### A Cost Model

For Scheme

<table>
<thead>
<tr>
<th>Expression</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier/variable</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Math</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>any number*</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(+ E₁ E₂)</td>
<td>time(E₁) + time(E₂) + 1</td>
<td>space(E₁) + space(E₂)</td>
</tr>
<tr>
<td></td>
<td>* assumes fixed size numbers</td>
<td>+ very generous bound; can you improve it?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Props</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>any boolean</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(if E₁ E₂ E₃)</td>
<td>time(E₁) + 1 + {time(E₂) if true, (\min) (space(E₁), (\frac{\text{space(E₂)}}{\text{time(E₂)}}))} (\frac{\text{space(E₃)}}{\text{time(E₃)}}) if false</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>List</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(cons E₁ E₂)</td>
<td>time(E₁) + time(E₂) + 1</td>
<td>space(E₁) + space(E₂) + 3</td>
</tr>
<tr>
<td>(empty? E)</td>
<td>time(E) + 1</td>
<td>space(E)</td>
</tr>
<tr>
<td>(first E)</td>
<td>time(E) + 1</td>
<td>space(E)</td>
</tr>
<tr>
<td>(rest E)</td>
<td>time(E) + 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Structures</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(make S E₁ ... Eₙ)</td>
<td>time(E₁) + ... + time(Eₙ) + 1</td>
<td>(n+1+n\cdot \min) (space(E₁), ... , space(Eₙ))</td>
</tr>
<tr>
<td>(S? E)</td>
<td>time(E) + 1</td>
<td>space(E)</td>
</tr>
<tr>
<td>(S = E)</td>
<td>time(E) + 1</td>
<td>space(E)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(F E₁ ... Eₙ)</td>
<td>time(E₁) + ... + time(Eₙ) + 1</td>
<td>n \cdot \min) (space(E₁), ... , space(Eₙ)) + time ((\text{body}(F))) + space ((\text{body}(F)))</td>
</tr>
</tbody>
</table>

Where \(\text{body}(F)\) is the body of the function, evaluated after binding formal to actual parameters.
Notes on "A Cost Model for Scheme"

- We have passed lightly over numbers. In most languages, numbers are
  fixed-width. Scheme numbers are more like lists in that they can be
  arbitrarily large, bounded only by the computer's memory. A further
  subtlety with numbers—especially relevant when their size is not fixed—
  is that the value of a number can be exponentially larger than its size.

- The bounds provided assume a particular implementation (corresponding to
  that of DrScheme). Other implementations of particular operators would
  yield different bounds. For instance, it is possible to make first and
  rest expensive to obtain a cheap appeal (as we will see later this semester).

- We have ignored the n-ary generalizations of operations like +, but their
  cost can be thought of as the natural generalization of the binary
  operation over a list (of arguments).

- We typically use + for time and max for space. This represents a sequential
  world-view. In a parallel system, we sometimes use max for time (i.e., the time
  of the longest-running parallel computation) and + for space (i.e., the space necessary
  when all the parallel computations are executing). Even in a parallel setting, however,
  we still find ourselves adding time but not space because space is a
  renewable resource (once no longer necessary for one purpose it can be used
  for another) but time is not (once used, we cannot reclaim it).

- It is critical to understand that call-by-value languages, such as Scheme and Java,
  do not copy complex values on parameter and function calls or identifier lookup.
Example: Define \( \text{append} \)

\[
(\text{define \text{append} \ l_1 \ l_2}) \]

\[
(\text{if \ (empty? \ l_1)} \]
\[
\ l_2 \]
\[
(\text{cons \ (first \ l_1)} \]
\[
(\text{append \ (rest \ l_1) \ l_2))) \]
\]

Given arbitrary lists \( l_1 \) and \( l_2 \):

\[
\text{time} \ [\text{append} \ l_1 \ l_2] = \text{time} \ [\text{(empty? \ l_1)}] + 1 + \begin{cases} \text{time} \ [\ l_2 \] & \text{if \ true} \\
\text{time} \ [(\text{cons} ...) \] & \text{if \ false} \end{cases} \]

\[
= 2 + 1 + \text{time} \ [(\text{cons} \ \text{(first} l_1) \]
\[
(\text{append \ (rest } l_1) \ l_2))] \]
\]

\[
= 3 + \text{time} \ [(\text{first} \ l_1)] + \text{time} \ [(\text{append} \ (\text{rest} \ l_1) \ l_2)] + 1 \]

\[
= 3 + 2 + \text{time} \ [(\text{append} \ (\text{rest} \ l_1) \ l_2)] + 1 \]
\[
= 6 + \text{time} \ [(\text{append} \ (\text{rest} \ l_1) \ l_2)] \]
\]

That is, given \( l_1 \) and \( l_2 \), we obtain a recursive call on \( \text{(rest} \ l_1) \) and \( l_2 \) after \( 6 \) time units.

Since the time \( (6 \) units) is independent of the actual values in the list, we can focus on just the length of \( l_1 \). What happens when it's empty? (we assumed not, earlier)

\[
\text{time} \ [(\text{append} \ \text{empty} \ l_2)] \]
\[
= \text{time} \ [(\text{empty} \ l_1)] + 1 + \text{time} \ [l_2] \]
\]

Thus, the time taken by \text{append} is independent of the second argument.

