Functional Programming

All functions are pure functions
No assignment and no mutable data types (except for re-define)
Name–value bindings are permanent
Advantages of functional programming:
• The value of an expression is independent of the order in which sub-expressions are evaluated
• Sub-expressions can safely be evaluated in parallel or lazily
• Referential transparency: The value of an expression does not change when we substitute one of its subexpression with the value of that subexpression.
But... Can we make basic loops efficient?
Yes!

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{def } \text{exp}(b, n):)</td>
<td>(\Theta(n))</td>
<td>(\Theta(n))</td>
</tr>
<tr>
<td></td>
<td>(\text{if } n == 0: )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(\text{return } 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(\text{return } b * \text{exp}(b, n-1))</td>
<td></td>
</tr>
<tr>
<td>(\text{def } \text{exp}(b, n):)</td>
<td>(\Theta(n))</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td></td>
<td>(\text{for } _\text{in range}(n):)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(\text{total }= 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(\text{return } \text{total }= \text{total }* b)</td>
<td></td>
</tr>
</tbody>
</table>

Tail Recursion

From the Revised^7 Report on the Algorithmic Language Scheme:

"Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

```
(define (factorial n k)
  (if (= n 0) k
    (factorial (- n 1) (* k n)))))
```

```
def factorial(n, k):
    while n > 0:
        n, k = n - 1, k * n
    return k
```

Example: Length of a List

```
(define (length s)
  (define (length-iter s n)
    (if (null? s) n
      (+ 1 (length-iter (cdr s) (+ 1 n)))))
  (length-iter s 0))
```

A call expression is not a tail call if more computation is still required in the calling procedure.
Linear recursions can often be re-written to use tail calls.

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      ((length-iter (cdr s) (+ 1 n)))))
  (length-iter s 0))
```

Recursive call is a tail call
Not a tail context
Eval with Tail Call Optimization

The return value of the tail call is the return value of the current procedure call.

Therefore, tail calls shouldn't increase the environment size.

In the interpreter, recursive calls to `scheme_eval` for tail calls must instead be expressed iteratively.

Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If expression:** `(if <predicate> <consequent> <alternative>)`
- **And and or:** `(and <e1> ... <en>)`, `(or <e1> ... <en>)`
- **Cond expr'n:** `(cond (<p1> <e1>) ... (<pn> <en>) (else <e>))`

The value of an **if** expression is the value of a sub-expression.

- Evaluate the **predicate**.
- Choose a sub-expression: `<consequent>` or `<alternative>`.
- Evaluate that sub-expression in place of the whole expression.

Evaluation of the `tail` context does not require a recursive call. E.g., replace `(if false 1 (+ 2 3))` with `(+ 2 3)` and repeat.

Example: Reduce

```scheme
(define (reduce fn s start)
  (if (null? s)
    start
    (reduce fn
      (cdr s)
      (fn start (car s)))))
```

Recursive call is a tail call.
Other calls are not; constant space depends on `fn`.

```scheme
(reduce + '(3 4 5) 2) 120
```

```scheme
(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) 5 4 3 2)
```

An Analogy: Programs Define Machines

Programs specify the logic of a computational device

```
factorial 1 1
5 1 120

- factorial
```

Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine

```
5 120

(define (factorial n)
  (if (zero? n) 1 (* n (factorial (- n 1)))))
```

Our Scheme interpreter is a universal machine

A bridge between the data objects that are manipulated by our programming language and the programming language itself

Internally, it is just a set of manipulation rules
Interpretation in Python

**eval**: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.

**exec**: Executes a statement in the current environment. Doing so may affect the environment.

```python
eval('2 + 2')
exec('def square(x): return x * x')
```

**os.system('python <file>')**: Directs the operating system to invoke a new instance of the Python interpreter.

**Demo**