National University of Singapore
School of Computing
CS1101S: Programming Methodology (JavaScript)
Semester I, 2012/2013

Recitation 0
Functional Abstraction

JavaScript

1. When running in a browser, JavaScript has a function called `alert` predefined. This means that the environment that JavaScript starts with already has a function associated with the symbol `alert`. This function always returns the value `undefined`, the same value that results from evaluating `var` statements and function definition statements. As a side-effect, the function `alert` displays its argument in a pop-up window of the browser. Try

   ```
   alert(100 + 200);
   ```

2. Conditional statements of the form

   `if (`*test1*`)` {*cons-stmt-1*} `else if (`*test2*`)` {*cons-stmt-2*} `else` {*alt-stmt;*}

   evaluate a series of tests in order. If the value of a test is not false, the corresponding consequent is evaluated, otherwise the next test is evaluated. If a test is evaluated as true, succeeding tests will not be evaluated. If all tests evaluate to `false`, the final alternative is evaluated.

   Example:

   ```
   function sign(x) {
       if (x < 0) {
           return -1;
       } else if (x > 0) {
           return 1;
       } else {
           return 0;
       }
   }
   ```

3. Similarly, conditional expressions of the form

   (*test1*) ? *consequent-expr-1 : (test2) ? consequent-expr-2 : alterative-expr*

   evaluate a series of tests in order. If the value of a test is not false, the value of whole conditional expression is the value corresponding consequent, otherwise the next test is performed. If a test evaluates to `true`, succeeding tests will no longer be evaluated. If all tests fail, the value of the whole conditional expression is the value of the remaining *alternative*.

   Example: The function above can be re-written as:

```
function sign(x) {
    return (x < 0) ?
           -1 : (x > 0) ?
                1 : 0;
}
```

Note that in JavaScript, there must not be any newline character between the `return` keyword and the expression.

4. *function* - `function(`*parameters*`){`*body*`}`
   Creates a function with the given parameters and body. Parameters is a comma-separated sequence of names of variables. Body is one or more JavaScript statements. When the function is applied, the body statements are evaluated in order. The function can return a value to the caller using `return`, followed by an expression.

Firefox

1. Start the web console of Firefox using Tools → Web Developer → Web Console.
2. Play with the examples of Lecture 1.
3. Separate the lines of input in the console using ⟨shift⟩ ⟨return⟩.
4. Do not feel discouraged when the console replies "undefined" after you enter a statement. Verify that the environment has a value for a symbol by typing the symbol, followed by ⟨return⟩. If you get anything other than "`ReferenceError:...is not defined`, then the environment has a value for the symbol.

Problems:

1. Evaluate the following statements, assuming `x` is bound to 3, and observe their effect:

```
if (true) { alert(1+1); } else { alert(17); } => 2

if (false) { alert(false); } else { alert(42); } => 42

if (x > 0) { alert(x); } else { alert(-x); } => 3

if (0) { alert(1); } else { alert(2); } => 2

if (x < 0) { alert(7); } else { alert(7); } => 7

if (true) { alert(1); }
else if(y < 1) { alert(false); }
else{ alert("wake up"); } => 1
```

2. Evaluate the following statements:

```
(function(x) { return x; }); => (function (x) {return x;})

(function(x) { return x; })(17); => 17
```

```
(function(x, y) { return x; })(42, 17); => 42

(function(x, y) { return y; })(z, 3); => error

(function(x, y) { return x(y, 3); })((function(a, b) { return a + b; }), 14); => 17
```

3. Suppose we're designing a point-of-sale and order-tracking system for a new burger joint. It is a small joint and it only sells 4 options for combos: Classic Single Combo (hamburger with one patty), Classic Double With Cheese Combo (2 patties), and Classic Triple with Cheese Combo (3 patties), Avant-Garde Quadruple with Guacamole Combo (4 patties). We shall encode these combos as 1, 2, 3, and 4 respectively. Each meal can be *biggie-sized* to acquire a larger box of fries and drink. A *biggie-sized* combo is represented by 5, 6, 7, and 8 respectively, for combos 1, 2, 3, and 4 respectively.

   (a) Write a function named `biggie_size` which when given a regular combo returns a *biggie-sized* version.
   Answer:

   ```
   function biggie_size(meal) { return meal + 4; }
   ```

   (b) Write a function named `unbiggie_size` which when given a *biggie-sized* combo returns a non-*biggie-sized* version.
   Answer:

   ```
   function unbiggie_size(meal) { return meal - 4; }
   ```

   (c) Write a function named `is_biggie_size` which when given a combo, returns true if the combo has been *biggie-sized* and false otherwise.
   Answer:

   ```
   function is_biggie_size(meal) { return meal > 4; }
   ```

   (d) Write a function named `combo_price` which takes a combo and returns the price of the combo. Each patty costs $1.17, and a *biggie-sized* version costs $.50 extra overall.
   Answer:

   ```
   function combo_price(meal) {
       if(is_biggie_size(meal)) {
           return 0.50 + (1.17 * unbiggie_size(meal));
       } else {
           return 1.17 * meal;
       }
   }
   ```

   (e) An order is a collection of combos. We'l encode an order as each digit representing a combo. For example, the order 237 represents a Double, Triple, and *biggie-sized* Triple.
   Write a function named `empty_order` which takes no arguments and returns an empty order which is represented by 0.
   Answer:

   ```
   function empty_order() { return 0; }
   ```

   (f) Write a function named `add_to_order` which takes an order and a combo and returns a new order which contains the contents of the old order and the new combo. For example, `add_to_order(1, 2) -> 12`.
   Answer:

```
function add_to_order(order, combo) {
    return order * 10 + combo;
}
```

(g) Write a function named `order_size` which takes an order and returns the number of combos in the order. For example, `order_size(237) -> 3`. You may find `Math.floor` useful. This functions rounds its argument downwards to the nearest integer. Thus, `Math.floor(5.9)` returns `5` and `Math.floor(-4.1)` returns `-5`.

Answer:

```
function order_size(order) {
    if(order === empty_order()) {
        return 0;
    } else {
        return 1 + order_size(Math.floor(order / 10));
    }
}
```

(h) Write a function named `order_cost` which takes an order and returns the total cost of all the combos. In addition to `Math.floor`, you may find the modulo operator `%` useful.

Answer:

```
function order_cost(order) {
    if(order === empty_order()) {
        return 0;
    } else {
        return combo_price(order % 10) + order_cost(Math.floor(order / 10));
    }
}
```

Notice that the solution is almost identical to `order_size`. The only difference is that instead of adding one for each combo we remove, we add the price of the combo. Note also how we are using the function `combo_price` which we defined earlier.

(i) **Homework:** Write a function named `add_orders` which takes two orders and returns a new order that is the combination of the two. For example, `add_orders(123, 234) -> 123234`. Note that the order of the combos in the new order is not important as long as the new order contains the correct combos. `add_orders(123, 234) -> 122334` would also be acceptable.

(j) **Homework 2:** Write iterative versions of `order_size` and `order_cost`.