Individual Exercises - Lecture 13

1. Read the articles under additional references.
   The articles will be used as a basis for the discussions in the group exercises.

2. Sebesta chapter 7, p 350-351, exercises 9,13, 19
   9. Assume the following rules of associativity and precedence for expressions:

Precedence	*Highest*	*, /, **not**
		+ , − , &, **mod**
		- (unary)
		=, / =, < , <=, >=, >
		and
	Lowest	**or**, **xor**
Associativity	*Left to right*	

Show the order of evaluation of the following expressions by parenthe- sizing all subexpressions and placing a superscript on the right parenthesis to indicate order. For example, for the expression
a + b * c +d
the order of evaluation would be represented as
((a + (b * c)[1])[2] + d)[3]

Exercise	Original Expression	Expression with Explicit Evaluation Order
A	a * b - 1 + c	$(((a * b)^{1} - 1)^{2} + c)^{3}$
B	a * (b - 1) / c mod d	$(((a * (b - 1)^{1})^{2} / c)^{3} mod\ d)^{4}$
C	(a - b) / c & (d * e / a - 3)	$(((a - b)^{1} / c)^{2} \& (((d * e)^{3} / a)^{4} - 3$
D	-a or c = d and e	$((- a)^{1} or ((c = d)^{2} and\ e)^{3})^{4}$
E	a > b xor c or d <= 17	$(((a > b)^{1} xor\ c)^{3} or\ (d <= 17)^{2})^{4}$
F	-a + b	$(- (a + b)^{1})^{2}$

13. Let the function fun be defined as

```
int fun (int* k) {
    *k += 4;
    return 3 * (*k) - 1;
}
```

Suppose fun is used in a program as follows:

```
void main() {
    int i = 10, j = 10, sum1, sum2;
    sum1 = (i / 2) + fun(&i);
    sum2 = fun(&j) + (j / 2);
}
```

This exercise does not specify which operator precedence and associativity to use. In our solution and the one provided by Sebesta we assume the program to be written in C (as the syntax indicates).

For more information on C: https://en.cppreference.com/w/c/language/eval_order

a. operands in the expressions are evaluated left to right? Sum1: 46, Sum2: 48

b. operands in the expressions are evaluated right to left? Sum1: 48, Sum2: 46

19. Consider the following C program:

```
int fun (int* i) {
    *i += 5;
    return 4;
}

void main(){
    int x = 3;
    x = x + fun(&x);
}
```

What is the value of x after the assignment statement in main, assuming
a. operands are evaluated left to right? X: 7
b. operands are evaluated right to left? X: 12

3. Sebesta chapter 7, p 352, programming exercises 1 and 2

1. Run the code given in Problem 13 (in the Problem Set) on some system that supports C to determine the values of sum1 and sum2. Explain the results.

C program:

```c
#include <stdio.h>
int fun(int* k) {
    *k += 4;
    return 3 * (*k) -1;
}
void main() {
    int i = 10, j = 10, sum1, sum2;
    sum1 = (i / 2) + fun(&i);
    sum2 = fun(&j) + (j / 2);
    printf("Sum1: %d, Sum2: %d \n", sum1, sum2);
}
```

Prints: Sum1: 46, Sum2: 48

The result indicates that the implementation of C used evaluated the expressions from left to right. By evaluating the expression from left to right `i` is divided by 2 before four is added to it so `sum1` is 46. In contrast `sum2` is 48 as 4 is added to `j` before j is divided by 2.

Again for more information on C: https://en.cppreference.com/w/c/language/eval_order

2. Rewrite the program of Programming Exercise 1 in C++, Java, and C#, run them, and compare the results.

C++ Program: the result for C++ is the same as for C.

Java Program: Java uses pass-by-value exclusively. To emulate the behavior of the c program a wrapper class is created and used in place of the integers i and j. This simulates pass-by-reference semantics.

```java
public class JavaVersion {

    private static class IntWrapper {
        int val;

        public IntWrapper(int val) {
            this.val = val;
        }
    }
```

```
    static int fun(IntWrapper wrappedInt){
        wrappedInt.val += 4;
        return 3 * wrappedInt.val - 1;
    }

    public static void main(String[] args) {
        IntWrapper i = new IntWrapper(10), j = new IntWrapper(10);
        int sum1, sum2;
        sum1 = (i.val / 2) + fun(i);
        sum2 = fun(j) + (j.val / 2);
        System.out.println(String.format("Sum1: %d, Sum2: %d", sum1, sum2));
    }
}
```

