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.

Saving and Restoring Contexts

```
Call stack push □  void bar(int i) {
                      int j = 0;
                      while (j++ < i) putChar('.');
 Call stack pop □     return;
                   }
                                   parameter
                                       |
                                       v
                   int foo(int i) {
                      int j = 7; □ local variable
  context saved □     bar(j); □ function call
context restored □    return i + j; □ restored context used
                   }
```

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);
}
```

n! Two Ways

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

5	1
4	5
3	20
2	60
1	120

```
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 disgarded

context saved

context restored

restored context used

```
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 disgarded

context saved

context restored

restored context used

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

context disgarded

context saved

context restored

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!

*`gcc` optimizes tail calls when you use `-O3` or higher.

Compiler Object Code

gcc -O1

```
                 fact:
push □                  pushl  %ebp
                        movl   %esp, %ebp
subtraction □           subl   $8, %esp
                        movl   8(%ebp), %ecx
                        movl   12(%ebp), %edx
                        movl   %ecx, %eax
test □                  testl  %edx, %edx
                        je  .L1
                        leal   -1(%edx), %eax
                        movl   %eax, 4(%esp)
                        movl   %ecx, %eax
multiplication □        imull  %edx, %eax
                        movl   %eax, (%esp)
function call □         call   fact
                 .L1:
                        movl   %ebp, %esp
pop □                   popl   %ebp
                        ret
```

□ function body

gcc -O4

```
                 fact:
push □                  pushl%ebp
                        movl %esp, %ebp
                        movl 8(%ebp), %eax
                        movl 12(%ebp), %edx
                        .p2align 4,,15
                 .L8:
test □                  testl%edx, %edx
                        je .L9
multiplication □        imull%edx, %eax
subtraction □           decl %edx
jump □                  jmp   .L8
                 .L9:
pop □                   popl %ebp
                        ret
```

loop body

Fibonacci Numbers Three Ways

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

n	0	1	2	3	4	5	6	7	8	...
fib(n)	0	1	1	2	3	5	8	13	21	...

Fibonacci Numbers Three Ways

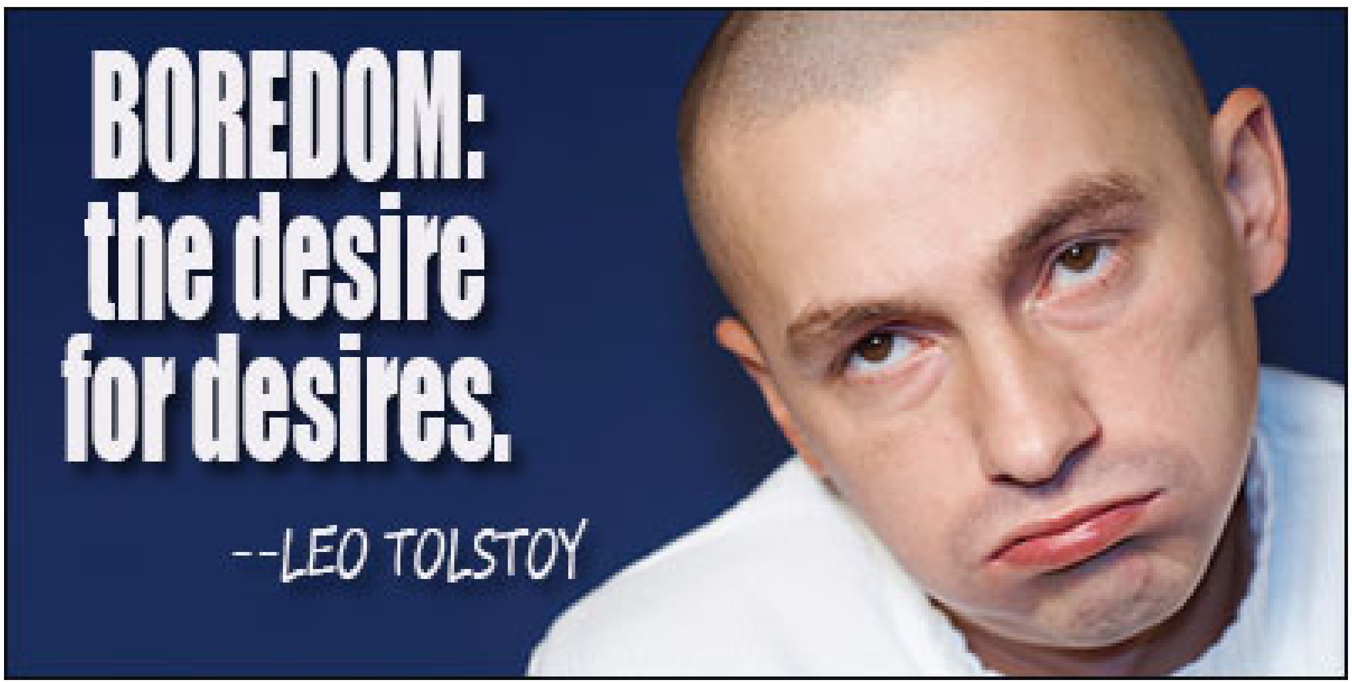
```
int fib(int n) {
  if (n < 2) return n;
  else return fib(n-1) + fib(n-2);
}
```

$O(2^n)$ space and time complexity!

Fibonacci Numbers Three Ways

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

n	acc0	acc1
→ 5	0	1
4	1	1
3	1	2
2	2	3
1	3	5
0	→ 5	8


BOREDOM:
the desire
for desires.
--LEO TOLSTOY

Fibonacci Numbers Three Ways

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

Fibonacci Numbers Three Ways

→ 5	0	1
4	1	1
3	1	2
2	2	3
1	3	5
0	→ 5	8

