3A: More Recursion; Higher-order Functions

CS1101S: Programming Methodology

Martin Henz

August 24, 2016

Leader Board

Some Administration

Use media

The Source Academy (comments)
IVLE Discussion Forum
Facebook (for background and social aspects of the module)

Avenger consultation

Make use of discussion groups
Make appointment with your avenger

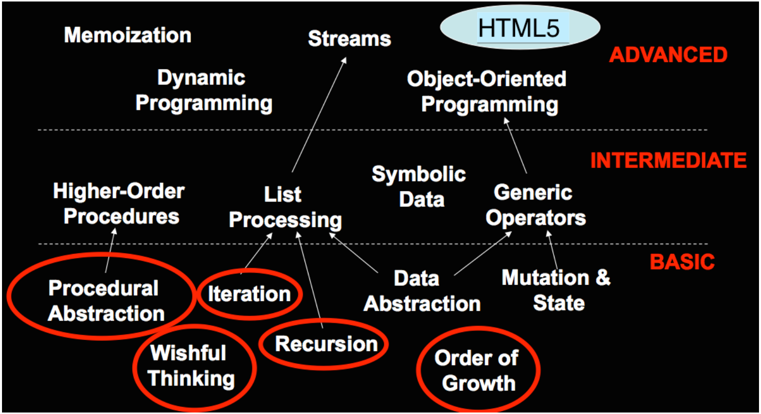
Consultation with Martin and Kok Lim

Don't be afraid to make an appointment:
`henz@comp.nus.edu.sg, 96991279,`
`lowkl@comp.nus.edu.sg, 93388638`

Words of Advice

- Think, then program
- Less is more
- It’s a marathon, not a sprint

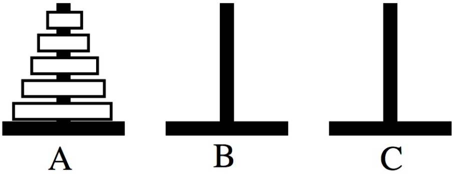
Roadmap So Far



Sequences of Statements

```
alert('first this'); alert('and then that');
```

First Example: Towers of Hanoi



- Given: A tower of disks in increasing size
- Wanted: How to move all disks from A to B, using C
- The rule: Never put a larger disk on top of a smaller one

In The Source (Week 3)

```
function move_tower(size, from, to, extra) {
    if (size === 0) {
        return true;
    } else {
        move_tower(size - 1, from, extra, to);
        display("move from " + from + " to " + to);
        return move_tower(size-1, extra, to, from);
    }
}
move_tower(3, "A", "B", "C");
```

In The Source (Week 4)

```
function move_tower(size, from, to, extra) {
   if (size === 0) {
      ;
   } else {
      move_tower(size - 1, from, extra, to);
      display("move from " + from + " to " + to);
      move_tower(size-1, extra, to, from);
   }
}
move_tower(3, "A", "B", "C");
```

`undefined`

`return`

A function that does not return anything, returns `undefined`.

Empty statement

A single semicolon is a statement that evaluates to `undefined` and whose evaluation has no further effect.

Second Example: Coin Change

- Given: Different kinds of coins (unlimited supply)
- Given: Amount of money in cents
- Wanted: Number of ways to change amount into coins

Highest Denomination

```
function highest_denom(kinds) {
    if      (kinds === 0) { return 0;          }
    else if (kinds === 1) { return 5;          }
    else if (kinds === 2) { return 10;         }
    else if (kinds === 3) { return 20;         }
    else if (kinds === 4) { return 50;         }
    else if (kinds === 5) { return 100;        }
    else                  { display('error');  }
}
```

Think

- Example: Number of ways to change 120 cents
- Idea: Highest coin is either 100 or not 100
- In the first case: Smaller problem
- In the second case: Smaller problem
- Base cases?

Idea in The Source

```
function cc(amount, kinds) {
    if ... /* base cases */
    else {
        return cc(amount, kinds - 1) +
               cc(amount - highest_denom(kinds),
                  kinds);
    }
}
```

Adding Base Cases

```
function cc(amount, kinds) {
    if (amount === 0) {
        return 1;
    } else if (amount < 0 || kinds === 0) {
        return 0;
    } else {
        return cc(amount, kinds - 1) +
               cc(amount - highest_denom(kinds),
                  kinds);
    }
}
```

Third Example: Power Function

Recursive process

```
// raise b to the power of non-neg integer e
function power(b, e) {
    return (e === 0) ? 1 : b * power(b, e - 1);
}
```

Correctness

Follows definition of *power* function

Termination

`e` decreases by 1 each time, leading to $e, e-1, e-2, \ldots, 2, 1, 0$

Power Function: Iterative Process

```
// raise b to the power of non-neg integer e
function power(b, e) {
    function power_iter(count, product) {
        return (count === 0)
               ? product
               : power_iter(count - 1,
                            b * product); }
    return power_iter(e, 1);
}
```

Power Function: Iterative Process

```
// raise b to the power of non-neg integer e
function power(b, e) {
    function power_iter(count, product) {
        return (count === 0)
               ? product
               : power_iter(count - 1,
                            b * product); }
    return power_iter(e, 1);
}
```

Correctness?
Termination?

