

Syntax of Common Lisp Function Definitions and Calls

For any integer $k \geq 0$, a new Common Lisp function that takes k arguments can be defined as follows:

```
(defun <func name> (<param>1 ... <param>k)  
  <body-expr>)
```

- <func name> is a symbol (e.g., G or FACTORIAL) that will be the **name** of the new function.
- <param>₁, ..., <param>_k are k different symbols that will be the k **formal parameters** of the new function.
- <body-expr> is a Lisp expression.

A call of the function can be written as follows:

```
(<func name> <arg>1 ... <arg>k)
```

Here <arg>₁, ..., <arg>_k are Lisp expressions:

- This call is evaluated by **evaluating** <arg>₁, ..., <arg>_k, **then** evaluating <body-expr> in an environment in which the value of <param>_i ($1 \leq i \leq k$) is the value of <arg>_i, **and then** returning the value produced by that evaluation.

Simple Examples of Functional Programming

Reminder re the Java (and C++) `? :` Ternary Operator:

- If the value of *boolean-expr* is **true**, the value of *boolean-expr* `? expr1 : expr2` is the value of *expr₁*.
(In this case *expr₂* is *not* evaluated.)
- If the value of *boolean-expr* is **false**, the value of *boolean-expr* `? expr1 : expr2` is the value of *expr₂*.
(In this case *expr₁* is *not* evaluated.)

Example 3 The value of `(3 < 4) ? 5+1 : 7/0` is: **6**

Example 4 The value of `(3 > 4) ? 5/0 : 7+2` is: **9**

In functional programming, each function we write *just returns the value of a single expression* (which may be fairly complicated with a number of cases) without changing values stored in variables and data structures.

In Java, the body of such a function can often be written as follows:

```
{  
    return a single expression ;  
}
```

However, it is also possible for the function to declare one or more local variables, each of which is used to store the value of a subexpression, *provided that the function never changes the value stored in any of those variables and any data structure it may refer to.*

Such local variables can be used to *improve the readability of code*, or to *improve its efficiency by avoiding repeated evaluation of computationally expensive expressions.*

For example, unless the argument `i` is very small,

```
double f (int i)
{
    return (i == 0) ? 1.0 : f(i-1) + Math.sqrt(f(i-1));
}
```

is much slower than:

```
double g (int i)
{
    double y; // used to avoid repeated evaluation of g(i-1)
    return (i == 0) ? 1.0 : (y = g(i-1)) + Math.sqrt(y);
}
```

As `y` is used only in the `: ...` part of the `... ? ... : ...` expression, it'd be preferable to declare `y` there (rather than outside the `... ? ... : ...` expression). While Java *doesn't* allow that, we could write:

```
double g (int i)
{
    if (i == 0) return 1.0;
    else { double y = g(i-1); return y + Math.sqrt(y); }
}
```

But this `g`'s body is not in the above-mentioned form!

In functional programming, each function we write *just returns the value of a single expression*.

In Java, the body of such a function can often be written as follows:

```
{  
    return a single expression ;  
}
```

Here is a very simple Java function of this kind:

```
static float f (int n, float x)  
{  
    return n+x ;  
}
```

In functional programming, each function we write *just returns the value of a single expression*.

Here is another Java function of this kind:

```
// factorial(n)  $\Rightarrow$   $n! = 1 * 2 * \dots * (n-1) * n$  if  $1 \leq n \leq 20$ 
static long factorial (int n)
{
    return (n == 1) ? 1 : factorial(n-1) * n ;
}
```

Why factorial Works (noting that if $1 \leq n \leq 20$ and $n \neq 1$, then $1 \leq n-1 < n \leq 20$)

- When $1 \leq n \leq 20$ and $n \neq 1$, **factorial(n) returns the right result if factorial(n-1) returns the right result:**
If factorial(n-1) returns $1 * 2 * \dots * (n-1)$,
then factorial(n) returns $1 * 2 * \dots * (n-1) * n$.
- When $n = 1$, **factorial(n) returns the right result, 1**,
since evaluating $(n==1) ? 1 : \text{factorial}(n-1) * n$ when $n==1$
does not cause factorial(n-1) * n to be evaluated!

A Common Lisp Version of the Factorial Function

We've been considering following Java function:

```
// factorial(n) returns 1 * 2 * ... * (n-1) * n if 1 ≤ n ≤ 20
static long factorial (int n)
{ return (n == 1) ? 1 : factorial(n-1) * n; }
```

We now write a **Common Lisp** analog of this function.

To do this, we use the following facts:

- In Lisp, the **=** function can be used to test whether two numbers are equal.
- The Lisp analog of $c ? e_1 : e_2$ is **(if c e₁ e₂)**. **Examples:**
(if (= 2 3) 4 5) => 5 **(if (= 2 (+ 1 1)) 4 5) => 4**
Notation: **expr₁ => expr₂** means the *value* of **expr₁** is **expr₂**.

