Procedural vs Functional Style

procedural = imperative

Abstracting Expressions

• Scale a 2D-point \( p \) by a factor \( c \)

\[
\text{(define scale } \quad (\text{lambda } (c \ p) \\
\text{(cons \ (cons \ (* \ c \ \text{first} \ p)) \ \\
\text{(list \ (* \ c \ \text{second} \ p)))) })
\]

• Can abbreviate a fixed and finite number of function applications for readability, modifiability, and reusability.

Procedural/Imperative vs Functional

- **Program**: a sequence of instructions for a von Neumann m/c.
- Computation by instruction execution.
- **Repetition**: Iteration.
- Modifiable or updateable variables.

- **Program**: a collection of function definitions (m/c independent).
- Computation by term rewriting.
- **Repetition**: Recursion.
- “Assign-only-once” variables.

Functional Style: Illustration

- **Definition**: Equations
  
  \[
  \begin{align*}
  \text{sum}(0) &= 0 \\
  \text{sum}(n) &= n + \text{sum}(n-1)
  \end{align*}
  \]

- **Computation**: Substitution and Replacement
  
  \[
  \begin{align*}
  \text{sum}(2) &= 2 + \text{sum}(2-1) \\
  &= \ldots \\
  &= 3
  \end{align*}
  \]
Paradigm vs Language

• Procedural Style
  i := 0; sum := 0;
  while (i < n) do
    begin
      i := i + 1;
      sum := sum + i
    end;
  – Storage efficient

• Functional Style
  func sum(n:int) : int;
  begin
    if n = 0 then 0
    else n + sum(n-1)
  end;
  – No Side-effect

Role of Variables

• Imperative (read/write)

<table>
<thead>
<tr>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

• Functional (read only)

<table>
<thead>
<tr>
<th>n1</th>
<th>n2</th>
<th>n3</th>
<th>sum1</th>
<th>sum2</th>
<th>sum3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Bridging the Gap

• A tail recursive definition can be automatically optimized for space by translating it into an equivalent while-loop.

  func sum(n : int, r : int) : int;
  begin
    if n = 0 then r
    else sum(n-1, n+r)
  end
  – So, Scheme does not have loop-constructs.

Iteration vs Recursion

Recursion = Iteration + Stack

Iteration = Tail Recursion