Prints: sum1 = 46, sum2 = 48

```
class Program { // C# version
    static int fun(ref int k) {
        k += 4;
        return 3 * (k) - 1;
    }
    static void Main(string[] args) {
        int i = 10, j = 10, sum1, sum2;
        sum1 = (i / 2) + fun(ref i);
        sum2 = fun(ref j) + (j / 2);
        Console.WriteLine($"Sum1: {sum1}, Sum2: {sum2}");
    }
}
```

Prints: Sum1 = 46, Sum2 = 48

4. Sebesta chapter 8, p 386, programming exercises 1.a

1.a Rewrite the pseudocode segment using a loop structure in C, C++, Java, or C#

```
   k = (j + 13) / 27
loop:
   if k > 10 then goto  out
   k = k + 1
   i = 3 * k - 1
   goto loop
out: . . .
```

```
// Java code example
int i, j, k;
for (k = (j + 13) / 27; k > 10; k++) {
    i = 3 * k - 1;
}
```

Group Exercises - Lecture 13

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Discuss the article “Python & Java - A Side-by-Side Comparison”
   Find it here:
   https://pythonconquerstheuniverse.wordpress.com/2009/10/03/python-java-a-side-by-side-comparison/
   Consider topics such as productivity and expressiveness, i.e. how much code do you need to write to express something?

3. Discuss the article “go to statement considered harmful” and “Structured Programming with go to Statements”
   Find them here:
   http://doi.acm.org/10.1145/362929.362947
   and
   http://doi.acm.org/10.1145/356635.356640
   Do you consider gotos harmful? Why do you have that opinion?

4. Sebesta chapter 7, p 350, exercises 4
   4. Would it be a good idea to eliminate all operator precedence rules and require parentheses to show the desired precedence in expressions? Why or why not?
   As input to the discussion it is worth looking into how the Lisp languages (such as Clojure, Common Lisp, and Scheme) are designed as these languages operate in that way. In Lisp operators are functions that are applied to arguments, e.g, 2 + 2 becomes (+ 2 2) which is very similar to the function syntax used by imperative languages like C plus(2, 2). So if the goal is to create a language that focuses on functions (or lists) as much as possible, then eliminating operator precedence could be a good idea.

   It is worth noting that an expression with operators can easily be ambiguous if the programmer isn’t familiar with the precedence, but that this isn’t possible in the same way when users are forced to add parentheses. However requiring parentheses can of course be a burden, e.g. by requiring more typing, and because users cannot rely on their knowledge of mathematics to get an intuitive understanding of how expressions work when learning the language. Extensive use of parentheses could also have a negative impact on readability.

5. Should C's assignment operations such as += be included in other languages? Why or why not?

The need to write an assignment that is based on the original value comes up often in imperative languages, e.g. as an index variable when iterating over data structures.

Writing:

```
A = A + B;
```

Is simply more verbose than:

```
A += B;
```

On the other hand, with a new operator not known from mathematics it can be argued that the operation becomes harder to read as users might not know what '+=' means? And as such, that the inclusion of assignment operations trade writability for readability.
This example is of course relatively mild, but as expressivity goes to extremes, readability goes down, so if the goal was to make an extremely readable language (perhaps a beginner's first imperative language?) compounded assignment operators like those featured in C would be harder to defend. For an example of a programming language with very concise code, arguably at the cost of readability, see APL: https://en.wikipedia.org/wiki/APL_(programming_language)

6. Should C's single-operand assignment form, such as ++, be included in other languages? Why or why not?

See Exercise 5 above, as the arguments for and against adding the ++ operator are quite similar to the arguments for and against adding the += operator.

7. Sebesta chapter 8, p 385-386, exercises 5 and 9

5. What are the arguments, pros and cons, for Python's use of indentation to specify compound statements in control statements?

In Python:

```
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

The scopes of control statements are marked with levels of indentation which makes them more readable. However, the lack of explicit end markers also make it harder to understand what scope a statement is part of if many nested scopes are used, and simply changing the indentation of a line can cause either parsing error or change the meaning of the program in the case of the last line. The latter might be more serious, since this error can go unnoticed by the programmer if the compiler does not express an error.

9. What are the arguments both for and against the exclusive use of Boolean expressions in the control statements in Java (as opposed to also allowing arithmetic expressions, as in C++)?

It can be argued that allowing arithmetic expressions to exist in control statements (where 0 would evaluate to false and any other number to true) grants a greater degree of expressivity.

An argument for exclusively using boolean values is that it can increase readability and allows the compiler to catch certain type errors at compile time, e.g, x = 1 is not a boolean expression but an assignment(which returns a value is some languages). This means that ‘if (x = 1)’ can be detected as an error. Furthermore, it is very easy to emulate the behavior of arithmetic expressions as booleans by adding a comparison operator e.g, aritmeticExpr != 0.