We also note that:

- The Lisp analog of the Java expression **factorial(n-1) * n** is:
(* (factorial (- n 1)) n)

Here is a Common Lisp version of the above function:

```
(defun factorial (n)
  (if (= n 1) 1 (* (factorial (- n 1)) n) ))
```


A Common Lisp Version of the Factorial Function

We've been considering following Java function:


```
// factorial(n) returns 1 * 2 * ... * (n-1) * n if 1 ≤ n ≤ 20
static long factorial (int n)
{ return (n == 1) ? 1 : factorial(n-1) * n; }
```

Here is a Common Lisp version of the above function:

```
(defun factorial (n)
  (if (= n 1) 1 (* (factorial (- n 1)) n) ))
```

As the (if ...) expression in this definition is quite long, it may be better to split it into 3 lines:

```
(defun factorial (n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```



Note: Do not put these last two closing parentheses on separate lines! That would *waste screen space* and also serve no good purpose because Lisp programmers read and write code in editors that match parentheses for them!

Here's a 3rd Java function that does nothing but return a value:

```
// returns  $n^k$  if  $1 \leq k$  and  $-2^{63} \leq n^k < 2^{63}$ 
static long pwr(long n, int k)
{
    return k == 1
        ? n
        : (k & 1) == 0           // true if k is even
            ? pwr(n*n, k/2)      // returned if k is even
            : pwr(n*n, k/2) * n; // returned if k is odd
} // Here / does integer division:  $k/2$  means  $\left\lfloor \frac{k}{2} \right\rfloor$  on this slide!
```

Why pwr Works (noting that if $1 \leq k$ and $k \neq 1$, then $1 \leq k/2 < k$)

When $1 \leq k$ and $-2^{63} \leq n^k < 2^{63}$, and $k \neq 1$,

$\text{pwr}(n, k) \Rightarrow$ the right value, n^k ,

if the recursive call $\text{pwr}(n*n, k/2) \Rightarrow$ the right value, $(n*n)^{k/2}$, because:

- If k is even, $(n*n)^{k/2} = n^k$.
- If k is odd, $(n*n)^{k/2} = (n*n)^{(k-1)/2} = n^{k-1}$.

When $-2^{63} \leq n < 2^{63}$ and $k = 1$, $\text{pwr}(n, k)$ returns the right value, n .

Like many functions in functional programming, `factorial` and `pwr` use:

- **conditional expressions** (`c ? e1 : e2` expressions)
- **recursion**

Functional programming also makes use of *functions that take functions as arguments*:

As an illustration of this, consider a function with header `static long sigma(Function<Integer,Long> g, int m, int n)` that returns the *sum of the results of applying the function given by its parameter g to each integer i, $m \leq i \leq n$.*

Examples Suppose `MyClass` is the class that contains the above functions `factorial` and `pwr`. Then:

```
sigma(MyClass::factorial, 3, 7)
  returns  3! + 4! + 5! + 6! + 7!      = 5910.
sigma(i->MyClass.pwr(i,5), 3, 7)
  returns 35 + 45 + 55 + 65 + 75 = 28975.
```

Here `i->MyClass.pwr(i,5)` is a "lambda expression": It denotes an unnamed function that maps an integer `i` to `i5`.

As another example, we now use the above function `sigma` to write a function

`static long sum_powers(int m, int n, int k)`
that returns $m^k + (m+1)^k + \dots + n^k$.

Thus when $m = 2$, $n = 5$, and $k = 4$ we have that:

$$\text{sum_powers}(2, 5, 4) \Rightarrow 2^4 + 3^4 + 4^4 + 5^4 = 16 + 81 + 256 + 625 = 978$$

This function can be written as follows:

```
static long sum_powers(int m, int n, int k)
{ return sigma(i -> MyClass.pwr(i,k), m, n); }
```

The function `sigma` we have been using can be written in a functional style, as follows:

```
static long sigma (Function<Integer,Long> g, int m, int n)
{ return (m > n) ? 0 : g.apply(m) + sigma(g, m+1, n); }
```

Here `g.apply(m)` calls the Function given by the value of parameter `g`, passing `m`'s value as its argument.

(Java does not allow this call to be written as `g(m)`!)

Functional programming can also use *functions that return functions as their results*.

Any function that takes a function as argument or returns a function as its result is called a higher order function.

Example of a Function That Returns a Function as Its Result

In math, we can *compose* functions $f : A \rightarrow B$ and $g : B \rightarrow C$ to give a function $g \circ f : A \rightarrow C$ such that $(g \circ f)(a) = g(f(a))$.

Thus if $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ and $g : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ are defined by $f(n) = n!$ and $g(n) = n + 5$, then $(g \circ f)(n) = n! + 5$.

Here is an analogous Java function:

```
static <A,B,C> Function<A,C> compose(Function<B,C> g,  
                                     Function<A,B> f)  
{ return n -> g.apply(f.apply(n)); }
```

`compose(n -> n+5, MyClass::factorial)` returns a function
that maps n to $n! + 5$.

\therefore `sigma(compose(n -> n+5, MyClass::factorial), 3, 6)`
returns $(3!+5) + (4!+5) + (5!+5) + (6!+5) = 890$.