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?
• 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) subproblems that would make the hard problem into an easy problem if solved.
- Give example input/output pairs for helper functions which would solve the subproblems.
- Define the helper functions and test your solutions.
- If any of the subproblems are themselves hard, 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 subproblems 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.
void bar(int i) {
    int j = 0;
    while (j++ < i) putChar('.');
    return;
}

int foo(int i) {
    int j = 7;   // local variable
    bar(j);     // function call
    return i + j;  // restored context used
}
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

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

<table>
<thead>
<tr>
<th>n!</th>
<th>5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>120</td>
</tr>
</tbody>
</table>

```c
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);
}

text context used

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

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

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

\(^*\text{gcc}\) optimizes tail calls when you use \(-O3\) or higher.
Compiler Object Code

```
gcc -O1

fact:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl 8(%ebp), %ecx
movl 12(%ebp), %edx
movl %ecx, %eax
testl %edx, %edx
je .L1
leal -1(%edx), %eax
movl %eax, 4(%esp)
movl %ecx, %eax
imull %edx, %eax
movl %eax, (%esp)
call fact
.popl %ebp
ret

.L1:
movl %ebp, %esp
.popl %ebp
ret

 gcc -O4

fact:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
movl 12(%ebp), %edx
.p2align 4,,15
.L8:
testl %edx, %edx
je .L9
imull %edx, %eax
dcl %edx
jmp .L8
.L9:
.popl %ebp
ret
```

function call → jump → multiplication → subtraction → test → subtraction → test → multiplication → subtraction → jump

loop body

function body
Fibonacci Numbers Three Ways

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!
Fibonacci Numbers Three Ways

```c
int fib(int n) {
    int temp;
    int acc0 = 0, acc1 = 1;
    while (n > 0) {
        temp = acc0;
        acc0 = acc1;
        acc1 += temp;
        n--;
    }
    return acc0;
}
```

<table>
<thead>
<tr>
<th>n</th>
<th>acc0</th>
<th>acc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>→ 5</td>
<td>8</td>
</tr>
</tbody>
</table>
BOREDOM: the desire for desires.

--LEO TOLSTOY
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);
}
```

```
5 0 1
4 1 1
```
int fib(int n, int acc0, int acc1) {
    if (n == 0) return acc0;
    else return fib(n-1, acc1, acc0+acc1);
}

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

int fib(int n, int acc0, int acc1) {
    if (n == 0) return acc0;
    else return fib(n-1, acc1, acc0+acc1);
}
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 \(val_1\) val_2)
  - (cond (pred_1 val_1) ... (pred_{N-1} val_{N-1}) (else val_N))
  - (or pred_1 pred_2 ... pred_{N-1} pred_N)
  - (and pred_1 pred_2 ... pred_{N-1} pred_N)

- 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 b c d)) (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 b c d)) (x y) (else z))
- (if (if a b c) x y)
- (cond (x y) ((if a b c) d) (else (or u v)))
O(2^n) Space Fibonacci in Scheme

(define fib
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2)))))
  )
(define fib
  (lambda (n acc0 acc1)
    (if (= n 0)
        acc0
        (fib (- n 1) acc1 (+ acc0 acc1))))

  tail position
> (let ((x 2) (y 3)) (+ x y))
5

> (let ((x 2)) (let ((x 3)) (+ x x))
6
  shadowed

> (let ((x 2)) (let ((y x)) (+ y y))
4

> (let* ((x 2) (y x)) (+ y y))
4
let, let* and letrec special-forms

collateral

(\(\text{let} \ (\langle \var_1 \ \text{val}_1 \rangle, \langle \var_2 \ \text{val}_2 \rangle, \ldots, \langle \var_N \ \text{val}_N \rangle) \ \	ext{body} \))

sequential

(\(\text{let*} \ (\langle \var_1 \ \text{val}_1 \rangle, \langle \var_2 \ \text{val}_2 \rangle, \ldots, \langle \var_N \ \text{val}_N \rangle) \ \	ext{body} \))

recursive

(\(\text{letrec} \ (\langle \var_1 \ \text{val}_1 \rangle, \langle \var_2 \ \text{val}_2 \rangle, \ldots, \langle \var_N \ \text{val}_N \rangle) \ \	ext{body} \))

\(\text{scope of } \var_1\)
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!

\[
\begin{align*}
\text{(let* } & ((\text{var}_1 \text{ val}_1)) \quad \text{(let } ((\text{var}_1 \text{ val}_1)) \\
& ((\text{var}_2 \text{ val}_2)) \quad \text{(let } ((\text{var}_2 \text{ val}_2)) \\
& \quad \ldots \\
& ((\text{var}_N \text{ val}_N)) \quad \text{(let } ((\text{var}_N \text{ val}_N)) \\
& \quad \text{body}) \quad \text{body}) \ldots \text{) )}
\end{align*}
\]
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>(\text{let} \ ( (\var_1 \ \text{val}_1 ) \ (\var_2 \ \text{val}_2 ) \ \ldots \ (\var_N \ \text{val}_N )) \ (\text{body}) )</td>
<td>(\text{let\text{*}} \ ( (\var_1 \ \text{val}_1 ) \ (\var_2 \ \text{val}_2 ) \ \ldots \ (\var_N \ \text{val}_N )) \ (\text{body}) )</td>
<td>(\text{let\text{rec}} \ ( (\var_1 \ \text{val}_1 ) \ (\var_2 \ \text{val}_2 ) \ \ldots \ (\var_N \ \text{val}_N )) \ (\text{body}) )</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 RE-DISCOVERING THE LISP ARTS.

THESE ARE YOUR FATHER’S PARENTHESES

ELEGANT WEAPONS

FOR A MORE... CIVILIZED AGE.