location whose address is 13.

Let’s look at the cons cell consisting of memory locations 13 and 14. Memory location stores a 10, which we will interpret as the number 10. Memory location 14 stores a 0. We would like to interpret this as a pointer but there is no memory location whose address is 0. We interpret the 0 as an indication that the process of following pointers has come to an end. The cons cell consisting of memory locations 13 and 14 is the last in the sequence.

Consider the numbers we saw in this process: 7, 10, 5, 10. Thus the cons cells we considered represent the list (7 10 5 10).

Let’s try another. Consider the cons cell consisting of memory locations 7 and 8. The number stored at memory location 7 is 1 and the pointer stored at memory location 8 is 11. We follow the pointer, and look next at the cons cell consisting of memory locations 11 and 12. Memory location 11 stores the number 14, and memory location 12 stores the pointer 0. The 0 pointer is traditionally called the null pointer (“null” being a synonym for “zero” or “nothing”). Thus the list represented by these cons cells is (1 14).

In Lisp, when a variable is bound to a list value, inside the program that actually executed Lisp (called the Lisp interpreter), the list value is represented by the pointer giving the address of the first cons cell. For example, given the contents of the memory as outlined above, the pointer 1 represents the list (7 10 5 10) and the pointer 7 represents the list (1 14). We refer to this as the internal representation because it happens inside the interpreter; it is not visible to us, the users of Lisp.

You should verify for yourself, examining the contents of memory, that the pointer 3 represents the list (6 5 10). Note that the representation of this list shares some cons cells with the representation of the list (7 10 5 10). Shared structure is a common occurrence, and reflects the way in which these lists were constructed.

Similarly, the pointer 5 represents the list (1 7 10 5 10). The internal representation of this list

1 Following pointers is like a treasure hunt. You are given an initial clue, which directs you to the first location. At the first location, you get a piece of the final clue and a hint where to find the next clue, and so on at each location until you reach the end.
includes in its entirety the internal representation of the list (7 10 5 10).

### 3.1 Interpreting the contents of the first memory location in a cons cell

The number in the first memory location of a cons cell can be interpreted in several ways:

- as a number (in case the item is a number)
- as a symbol (in case the item is a symbol)
- as a pointer to another cons cell (in case the item is itself a list)

... and so on.

How does the computer know whether to interpret the number, e.g., as a pointer or an integer? We are not going into detail, but one way to store numbers would be to assign different ranges to different data types.

### 3.2 car, cdr, and cons

Given what we know about internal representation now, let’s revisit car, cdr, and cons.

Consider car. Given a pointer to a cons cell, the car operation returns the value stored in the first memory location of that cons cell.

The cdr operation is similar: given a pointer to a cons cell, the cdr operation returns the value stored in the second memory location of that cons cell.

The value empty, which is supposed to be the empty list, is represented by a null pointer.

Finally, what does cons do? The Lisp interpreter keeps its own list of pointers to cons cells that are currently unused, called the free list. When the Lisp interpret is asked to cons an item x onto a list L, it extracts from the free list the address of an unused cons cell. It stores x in the first memory location of the cons cell, stores the pointer representing L in the second memory location of the cons cell, and returns the address of the cons cell.

As more cons operations take place, the free list contains the addresses of fewer and fewer unused cells. There is a process called garbage collection that happens every now and then; it finds cons cells that your program will never use, and adds them to the free list.

Now you can see why car, cdr, and cons each take constant time, whereas length? takes linear time.

You should also be able to see why adding an element to the end of a list does not take constant time: the Lisp interpreter would have to follow pointers from the beginning of the end to find the last cons cell in the internal representation of the list. Furthermore, suppose one did add an element to the end of the list (7 10 5 10) represented by pointer 1 by replacing the null pointer in memory location 14 with a pointer to another cons cell and storing the new element in that cons
cell. That would (perhaps inadvertently) add the same element to the list (6 5 10) represented by the pointer 3 and to the list (1 7 10 5 10) represented by the pointer 7.

4 More Order of Growth

**Big Omega** The terminology imagines that the input size \( n \) is slowly increasing over time. So another way to say \( f(n) \) is \( O(g(n)) \) is this: \( f(n) \) grows no faster as \( n \) increases than \( g(n) \) does. We refer to this as *order of growth*. We say that order of growth of \( f(\cdot) \) is no greater than the order of growth of \( g(\cdot) \).

Note that in this sense the big-O notation is a kind of less-than-or-equal to. There is a corresponding greater-than-or-equal to: big \( \Omega \), pronounced “Omega”.

We say a function \( f(n) \) is \( \Omega(g(n)) \) if there exist nonnegative constants \( n_0 \) and \( c \) such that, for every value of \( n \) that is greater than \( n_0 \), \( f(n) \geq c \cdot g(n) \).

For example, the function \( f(n) = 3n + 10 \) is \( \Omega(n) \), but so is the function \( g(n) = 2n^2 + 5n + 10 \). In fact, \( g(n) \) is also \( \Omega(n^2) \).

Let’s practice:

<table>
<thead>
<tr>
<th>function</th>
<th>is it ( O(n) )?</th>
<th>is it ( \Omega(n) )?</th>
<th>is it ( O(n^2) )?</th>
<th>is it ( \Omega(n^2) )?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) = 3n + 7 )</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>( f(n) = 2\sqrt{n} + 1 )</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>( f(n) = n^{1.5} )</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>( f(n) = 300n^2 + 1000n )</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>( f(n) = n^2 + 7\sqrt{n} )</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>( f(n) = n^3 + 2n^2 )</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>( f(n) = 2^n )</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).