Iterative Power: Correctness

```
// raise b to the power of non-neg integer e
function power(b, e) {
    function power_iter(count, product) {
        return (count === 0)
               ? product
               : power_iter(count - 1,
                            b * product); }
    return power_iter(e, 1); }
```

Invariant

At any point, in `power_iter(count, product)`, we have

$$\texttt{product} = \texttt{b}^{\texttt{e}-\texttt{count}}$$

Iterative Power: Termination

```
// raise b to the power of non-neg integer e
function power(b, e) {
    function power_iter(count, product) {
        return (count === 0)
               ? product
               : power_iter(count - 1,
                            b * product); }
    return power_iter(e, 1); }
```

`count` decreases by 1

leading to $n, n-1, n-2, \ldots, 2, 1, 0$

Fast Power Function

```
// raise b to the power of non-neg integer e
function fast_power(b, e) {
    if (e === 0) {
        return 1;
    } else if (is_even(e)) {
        return fast_power(b * b, e / 2);
    } else {
        return b * fast_power(b, e - 1);
    }
}
```

Fast Power Function: Correctness

```
// raise b to the power of non-neg integer e
function fast_power(b, e) {
    if (e === 0) {
        return 1;
    } else if (is_even(e)) {
        return fast_power(b * b, e / 2);
    } else {
        return b * fast_power(b, e - 1);
}   }
```

Correctness

- $b^0 = 1$
- $b^e = (b^2)^{\frac{e}{2}}$
- $b^e = bb^{e-1}$

Fast Power Function: Termination

```
// raise b to the power of non-neg integer e
function fast_power(b, e) {
    if (e === 0) {
        return 1;
    } else if (is_even(e)) {
        return fast_power(b * b, e / 2);
    } else {
        return b * fast_power(b, e - 1);
}   }
```

Termination

- Exponent `e` remains non-negative integer
- Exponent `e` strictly decreases in each call
- The set of integers is well-founded (Noetherian); there is no infinite decreasing chain of elements.

Homework: Fast Power, Iterative

- Time: $O(\log n)$
- Space: $O(1)$
- Correctness: What invariant?
- Bring your solution/questions to next recitation class

Variables Hold Intermediate Values

Example

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$$

Compute $f(2, 3)$

```
function f(x, y) {
    var a = 1 + x * y;
    var b = 1 - y;
    return x * square(a) + y * b + a * b;
}

f(2, 3);
```

Take a typical `fractal` function...

```
function fractal(p, n) {
    if (n === 0) {
        return p;
    } else {
        return beside(p,
                      stack(fractal(p, n - 1),
                            fractal(p, n - 1)));
    }
}
```

...and compare with the following `fractal` function

```
function fractal(p, n) {
    if (n == 0) {
        return p;
    } else {
        var p1 = fractal(p, n - 1);
        return beside(p, stack(p1, p1));
    }
}
```

Passing Functions to Functions

```
function f(g, x) {
    return g(x);
}

f( function(y) {
        return y + 1;
   },
   7
 );
```

Passing Functions to Functions

```
function f(g, x) {
    return g(g(x));
}

f( function(y) {
        return y + 1;
   },
   7
 );
```

Passing Functions to Functions

```
var z = 1;

function f(g) {
    return g(z);
}

f( function(y) {
       return y + z;
   }
 );
```

Variable Scope

```
function f(g) {
    var z = 4;
    return g(z);
}

f( function(y) {
       var z = 2;
       return y + z;
   }
 );
```

Returning Functions from Functions

```
function make_adder(x) {
    return function(y) {
               return x + y;
           };
}

var adder_four = make_adder(4);
adder_four(6);
```

Returning Functions from Functions

```
function make_adder(x) {
    return function(y) {
               return x + y;
           };
}

( make_adder(4) )(6);
```

Returning Functions from Functions

```
function make_adder(x) {
    return function(y) {
               return x + y;
           };
}

var adder_1 = make_adder(1);
var adder_2 = make_adder(2);
adder_1(10);
adder_2(20);
```

Look at this function

```
function sum_numbers(a, b) {
    return (a > b) ? 0
                   : a + sum_numbers(a + 1, b);
}
```

...and at this one

```
function sum_cubes(a, b) {
    return (a > b) ? 0
                   : cube(a) + sum_cubes(a + 1, b);
}

function cube(x) {
    return x * x * x;
}
```

...and at computing π

```
function frac_sum(a, b) {
    return (a > b) ? 0
                   : 1 / (a * (a + 2))
                     +
                     frac_sum(a + 4, b);
}

// Approximation of pi.
frac_sum(1, 30) * 8;
```

Abstraction

```
function sum(a, b) {
    return (a > b) ? 0
                   : ⟨compute value with a⟩
                     +
                     sum(⟨next value from a⟩, b);
}
```

In The Source

```
function sum(term, a, next, b) {
    return (a > b) ? 0
                   : term(a)
                     +
                     sum(term, next(a), next, b);
}
```

Applications

```
function sum_numbers(a, b) {
    return sum(function(x) { return x; },
               a,
               function(x) { return x + 1; },
               b);
}

function sum_cubes(a, b) {
    return sum(cube,
               a,
               function(x) { return x + 1; },
               b);
}
```

Applications

```
function frac_sum(a, b) {
    return sum(function(x) {
                   return 1 / (x * (x + 2));
               },
               a,
               function(x) { return x + 4; },
               b);
}

frac_sum(1, 30) * 8;
```

Important Ideas

- *Recursion* is an elegant pattern of problem solving
- *Invariants* can guarantee correctness
- Functions can be *passed to* functions
- Functions can be *returned from* functions
- Higher-order functions are *useful* for building abstractions
  ...and fractals!