2A: Abstraction, Scope, Recursion

CS1101S: Programming Methodology

Martin Henz

August 17, 2016

Recall: Elements of Programming

- Primitives
- Combination
- Abstraction

What makes a good abstraction?

Tasks and subtasks

Answer 1

One that makes it more natural the think about tasks and subtasks.

Example

Houses $\rightarrow$ Bricks?
Houses $\rightarrow$ Walls $\rightarrow$ Bricks!

Underlying principle

“Divide and Conquer”

Simplicity

Answer 2

One that makes programs easier to read and understand.

Simplicity

Answer 2

One that makes programs easier to read and understand.

Example

```
function beside(p1, p2) {
    return quarter_turn_right(
              stack(quarter_turn_left(p2),
                    quarter_turn_left(p1)
                   )
           );
}
```

Familiarity

Answer 3

One that captures common patterns.

Familiarity

Answer 3

One that captures common patterns.

```
var my_cross =
   stack(beside(quarter_turn_right(rcross_bb),
                turn_upside_down(rcross_bb)),
         beside(rcross_bb,
                quarter_turn_left(rcross_bb)));
function make_cross(p) {
    return stack(beside(quarter_turn_right(p),
                        turn_upside_down(p)),
                 beside(p,
                        quarter_turn_left(p)));
}
```

Reuse

Answer 4

One that allows for program reuse.

Reuse

Answer 4

One that allows for program reuse.

Example

```
var pi = 3.141592653589793;
function square(x) {
    return x * x;
}
function circle_area_from_radius(r) {
    return pi * square(r);
}
function circle_area_from_diameter(d) {
    return circle_area_from_radius(d / 2);
}
```

Information Hiding

Answer 5

One that hides irrelevant details.

Information Hiding

Answer 5

One that hides irrelevant details.

Example

```
function persian(n, pic) {
    function beside_n(n) {
        return quarter_turn_left(
                  stackn(n,
                         quarter_turn_right(pic)));
    }
    return ...beside_n...;
}
```

Separation of concerns

Answer 6

One that separates specification from implementation.

Separation of concerns

Answer 6

One that separates specification from implementation.

Example

```
// version 1
function square(x) { return x * x; }

// version 2
function double(x) { return x + x; }
function square(x) {
    return Math.exp(double(Math.log(x)));
}
```

Debugging

Answer 7

One that makes it easy to find errors

Debugging

Answer 7

One that makes it easy to find errors

Example 1

```
function hypotenuse(a, b) {
    return Math.sqrt((a + a) * (b + b));
};
```

Finding Errors

Answer 7

One that makes it easy to find errors

Example 2

```
function sum_of_squares(a, b) {
    return square(x) * square(y);
}
function square(x) {
    return x + x;
}
function hypotenuse(a, b) {
    return Math.sqrt(sum_of_squares(a,b));
};
```

Variable Scope: An Example

```
var x = 10;
function square(x) {
    return x * x;
}
function addx(y) {
    return y + x;
}
square(5) + addx(20);
```

Variable Scope: A Bit of Confusion

```
var pi = 3.141592653589793;
function circle_area_from_radius(r) {
    var pi = 22 / 7;
    return pi * square(r);
}
```

Which *pi*?

Variable Scope: Yet Another Example

```
function hypotenuse(a, b) {
    function sum_of_squares(a, b) {
        return square(a) + square(b);
    }
    return Math.sqrt(sum_of_squares(a, b));
};
```

Variable Scope: Simplified

```
function hypotenuse(a, b) {
    function sum_of_squares() {
        return square(a) + square(b);
    }
    return Math.sqrt(sum_of_squares());
};
```

Simpler version: In case you're wondering

```
function hypotenuse(a, b) {
    var sum_of_squares = square(a) + square(b);
    return Math.sqrt(sum_of_squares);
};
```

Variables in The Source

Mandatory

All variables in The Source must be declared.

Forms of declaration

1. Pre-declared variables (`alert`)
2. **`var`** statements
3. Formal parameters of **`function`** expressions/statements
4. Function variable of **`function`** statements

Scoping rule

A variable occurrence refers to the closest surrounding declaration.

Forms of Declaration

(1) Pre-declared variables

The Source has several variables pre-declared, for the convenience of the programmer, including `alert`, `Math.floor`, `Math.sqrt`, `Math.log`, and `Math.exp`

Forms of Declaration

(2) **var** statements

The scope of a **var** statement is the closest surrounding function definition, or the “top-level”, if there is none.

