To see the usefulness of futures, consider the usual definition of Scheme's map function:

```
(define (map f L)
  (if (null? L)
      ()
      (cons (f (car L))
            (map f (cdr L)))))
```

If we have a call

```
(map slow-function long-list)
```

where `slow-function` executes slowly and `long-list` is a large data structure, we can expect to wait quite a while for computation of the result list to complete.

Now consider `fastmap`, a version of `map` that uses futures:

```
(define (fastmap f L)
  (if (null? L)
      ()
      (cons
        (future (f (car L)))
        (fastmap f (cdr L))
      ))
)
```

Now look at the call

```
(fastmap slow-function long-list)
```

We will exploit a useful aspect of futures—they can be cons'ed together without delay, even if the computation isn't completed yet. Why? Well a cons just stores a pair of pointers, and it really doesn't matter what the pointers reference (a future or an actual result value).

The call to `fastmap` can actually return before any of the call to `slow-function` have completed:

Eventually all the futures automatically transform themselves into data values:

Note that `pcall` can be implemented using futures. That is, instead of

```
(pcall F X Y Z)
```

we can use

```
((future F) (future X) (future Y) (future Z))
```

In fact the latter version is actually more parallel—execution of `F` can begin even if all the parameters aren’t completely evaluated.
Another Example of Futures

The following function, \texttt{partition}, will take a list and a data value (called \texttt{pivot}). \texttt{partition} will partition the list into two sublists:

(a) Those elements \( \leq \) pivot
(b) Those elements \( > \) pivot

\begin{verbatim}
(define (partition pivot L)
  (if (null? L)
    (cons () ()
    (let ((tail-part
      (partition pivot (cdr L))))
      (if (<= (car L) pivot)
        (cons
          (cons (car L) (car tail-part))
          (cdr tail-part))
        (cons
          (car tail-part)
          (cons (car L) (cdr tail-part))))
    ) ) )

 butt one change isn't enough! We soon access the \texttt{car} and \texttt{cdr} of \texttt{tail-part}, which forces us to wait for its computation to complete. To avoid this delay, we can place the four references to \texttt{car} or \texttt{cdr} of \texttt{tail-part} into futures too:

\begin{verbatim}
(define (partition pivot L)
  (if (null? L)
    (cons () ()
    (let ((tail-part
      (partition pivot (cdr L))))
      (if (<= (car L) pivot)
        (cons
          (cons (car L) (car tail-part))
          (cdr tail-part))
        (cons
          (car tail-part)
          (cons (car L) (cdr tail-part))))
    ) ) )
\end{verbatim}

We want to add futures to \texttt{partition}, but where?

It makes sense to use a future when a computation may be lengthy and we may not need to use the value computed immediately.

What computation fits that pattern? The computation of \texttt{tail-part}. We'll mark it in a blue box to show we plan to evaluate it using a future:

\begin{verbatim}
define (partition pivot L)
  (if (null? L)
    (cons () ()
    (let ((tail-part
      (partition pivot (cdr L))))
      (if (<= (car L) pivot)
        (cons
          (cons (car L) (car tail-part))
          (cdr tail-part))
        (cons
          (car tail-part)
          (cons (car L) (cdr tail-part))))
    ) ) )
\end{verbatim}
Now we can build the initial part of the partitioned list (that involving pivot and \((\text{car } L)\) independently of the recursive call of \text{partition}, which completes the rest of the list. For example,

\[
(\text{partition 17 '}(5\ 3\ 8\ \ldots))
\]

creates a future (call it \text{future1}) to compute

\[
(\text{partition 17 '}(3\ 8\ \ldots))
\]

It also creates \text{future2} to compute \((\text{car } \text{tail-part})\) and \text{future3} to compute \((\text{cdr } \text{tail-part})\). The call builds

\[
\text{future2}\quad \text{future3}
\]

\[
5\quad \text{future2}
\]

\[
\text{future3}
\]

---

**Reading Assignment**

- Roosta: Section 13.3
- Introduction to Standard ML (linked from class web page)
- Webber: Chapters 5, 7, 9, 11

---

**ML — Meta Language**

\text{SML} is Standard ML, a popular ML variant.

\text{ML} is a functional language that is designed to be efficient and type-safe. It demonstrates that a functional language need not use Scheme’s odd syntax and need not bear the overhead of dynamic typing.

\text{SML}'s features and innovations include:

1. Strong, compile-time typing.
2. Automatic type inference rather than user-supplied type declarations.
3. Polymorphism, including “type variables.”