```
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* *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

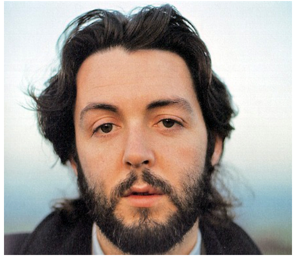
```
(define fib
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2))))))
```

non-tail position   non-tail position

O(1) Space Fibonacci in Scheme

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

tail position



Let It Be

```
> (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

```
(let ((var_1 val_1)
      (var_2 val_2)
          .
          .
          .
      (var_N val_N))
  body)
```

sequential

```
(let* ((var_1 val_1)
       (var_2 val_2)
           .
           .
           .
       (var_N val_N))
  body)
```

recursive

```
(letrec ((var_1 val_1)
         (var_2 val_2)
             .
             .
             .
         (var_N val_N))
  body)
```

scope of var_1

let is just **lambda**!

(**let** ((var_1 val_1)
(var_2 val_2)
.
.
.
(var_N val_N))
body)

((**lambda** (var_1 var_2 ...var_N) *body*)
(val_1 val_2 ...val_N))

Example

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

```
((lambda (x y) (+ x y)) 2 3)
```

let* is just nested **let**'s!

```
(let* ((var_1 val_1)
       (var_2 val_2)
            .
            .
            .
       (var_N val_N))
  body)
```

```
(let ((var_1 val_1))
  (let ((var_2 val_2))
            .
            .
            .
       (let ((var_N val_N))
         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!

collateral

```
(let ((var_1 val_1)
      (var_2 val_2)
         .
         .
         .
      (var_N val_N))
  body)
```

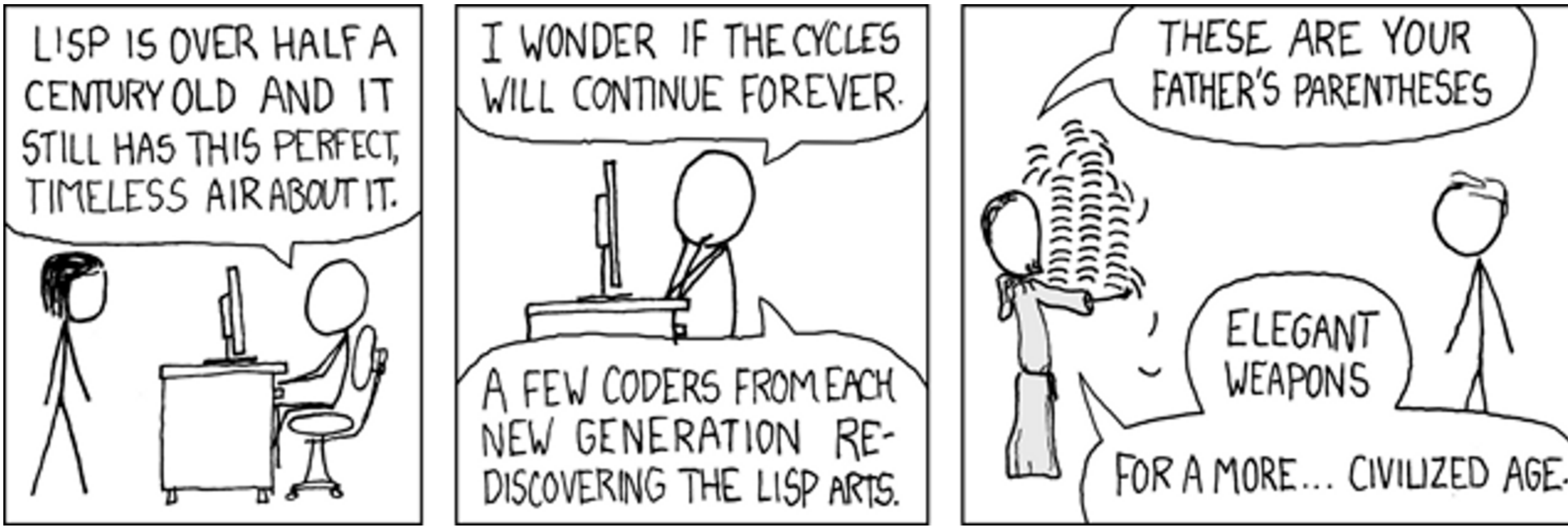
sequential

```
(let* ((var_1 val_1)
       (var_2 val_2)
          .
          .
          .
       (var_N val_N))
  body)
```

recursive

```
(letrec ((var_1 val_1)
         (var_2 val_2)
            .
            .
            .
         (var_N val_N))
  body)
```

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.