Example

```
function f(x, y) {
    if (x > 0) {
        var z = x * y;
        return Math.sqrt(z);
    } else {
        ...
    }
}
```

Forms of Declaration

(3) Formal Parameters

The scope of the formal parameters of a function definition is the body of the function.

```
function f(x, y, z) {
    ... x ... y ... z
}
```

Forms of Declaration

(4) Function variable

The scope of the function variable is as if the function was declared with **`var`**.

```
function f(x) {
    ...
}

var f = function(x) {
            ...
        };
```

Finally, the most important rule

Scoping rule

A variable occurrence refers to the closest surrounding declaration.

```
function f(x) {
    return function() {
               var x = ...;
               return function(x) {
                          ...x...
                      };
           };
}
```

A Recursive Function

```
function stackn(n, pic) { // sf: stack_frac
    sf(1/n, pic, stackn(n - 1, pic));
}
```

A Recursive Function

```
function stackn(n, pic) { // sf: stack_frac
    sf(1/n, pic, stackn(n - 1, pic));
}

stackn(2, p);
sf(1/2, p, stackn(1, p));
sf(1/2, p, sf(1/1, p, stackn(0, p)));
sf(1/2, p, sf(1/1, p, sf(1/0, stackn(-1, p))));
...
```

Remarks

Computers will follow orders precisely

We have no choice but to *precisely* describe *how* a computational process should be executed.

Substitution model

A simple model to understand how functions work is to imagine that a function call is repeatedly replaced by the body of the function, where the formal parameter is replaced by the actual argument.

The correct version

```
function stackn(n, pic) {
  return n===1 ? pic
               : sf(1/n, pic, stackn(n - 1, pic));
}
```

The correct version

```
function stackn(n, pic) {
  return n===1 ? pic
               : sf(1/n, pic, stackn(n - 1, pic));
}
```

Obvervation

The solution for n is computed using solution $n-1$,
the solution for $n-1$ is computed using solution $n-2$, etc
until we reach a case that we can solve trivially.

A Recipe

```
function stackn(n, pic) {
  return n===1 ? pic
             : sf(1/n, pic, stackn(n - 1, pic));
}
```

Recipe for recursion

- Figure out a *base case* that we can solve trivially
- Assume that you know how to solve the problem for $n - 1$. How can we solve the problem for n?

Second example: Factorial

Factorial

$$n! = n(n-1)(n-2)\cdots 1$$

After grouping and rewriting, we get

$$\begin{aligned} n! &= n(n-1)! && \text{if } n > 1 \\ &= 1 && \text{if } n = 1 \end{aligned}$$

Translation into The Source

After grouping and rewriting, we get

$$\begin{aligned} n! &= n(n-1)! & \text{if } n > 1 \\ &= 1 & \text{if } n = 1 \end{aligned}$$

In The Source

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n-1);
}
```

Example Execution using Substitution Model

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n-1);
}
```

```
factorial(4)
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Notice the build-up of pending operations

A Closer look at performance

Dimensions of performance

- Time: how long does the program run
- Space: how much memory do we need to run the program

Time for calculating *n*!

Number of operations

grows linearly proportional to *n*.

```
factorial(4)
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Space for calculating *n*!

Deferred operations: Number of “things to remember”

grows linearly proportional to *n*.

```
factorial(4)
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Third example: Fibonacci numbers

Leonardo Pisano Fibonacci

12th century, was interested in the sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Each number is the sum of the previous two.

More precise definition

The function *fib* maps 0 to 0, 1 to 1, and every subsequent natural number n to the sum of the two previous Fibonacci numbers: $\mathit{fib}(n-2) + \mathit{fib}(n-1)$.

What was Fibonacci's most significant achievement?

Computing Fibonacci numbers: A Naive Attempt

Definition

The function *fib* maps 0 to 0, 1 to 1, and every subsequent natural number n to the sum of the two previous Fibonacci numbers: $fib(n-2) + fib(n-1)$.

```
function fib(n) {
    return n <= 1 ? n
                  : fib(n - 1) + fib(n - 2);
}
```

Tree recursion for Fibonacci numbers: Time

Time for function `fib`

The tree grows very quickly when we apply `fib` to larger and larger numbers.

Tree recursion for Fibonacci numbers: Time

Time for function `fib`

The tree grows very quickly when we apply `fib` to larger and larger numbers.

A job for recitations

We will take a closer look at this during the recitations.

Tree recursion for Fibonacci numbers: Space

- At any time computing the tree, we need to remember the path to the current node
- Depth of tree grows linearly with n
- Space consumption grows linearly with n

Summary

- Abstraction techniques
- Variable scope
- Resources for computational processes: time and space
- Kinds of recursion: linear recursion and tree recursion