A Cost Model for Scheme

time: expr → num [time units]
space: expr → num [space units]

	Expression	Time	Space
	identifier / variable	1	0
Numbers	any number*	1	1
	(+ E_1 E_2)	$time(E_1) + time(E_2) + 1$	$space(E_1) + space(E_2)$ †
Booleans	any boolean	1	1
	(if E_1 E_2 E_3)	$time(E_1) + 1 + \begin{cases} time(E_2) & \text{if true} \\ time(E_3) & \text{if false} \end{cases}$	$\max\left(space(E_1), \begin{cases} space(E_2) & \text{if true} \\ space(E_3) & \text{if false} \end{cases}\right)$
Lists	empty	1	1
	(cons E_1 E_2)	$time(E_1) + time(E_2) + 1$	$space(E_1) + space(E_2) + 3$
	(empty? E)	$time(E) + 1$	$space(E)$
	(first E)	$time(E) + 1$	$space(E)$
	(rest E)	$time(E) + 1$	$space(E)$
Structures	(make-S $E_1 \cdots E_n$)	$time(E_1) + \cdots + time(E_n) + 1$	$n + 1 + n \cdot \max(space(E_1), \ldots, space(E_n))$
	(S? E)	$time(E) + 1$	$space(E)$
	(S-f E)	$time(E) + 1$	$space(E)$
Functions (Application)	(F $E_1 \cdots E_n$)	$time(E_1) + \cdots + time(E_n) + 1 + time(body(F))$	$n \cdot \max(space(E_1), \ldots, space(E_n)) + space(body(F))$

* assumes fixed size numbers

† a very generous bound; can you improve it?

where 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 sublety 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 *append* (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~~ ~~call~~ function calls or identifier lookup.

— x —

EXAMPLE : APPEND

```
(define (append l1 l2)
    (if (empty? l1)
        l2
        (cons (first l1)
              (append (rest l1) l2))))
```

Given arbitrary lists l_1 & l_2:

time[(append l_1 l_2)]

$= \text{time}[(\text{empty? } l_1)] + 1 + \begin{cases} \text{time}[l_2] & \text{if true} \\ \text{time}[(\text{cons} \ldots)] & \text{if false} \end{cases}$ – let's assume this case for now

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

|| time[(empty? l_1)] = time[l_1] + 1 = 1 + 1 = 2

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

|| time[(first l_1)] = time[l_1]+1 = 1+1 = 2

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

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

That is, given l_1 & l_2, we obtain a recursive call on (rest l_1) & 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.)

time[(append empty l_2)]   where empty is the value passed for l_1

$= \text{time}[(\text{empty? } l_1)] + 1 + \text{time}[l_2]$ (the true branch)

$= 2 + 1 + 1 = 4$

Thus, the time taken by append is independent of the second argument.

Let $T(k)$ = the time consumed by (append l_1 l_2) where l_1 is of size (length) k.

We then have: $T(0) = 4$

$T(k) = 6 + T(k-1)$ for $k > 0$

$\Rightarrow T(k) = 6k + 4$ for all $k \geq 0$

or, $T(k) \sim k$, i.e., append takes time linear in the length of its first argument ▪

EXAMPLE: MAX (without helper)

```
(define (max l)
  (if (empty? (rest l))
      (first l)
      (if (> (first l) (max (rest l)))
          (first l)
          (max (rest l))))))
```

Given an arbitrary non-empty list l:

$$\text{time}[\texttt{(max l)}] = \text{time}[\texttt{(empty? (rest l))}] + 1 + \begin{cases} \text{time}[\texttt{(first l)}] & \text{if true} \\ \text{time}[\texttt{(if ...)}] & \text{if false - assume} \end{cases}$$

$$= 3 + 1 + \text{time}\left[\begin{array}{l} \texttt{(if (> (first l) (max (rest l)))} \\ \quad \cancel{\texttt{(first l)}} \\ \quad \texttt{(max (rest l)))} \end{array}\right]$$

$$= 4 + \text{time}[\texttt{(> (first l) (max (rest l)))}] + 1 + \begin{cases} \text{time}[\texttt{(first l)}] & \text{if true} \\ \text{time}[\texttt{(max (rest l))}] & \text{if false} \end{cases}$$

$$= 4 + \overset{\text{comparison}}{1} + \overset{\texttt{(first l)}}{2} + \text{time}[\texttt{(max (rest l))}] + 1 + \begin{cases} \text{time}[\texttt{(first l)}] & \text{if true} \\ \text{time}[\texttt{(max (rest l))}] & \text{if false} \end{cases}$$

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

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

$$= 8 + 2 \cdot \text{time}[\texttt{(max (rest l))}]$$

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 append, we assumed nothing about the actual values in the (first) 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 (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: MAX (with helper)

```
(define (max l)
  (if (empty? (rest l))
      (first l)
      (gtof (first l)
            (max (rest l)))))

(define (gtof n1 n2)
  (if (> n1 n2)
      n1
      n2))
```

Given an arbitrary non-empty list l:

$$\text{time}[(\text{max } l)] = 3+1+\begin{cases}\text{time}[(\text{first } l)] & \text{if true}\\ \text{time}[(\text{gtof} \ldots)] & \text{if false} \quad \text{- assume}\end{cases}$$

$$= 4 + \text{time}[(\text{gtof } (\text{first } l)\ (\text{max } (\text{rest } l)))]$$

Note that (first l) & (max (rest l)) will both evaluate to numbers; for arbitrary numbers n_1 & n_2:

$$\text{time}[(\text{gtof } n_1\ n_2)] = \text{time}[(>\ n_1\ n_2)] + 1 + \begin{cases}\text{time}[n_1] & \text{if true}\\ \text{time}[n_2] & \text{if false}\end{cases}$$

$$= 3+1+\begin{cases}1 & \text{if true}\\ 1 & \text{if false}\end{cases}$$

$$= 5$$

Since time[(first l)] = time[(rest l)], the time of (max (rest l)) clearly dominates in the two arguments; the other is a constant

If we take $T(n)$ = the time consumed by (max l) where l has n elements,

$T(1) = c_1$ for some small constant c_1

$$T(n) = \underbrace{5}_{\text{time for gtof once args are computed}} + \underbrace{2}_{\text{time to compute first arg}} + T(n-1)$$

Thus $T(n) = 7n + c_1$ for $n \geq 1$

or, $T(n) \sim n$

Note: We did not make any assumptions about the location of the maximum element; indeed, the two branches in gtof are symmetric in time. Thus, the analysis of this version of max is more "robust": it applies to all inputs, à la append.