Delayed evaluation also allows us a neat implementation of suspensions.

The following definition of an infinite list of integers clearly fails:

\[
\text{(define (inflist i)} \\
\quad \text{(cons i (inflist (+ i 1)))})
\]

But with use of delays we get the desired effect in finite time:

\[
\text{(define (inflist i)} \\
\quad \text{(cons i} \\
\quad \text{(delay (inflist (+ i 1)))})
\]

Now a call like \text{(inflist 1)} creates

\[
\begin{array}{c}
\downarrow \\
1 \\
\end{array} \\
\begin{array}{c}
\downarrow \\
\text{promise for} \\
\text{(inflist 2)}
\end{array}
\]
We need to slightly modify how we explore suspended infinite lists. We can’t redefine `car` and `cdr` as these are far too fundamental to tamper with.

Instead we’ll define `head` and `tail` to do much the same job:

```
(define head car)
(define (tail L)
    (force (cdr L)))
```

`head` looks at `car` values which are fully evaluated.

`tail` forces one level of evaluation of a delayed `cdr` and saves the evaluated value in place of the suspension (promise).
Given

\[(\text{define IL (inflist 1)})\]

\[(\text{head (tail IL)) returns 2 and expands IL into} \]

\[\text{promise for (inflist 3)}\]
Exploiting Parallelism

Conventional procedural programming languages are difficult to compile for multiprocessors. Frequent assignments make it difficult to find independent computations.

Consider (in Fortran):

```
do 10 I = 1,1000
   X(I) = 0
   A(I) = A(I+1)+1
   B(I) = B(I-1)-1
   C(I) = (C(I-2) + C(I+2))/2
10 continue
```

This loop defines 1000 values for arrays X, A, B and C.
Which computations can be done in parallel, partitioning parts of an array to several processors, each operating independently?

- \( \text{x}(I) = 0 \)
  Assignments to \( \text{x} \) can be readily parallelized.

- \( \text{A}(I) = \text{A}(I+1) + 1 \)
  Note that each computation of \( \text{A}(I) \) uses an \( \text{A}(I+1) \) value that is yet to be changed. Thus a whole array of new \( \text{A} \) values can be computed from an array of “old” \( \text{A} \) values in parallel.

- \( \text{B}(I) = \text{B}(I−1) − 1 \)
  This is less obvious. Each \( \text{B}(I) \) uses \( \text{B}(I−1) \) which is defined in terms of \( \text{B}(I−2) \), etc. Ultimately all new \( \text{B} \) values depend only on \( \text{B}(0) \) and \( I \). That is, \( \text{B}(I) = \text{B}(0) - I \). So this
computation can be parallelized, but it takes a fair amount of insight to realize it.

- \( C(I) = \frac{(C(I-2) + C(I+2))}{2} \)

It is clear that even and odd elements of \( C \) don’t interact. Hence two processors could compute even and odd elements of \( C \) in parallel. Beyond this, since both earlier and later \( C \) values are used in each computation of an element, no further means of parallel evaluation is evident. Serial evaluation will probably be needed for even or odd values.
Exploiting Parallelism in Scheme

Assume we have a shared-memory multiprocessor. We might be able to assign different processors to evaluate various independent subexpressions.

For example, consider

\[(\text{map } (\text{lambda}(x) ((*) 2 x)) \ (1\ 2\ 3\ 4\ 5))\]

We might assign a processor to each list element and compute the lambda function on each concurrently:

```
Processor 1
  1  2  3  4  5

  2  4  6  8  10
Processor 5
```