CS320 Handout 01
Different Ways of Programming the Factorial Function

The factorial function fact() consumes a non-negative integer and returns a non-negative integer. It is defined by:

$$\mathsf{fact}(x) = \begin{cases} 1 & \text{if } x = 0, \\ 1 \times 2 \times 3 \times \cdots \times x & \text{if } x \geqslant 1, \end{cases}$$

or, more compactly, by:

$$\mathsf{fact}(x) = \begin{cases} 1 & \text{if } x = 0, \\ \mathsf{fact}(x-1) \times x & \text{if } x \geqslant 1. \end{cases}$$

We can program the factorial function in many different ways, using different programming languages and also using different features of the same programming language. Among good qualities of a program are: its *correctness* ("the program indeed computes the desired function"), its *efficiency* ("the program is not wasteful of time and memory resources"), and its *transparency* ("the program is easy to understand and document"). Beyond these qualities, there is typically no "best" way of programming the same function without considering the particular application where the program will be used.

We illustrate some of these issues by programming the factorial function with four well-known languages: C, Java, ML and Scheme.

1 A Procedural Language: C

Here is a straightforward implementation in C, using a `for`-loop:

```
int fact(int x)
{   int i, answer = 1;
    for (i = 2; i <= x; ++i)
        answer = answer * i;
    return answer;
}
```

Using a `while`-loop, we can also write:

```
int fact(int x)
{  int i = 1, answer = 1;
   while (i++ < x)
        answer = answer * i;
   return answer;
}
```

Using a `while`-loop again, if we do not need to preserve the value of the input `x` but building up the answer by multiplying values of decreasing size, we can write:

```
int fact(int x)
{  int answer = 1;
   while (x > 0)
       { answer = answer * x; x = x - 1; }
   return answer;
}
```

or, more compactly (if somewhat obscurely):

```
int fact(int x)
{  int answer = 1;
   while (x > 0)
        answer *= x--;
   return answer;
}
```

2 An Object-Oriented Language: Java

Object-orientation can be seen as a refinement of the procedural approach, and both are cases of the *imperative* approach to programming. An *object* is a package of data together with operations on this data, and therefore knows how to do things to itself.

Because Java is object-oriented, it makes it easier to write programs usign objects. This approach applied to the factorial function produces a rather verbose code, consisting of objects that hold an integer value and can return both that value and its factorial:

```
public class MyInt
{  private int value;
   public MyInt(int value)
   {  this.value = value;
   }
   public int getValue()
   {  return value;
   }
   public MyInt getFact()
   {  return new MyInt(fact(value));
   }
   private int fact(int x)
   {  int answer = 1;
      while (x > 1) answer *= x--;
      return answer;
   }
}
```

The object-oriented approach is best applied to "programming in the large". The factorial function is very simple and should not require more than a couple of lines of code. A comparison with other languages shows Java's many advantages when dealing with the development of a large software package and organizing the inter-dependence of its many parts.

3 A Typed Functional Language: ML

The simplest and most natural implementation of the factorial function is in ML:

```
fun fact (x : int) : int =
    if x <= 0 then 1 else fact(x - 1) * x;
```

But you can also leave out the types, if you wish:

```
fun fact (x) =
    if x <= 0 then 1 else fact(x - 1) * x;
```

More precisely, the types are *implicit* in the preceding implementation: You do not see them, but the ML type-checker has to *infer* the types before the code is further processed and executed.

What stands out in the functional approach is that there are no *commands* in the code: There are only *expressions* and you only need to worry about their *values*. Another way of saying the same, there are no *states* and no *side-effects* in functional programming (at least in its pure form). You may feel uncomfortable with a programming language that does not have "commands" and give you the ability to allocate and de-allocate storage space. But you will see there is no loss of computational power and, very often, the more direct and transparent implementation of a computational task is to use a functional approach.

4 An Untyped Functional Language: Scheme

Much in the style of the implicitly-typed ML code, we can write in Scheme:

```
(define (fact x)
    (if (<= x 0) 1 (* (fact (- x 1)) x)))
```

The Scheme implementation is very succint, but you may be uncomfortable with several things, including:

1. Expressions are written in *prefix* rather than *infix* notation. For example,

`(<= x 0)`	rather than	`(x <= 0)`
`(- x 1)`	rather than	`(x - 1)`
`(* (fact (- x 1)) x)`	rather than	`(fact(x - 1) * x)`

   Granted, this notation may appear strange at first, but it also has several advantages. The infix notation is convenient in the case of binary operators, such as `<=` and `*`, but it is not clear how to extend it (in a natural way) to ternary operations. A clear advantage of the prefix notation is that operators, whether pre-defined or user-defined and regardless of how many arguments they take, all follow the same pattern:

   (⟨operator⟩ ⟨argument-1⟩ ⟨argument-2⟩ ⋯ ⟨argument-n⟩)

2. All expressions and subexpressions have to be each enclosed in a pair of matching parentheses. Moreover the application of `fact` to `x` is written `(fact x)` rather than `fact(x)`. Would it not be simpler to adjust the syntax of Scheme, omitting parentheses wherever we can and changing the "`if`" to "`if-then-else`", so that we can write the following declaration instead?

   ```
   define fact(x)
        if (<= x 0) then 1 else (* fact(x-1) x)
   ```

   Or even the following declaration?

   ```
   define fact(x)
        if x <= 0 then 1 else fact(x-1) * x
   ```

   Perhaps. But this would work only if we could also use a delimiter (e.g., "=" in ML) to separate the declaration heading "`fact(x)`" from its body "`if x <= 0 then 1 else fact(x-1) * x`". However, in keeping with Scheme's minimalist approach to syntax, no such delimiter is introduced, and the only markers used to delimit (valid) program phrases are pairs of matching parentheses.