National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2013/2014

Recitation 1
Functional Abstraction

JediScript

1. JediScript Week 3 allows for `true` and `false` as primitive expressions. These expressions evaluate to *boolean values* that can be used to take yes/no decisions.

2. JediScript Week 3 provides for the following comparison operators for numbers: `>`, `<`, `>=`, `<=`, `===` and `!==`. They work as expected: You can place them between two expressions. When such a comparison is evaluated, the two expressions are evaluated first. When they evaluate to numbers, the comparison evaluates to either `true` or `false`, depending on the numbers and the operator. For example, executing the program

   ```
   1 + 2 < 5;
   ```

   results in `true`.

3. Conditional statements of the form

   ```
   if (test-1) {
       cons-stmt-1
   } else if (test-2) {
       cons-stmt-2
   } else if (test-3) {
       cons-stmt-3
   } else {
       alt-stmt
   }
   ```

   evaluate a series of tests in order. If a test evaluates to `true`, the corresponding consequent is evaluated, otherwise the next test is evaluated. If a test evaluates to `true`, succeeding tests will not be evaluated. If none of the tests evaluate to true, the final alternative is evaluated.

   Example:

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

4. Similarly, conditional expressions of the form

```
(test-1) ?
 consequent-expr-1 : (test-2) ?
                      consequent-expr-2 : (test-3) ?
                                           consequent-expr-1 : alt-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. In the former case, 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.

5. Function expressions of the form

function(*parameters*){ *body* }

create a function with the given parameters and body. *Parameters* is a comma-separated sequence of names of variables. *Body* is a JediScript statement. When the function is applied, the body statement is executed. The function can return a value to the caller using `return`, followed by an expression.

JFDI Academy

1. If you don't have one yet, get a Facebook account.
2. Use your Facebook ID to log into the JFDI Academy at `http://jedi.ddns.comp.nus.edu.sg/register/into/the/academy/`.
3. Go to the tab "Source", then to "Console". At the bottom of this tab, you find a white input text book, where you can enter JediScript programs, followed by the "enter"/"return" key.
4. This feature of the JFDI Academy is a "read-eval-print loop" (REPL). It reads the input, executes (evaluates) the input as a JediScript program, and prints the result in the text area above the input text book.

Problems:

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

```
if (true) { 1+1; } else { 17; }

if (false) { false; } else { 42; }

if (x > 0) { x; } else { -x; }

if (x === 0) { 1; } else { 2; }

if (x < 0) { 7; } else { 7; }

if (true) { 1; }
else if (y < 1) { 4711; }
else{ 42; }
```

2. Evaluate the following statements:

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

(function(x) { return x; })(17);

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

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

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

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.

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

(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.

(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.

(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.

(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`.