Let \( T(k) \) be the time consumed by \( \text{append} \ l_1 \ l_2 \) where \( l_1 \) is of size \( k \).

We then have:

\[
T(0) = 4 \]
\[
T(k) = 6 + T(k-1) \quad \text{for} \ k > 0 \]
\[
\Rightarrow T(k) = 6k + 4 \quad \text{for all} \ k \geq 0 \]
\]

or, \( T(k) \sim k \), i.e., \text{append} takes time linear in the length of its first argument.
Example: \( \text{Max} \) (without helper)

\[
\text{(define (max l)}
\]
\[
\text{(if (empty? (rest l)}
\]
\[
\text{(first l)}
\]
\[
\text{(if (> (first l) (max (rest l)))}
\]
\[
\text{(first l)}
\]
\[
\text{(max (rest l))))}
\]

Given an arbitrary non-empty list \( l \):

\[
\text{time \([\text{max } l]\) = time \([\text{empty? (rest l)}]\) + 1 + \begin{cases} \text{time \([\text{first l}\)] if true} \\ \text{time \([\text{if} \ldots]\) if false - assume} \end{cases}
\]
\[
= 3 + 1 + \text{time \([\text{if (> (first l) (max (rest l)))}\]}
\]
\[
\text{(first l)}
\]
\[
\text{(max (rest l)))}\]
\[
= 4 + \text{time \([\text{if (> (first l) (max (rest l)))}\]}
\]
\[
+ 1 + \begin{cases} \text{time \([\text{first l}\)] if true} \\ \text{time \([\text{if} \ldots]\) if false} \end{cases}
\]
\[
\text{comparison (first l)}
\]
\[
= 4 + 1 + 2 + \text{time \([\text{max (rest l)}]\]}
\]
\[
+ \text{time \([\text{max (rest l)}]\)]
\]

Clearly the false case dominates the true case. Being pessimistic, let's assume that case.

\[
= 8 + \text{time \([\text{max (rest l)}]\]}
\]
\[
+ \text{time \([\text{max (rest l)}]\)]
\]

By assuming the first element is not the maximum (a strong assumption — in the worst case, it assumes the highest element is the last one in the list), we see that the above relation holds. [N.B. For appeal, we assumed nothing about the actual values in the \( \text{(first l)} \) list. Here we have made a strong assumption!] When the list has only one element, it is easy to see we need some constant \( c \) number of operations.

Let \( T(n) \) be the time consumed by \( \text{max l} \) where \( l \) has \( n \) elements. In the worst case:

\[
T(1) = c, \quad T(n) = 8 + 2 \cdot T(n-1) \text{ for } n > 1
\]

Thus \( T(n) = 8 \cdot 2^n + 16 - c \) or \( T(n) \sim 2^n \).
Example: \textsc{Max} (with helper)

\begin{align*}
\text{(define (max l)} & \quad \text{(define (gt\textsubscript{of} n\_1 n\_2)} \\
\text{ (if (empty? (rest l)))} & \quad \text{ (if (> n\_1 n\_2)} \\
\text{\hspace{1cm} (first l))} & \quad \text{\hspace{1cm} n\_1} \\
\text{\hspace{1cm} (gt\textsubscript{of} (first l) \quad (max (rest l)))))} & \quad \text{\hspace{1cm} n\_2))}
\end{align*}

\begin{enumerate}
\item \textbf{Given an arbitrary non-empty list l:}
\item \textit{time} \left[ \text{(max l)} \right] = 3 + 1 + \begin{cases} \text{time} \left[ \text{(first l)} \right] & \text{if true} \\
\text{time} \left[ \text{(gt\textsubscript{of} \ldots)} \right] & \text{if false - assume}
\end{cases}
= 4 + \text{time} \left[ \text{(gt\textsubscript{of} (first l) (max (rest l))}] \right.
\item \textit{Note that (first l) \& (max (rest l)) will both evaluate to numbers;}
\item \textbf{for arbitrary numbers} \(n\_1, n\_2\):
\item \textit{time} \left[ \text{(gt\textsubscript{of} n\_1 \ldots \_2)} \right] = \text{time} \left[ (> n\_1 n\_2) \right] + 1 + \begin{cases} \text{time[n\_1]} & \text{if true} \\
\text{time [\ldots]} & \text{if false}
\end{cases}
= 3 + 1 + \begin{cases} 1 & \text{if true} \\
1 & \text{if false}
\end{cases}
= 5
\item \textbf{Since} \textit{time} \left[ \text{(first l)} \right] = \text{time} \left[ \text{(rest l)} \right], \text{the time of (max (rest l))}
\item \textbf{clearly dominates in the two arguments; the other is a constant}
\item \textbf{If we take} \(T(n) = \text{the time consumed by (max l)}\) \text{where} \(l\text{ has} n\text{ elements,}
\item \textit{T} (1) = c, \textbf{for some small constant} \(c,\)
\item \(T (n) = 5 + 2 + T(n-1)\)
\item \textbf{Thus} \(T (n) = 7n + c, \text{ for} n \geq 1\)
\item \textbf{or,} \(T(n) \sim n\)
\end{enumerate}

\textbf{Note:} \textit{We did not make any assumptions about the location of the maximum element;}
\item \textit{indeed, the two branches in gt\textsubscript{of} are symmetric in time. Thus, the analysis}
\item \textit{of this version of \textsc{max} is more "robust" : it applies to all inputs.} \textit{Appeared}