An Elegant Weapon for a More Civilized Age
Solving an Easy Problem

- What are the input types? What is the output type? Give example input/output pairs.
- Which input represents the domain of the recursion, i.e., which input becomes smaller? How is problem size defined?
- What function is used to produce smaller problem instances?
- What functions can construct the output type?
- What is the output value when the problem is smallest?
Solving an Easy Problem (contd.)

- How can a problem instance be reduced to one or more smaller problem instances? What function creates the output value?
- Is your case analysis correct and complete?
- If an input can be of more than one type, e.g., sometimes an atom, sometimes a pair, then you will need to provide a case for each type.
Solving a Hard Problem

• Identify one (or more) sub problems that would make the hard problem into an easy problem if solved.

• Give example input/output pairs for helper functions which would solve the sub problems.

• Define the helper functions and test your solutions.

• If any of the sub problems are hard themselves then identify additional helper functions which would permit you to solve them.
Debugging Imperative Programs

- An imperative program is understood by the programmer as a process which transforms the state of an abstract machine.
- The state of the abstract machine is comprised of the values of variables and the contents of the stack and heap.
- By observing how the values of variables change over time, the programmer verifies that the process is defined correctly.
Debugging Functional Programs

• A functional program is understood by the programmer as the definition of the solution to a problem.

• A functional programmer fixes errors by reformulating this definition using new terms.

• These terms are the solutions of sub problems each of which can be independently verified by testing.

• A functional program is debugged by rewriting it using simpler and simpler pieces until each piece is demonstrably correct.
Compiling Function Calls in C

- A function's *local environment* consists of the values bound to its parameters and local variables.

- When a function is called, the local environment of the calling function is pushed onto the *call stack*.

- The saved local environment is termed an *activation record*.

- A *return* statement pops the call stack and restores the local environment.
Recursion is Expensive!

- Repeatedly saving and restoring the contexts associated with function calls requires time.
- The saved contexts cause the call stack to grow.
n! Two Ways

```c
int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}

int fact(int n, int acc) {
    if (n == 0) return acc;
    else return fact(n-1, acc*n);
}
```
n! Two Ways

int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}

context used
context saved
context discarded

int fact(int n, int acc) {
    if (n == 0) return acc;
    else return fact(n-1, acc*n);
}
n! Two Ways

int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}

int fact(int n, int acc) {
    if (n == 0) return acc;
    else return fact(n-1, acc*n);
}
Tail Call Optimization

- A good compiler* will recognize the pointlessness of the push-pop sequence and compile the tail call as a jump.
- This saves the expense of saving and restoring the local environment.
- The call stack does not grow.
- Tail recursion is as efficient as iteration!

*gcc optimizes tail calls when you use \(-O3\) or higher.
Fibonacci Numbers Three Ways

```c
int fib(int n) {
    if (n < 2) return n;
    else return fib(n-1) + fib(n-2);
}
```
Fibonacci Numbers Three Ways

```c
int fib(int n) {
    if (n < 2) return n;
    else return fib(n-1) + fib(n-2);
}
```

\(O(2^n)\) space and time complexity!
int fib(int n) {
    int temp;
    int acc0 = 0, acc1 = 1;
    while (n > 0) {
        temp = acc0;
        acc0 = acc1;
        acc1 += temp;
        n--;
    }
    return acc0;
}
BOREDOM: the desire for desires.

--LEO TOLSTOY
Fibonacci Numbers Three Ways

```c
int fib(int n, int acc0, int acc1) {
    if (n == 0) return acc0;
    else return fib(n-1, acc1, acc0+acc1);
}
```
Fibonacci Numbers Three Ways

int fib(int n, int acc0, int acc1) {
    if (n == 0) return acc0;
    else return fib(n-1, acc1, acc0+acc1);
}

O(1) space and O(n) time and no temporary variables!
Tail Positions

- *Tail positions* are shown in red:
  - `(if pred val₁ val₂ )`
  - `(cond (pred₁ val₁ ) ... (pred₁⁻¹ val₁⁻¹ )
    (else valₙ ) )`
  - `(or pred₁ pred₂ ... pred₁⁻¹ predₙ )`
  - `(and pred₁ pred₂ ... pred₁⁻¹ predₙ )`

- These positions within special forms in tail positions are also tail positions!
Identify the Tail Positions

• (and a b (if x y z) c)
• (if a (if x y z) (if u v w))
• (or a (and a b))
• (if a (or b c d) e)
• (cond (a (if a b c)) (x y) (else z))
• (if (if a b c) x y)
• (cond (x y) ((if a b c) d) (else (or u v)))
Identify the Tail Positions

- (and a b (if x y z) c)
- (if a (if x y z) (if u v w))
- (or a (and a b))
- (if a (or b c d) e)
- (cond (a (if a b c)) (x y) (else z))
- (if (if a b c) x y)
- (cond (x y) ((if a b c) d) (else (or u v)))
let, let* and letrec special-forms

<table>
<thead>
<tr>
<th>collateral</th>
<th>sequential</th>
<th>recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{let} ((\text{var}_1 \ \text{val}_1)))</td>
<td>(\text{let*} ((\text{var}_1 \ \text{val}_1)))</td>
<td>(\text{letrec} ((\text{var}_1 \ \text{val}_1)))</td>
</tr>
<tr>
<td>(\text{var}_2 \ \text{val}_2)</td>
<td>(\text{var}_2 \ \text{val}_2)</td>
<td>(\text{var}_2 \ \text{val}_2)</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>(\text{var}_N \ \text{val}_N))</td>
<td>(\text{var}_N \ \text{val}_N))</td>
<td>(\text{var}_N \ \text{val}_N))</td>
</tr>
<tr>
<td>(\text{body}))</td>
<td>(\text{body}))</td>
<td>(\text{body}))</td>
</tr>
</tbody>
</table>

scope of \(\text{var}_1\)
let is just lambda!

(let ((var₁ val₁) (var₂ val₂) . . . (varₙ valₙ)) body)

((lambda (var₁ var₂ ... varₙ) body) (val₁ val₂ ... valₙ))
Example

(let ((x 2) (y 3)) (+ x y))

((lambda (x y) (+ x y)) 2 3)
let* is just nested let's!

(let* ((var₁ val₁) (let ((var₁ val₁))
(var₂ val₂) (let ((var₂ val₂))
  .
  .
  .
(varₙ valₙ)) (let ((varₙ valₙ))
body)) ...))
Example

(let* ((x 2) (y 3)) (+ x y))

(let ((x 2)) (let ((y 3)) (+ x y)))

((lambda (x) ((lambda (y) (+ x y)) 3) 2))
Also Tail Positions!

<table>
<thead>
<tr>
<th>collateral</th>
<th>sequential</th>
<th>recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(let ((var₁₁ val₁₁) (var₂₂ val₂₂)) ... (varₙₙ valₙₙ)) body)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>(let* ((var₁₁ val₁₁) (var₂₂ val₂₂)) ... (varₙₙ valₙₙ)) body)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>(letrec ((var₁₁ val₁₁) (var₂₂ val₂₂)) ... (varₙₙ valₙₙ)) body)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lisp

Lisp is over half a century old and it still has this perfect, timeless air about it.

I wonder if the cycles will continue forever.

A few coders from each new generation rediscovering the Lisp arts.

These are your father's parentheses.

Elegant weapons for a more... civilized age.