\textbf{4. Pattern-directed Programming}

\begin{verbatim}
fun len([]) = 0
| len(a::b) = 1+len(b);
\end{verbatim}

\textbf{5. Exceptions}

\textbf{6. First-class functions}

\textbf{7. Abstract Data Types}

\begin{verbatim}
coin of int
bill of int
check of string*real;
val dime = coin(10);
\end{verbatim}

A good ML reference is “Elements of ML Programming,” by Jeffrey Ullman (Prentice Hall, 1998)
SML is Interactive

You enter a definition or expression, and SML returns a result with an inferred type.
The command
use "file name";
loads a set of ML definitions from a file.
For example (SML responses are in blue):
21;
val it = 21 : int
(2 div 3);
val it = 0 : int
true;
val it = true : bool
"xyz";
val it = "xyz" : string

Basic SML Predefined Types

- Unit
  Its only value is (). Type unit is similar to void in C; it is used where a type is needed, but no "real" type is appropriate. For example, a call to a write function may return unit as its result.

- Integer
  Constants are sequences of digits. Negative values are prefixed with a ~ rather than a - (– is a binary subtraction operator). For example, ~123 is negative 123.
  Standard operators include
  +  -  *  div  mod
  <  >  <=  >=

- Real
  Both fractional (123.456) and exponent forms (10e7) are allowed. Negative signs and exponents use ~ rather than – (~10.0e~12).
  Standard operators include
  +  -  *  /  
  < > <= >=
  Note that = and <> aren’t allowed! (Why?)
  Conversion routines include
  real(int) to convert an int to a real,
  floor(real) to take the floor of a real,
  ceil(real) to take the ceiling of a real.
  round(real) to round a real,
  trunc(real) to truncate a real.

- Strings
  Strings are delimited by double quotes. Newlines are \n, tabs are \t, and \" and \\
escape double quotes and backslashes. E.g. "Bye now\n" The ^ operator is concatenation.
"abc" ^ "def" = "abcdef"
The usual relational operators are provided: < > <= >= = <>
. **Characters**

Single characters are delimited by double quotes and prefixed by a #. For example, #"a" or #"\t". A character is not a string of length one. The `str` function may be used to convert a character into a string. Thus `str(#"a") = "a"`

. **Boolean**

Constants are `true` and `false`. Operators include `and also` (short-circuit and), `orelse` (short-circuit or), `not`, `=`, and `<>`. A conditional expression, `(if boolval v1 else v2)` is available.

**Tuples**

A tuple type, composed of two or more values of any type is available. Tuples are delimited by parentheses, and values are separated by commas. Examples include:

- `(1,2);`  
  ```
  val it = (1,2) : int * int
  ```
- `("xyz",1=2);`  
  ```
  val it = ("xyz",false) : string * bool
  ```
- `(1,3.0,false);`  
  ```
  val it = (1.3.0,false) : int * real * bool
  ```
- `(1,2,(3,4));`  
  ```
  val it = (1,2,(3,4)) : int * int * (int * int)
  ```

Equality is checked componentwise:

- `(1,2) = (0+1,1+1);`  
  ```
  val it = true : bool
  ```
- `(1,2,3) = (1,2)` causes a compile-time type error (tuples must be of the same length and have corresponding types to be compared).

# selects the i-th component of a tuple (counting from 1). Hence

- `#2(1,2,3);`  
  ```
  val it = 2 : int
  ```

**Lists**

Lists are required to have a single element type for all their elements; their length is unbounded. Lists are delimited by `[` and `]` and elements are separated by commas. Thus `[1,2,3]` is an integer list. The empty (or null) list is `[]` or `nil`. The cons operator is `::`

Hence `[1,2,3] ≡ 1::2::3::[]`

Lists are automatically typed by ML:

- `val it = [1,2];`  
  ```
  val it = [1,2] : int list
  ```
Cons

Cons is an infix operator represented as ::
The left operand of :: is any value of type T.
The right operand of :: is any list of type T list.
The result of :: is a list of type T list.
Hence :: is polymorphic.

[] is the empty list. It has a type 'a list. The symbol 'a, read as "alpha" or "tic a" is a type variable.
Thus [] is a polymorphic constant.

List Equality

Two lists may be compared for equality if they are of the same type. Lists L1 and L2 are considered equal if:
(1) They have the same number of elements
(2) Corresponding members of the two lists are equal.

List Operators

hd ≡ head of list operator ≈ car
tl ≡ tail of list operator ≈ cdr
null ≡ null list predicate ≈ null?
@ ≡ infix list append operator ≈ append