Functional recursion

"Recursion" is a programming technique whereby the algorithm to solve a problem is decomposed into a *base case* where the result is simple, and a *general case* where the result is a computation that reuses the computation for a simpler case.

The technique is closely related to the *principle of induction* in mathematics. Actually, the little-known but very effective trick to learn to use recursion is to first ignore all aspects related to programming and concentrate on the mathematical notion of induction as a first step.

Contents

Inductive definitions in maths

An *inductive definition* in maths is the definition of some object or concept $A(n)$, depending on a non-negative integer parameter n, according to the following scheme: a) the value of a finite number of base cases, at least one (eg. $A(0)$) is given; and b) a rule is given for obtaining the value of $A(n)$ from n and fixed number of values in $A(0)...A(n-1)$.

Some typical inductive definitions:

Function	Base case(s)	General case
$n!$	$0! = 1$	$n! = n \times (n-1)!$
x^n	$x^0 = 1$	$x^n = x \times x^{n-1}$
Fibonacci	$F(0) = F(1) = 1$	$F(n) = F(n-1) + F(n-2)$

For the purpose of this course, you need to learn how to write all inductive definitions in maths according to the same fixed format:

- on the first line, the function name, the domain of its parameter(s) and its image set;
- on the second and following lines, the base cases, expressed using the function name;
- on the last line, the general case, expressed using the function name.

For example:

$$\begin{cases} \text{fact} : \mathbb{N} & \mapsto & \mathbb{N}^* \\ \text{fact}(0) & = & 1 \\ \text{fact}(n) & = & n \times \text{fact}(n-1) \end{cases}$$

(Now how we use "fact(n)" instead of "n!" here.)

$$\begin{cases} \text{pow} : \mathbb{R} \times \mathbb{N} & \mapsto & \mathbb{R} \\ \text{pow}(x, 0) & = & 1 \\ \text{pow}(x, n) & = & x \times \text{pow}(x, n-1) \end{cases}$$

(Note how we use "pow(x,n)" instead of "x^n" here.)

$$\begin{cases} \text{fib} : \mathbb{N} & \mapsto & \mathbb{N}^* \\ \text{fib}(0) & = & 1 \\ \text{fib}(1) & = & 1 \\ \text{fib}(n) & = & \text{fib}(n-1) + \text{fib}(n-2) \end{cases}$$

Finding your own inductive definition

The idea to use inductive definition should come to mind automatically as soon as you are faced with a description of what a function does based on some examples using an ellipsis "...".

Consider for example the following description:

> "Implement a function `sine(x,n)` which approximates the trigonometric function of the same name using its Taylor series with n terms, that is:
>
> ```
> sine(x,n) = x - x^3/3!  + x^5/5!  - x^7/7!  + ...
> ```
>
> with n terms, with n at least 1."

This description uses the ellipsis "..." so it invites an inductive definition.
Once you have this idea, you can achive an inductive definition as follows:

1. recognize in the description example which are the results for the *most simple form(s)* of the function, and write this down. This becomes the base case.
2. *reformulate the function at a step n using the result at step n-1, n and/or the other parameters.* Then write this down. This becomes the general case.
3. Write down the resulting definition using the canonical format described above.

In the example above, the most general form is when n = 1:

$\text{sine}(x, 1) = x$

Then at a given term n:

$$\text{sine}(x,n) = \text{sine}(x,n-1) + (-1)^n \frac{x^{(2n-1)}}{(2n-1)!}$$

So we can then write using the canonical format:

$$\begin{cases} \text{sine} : \mathbb{R} \times \mathbb{N} & \mapsto & \mathbb{R} \\ \quad \text{sine}(x,1) & = & x \\ \quad \text{sine}(x,n) & = & \text{sine}(x,n-1) + (-1)^n \frac{x^{(2n-1)}}{(2n-1)!} \end{cases}$$

Likewise, consider the following description:

"Implement a function $p(n)$ which for any $n \geq 1$ is equal to the following:

```
p(1) = 1
p(2) = 1 * 2 * 1
p(3) = 1 * 2 * 1 * 3 * 1 * 2 * 1
p(4) = 1 * 2 * 1 * 3 * 1 * 2 * 1 * 4 * 1 * 2 * 1 * 3 * 1 * 2 * 1
...
```

As a first step we need to recognize the most simple case, this is given for p(1).

Then we need to recognize how p(n) is expressed from p(n-1). For this we can look how p(2) is made from p(1):

```
p(2) = p(1) * 2 * p(1)
```

Then how p(3) is made from p(2):

```
p(3) = p(2) * 3 * p(2)
```

by trying out the first few terms, we recognize the pattern and we can generalize:

```
p(n) = p(n-1) * n * p(n-1)
```

From this we can express the full mathematical definition:

$$\begin{cases} p : \mathbb{N}^* & \mapsto & \mathbb{N}^* \\ \quad p(1) & = & 1 \\ \quad p(n) & = & p(n-1) \times n \times p(n-1) \end{cases}$$

From mathematical definition to recursive function

Once you have obtained a mathematical inductive definition, it is possible to transcribe the definition directly into a programming language, to obtain a *recursive function*. To do this, proceed as follows:

1. use the name of the function in the first line of the mathematical definition.
2. define the argument and return types using the domain and image sets of the mathematical definition.
3. inside the function body, use a `if` construct to detect the base cases. For each base case, use `return` to set the corresponding value.

4. in the `else` branch that does not correspond to the base case, write the general case. Do not hesitate to transcribe the mathematical definition without changes. It is OK that the function name appears in your transcription.

For example, consider the mathematical definition earlier:

$$\begin{cases} \text{fact} : \mathbb{N} & \mapsto & \mathbb{N}^* \\ \text{fact}(0) & = & 1 \\ \text{fact}(n) & = & n \times \text{fact}(n-1) \end{cases}$$

We take the name of the function, its domain and image and we transcribe:

```
int fact(int n) {
```

Then we use a `if` construct for the base case:

```
if (n == 0)
   return 1;
```

Then we finish with the general case:

```
else
   return n * fact(n-1);
}
```

This gives us the following recursive function:

```
int fact(int n)
{
   if (n == 0)
      return 1;
   else
      return n * fact(n-1);
}
```

Likewise for the following definition:

$$\begin{cases} p : \mathbb{N}^* & \mapsto & \mathbb{N}^* \\ p(1) & = & 1 \\ p(n) & = & p(n-1) \times n \times p(n-1) \end{cases}$$

The resulting code is:

```
int p(int n)
{
   if (n == 1)
      return 1;
   else
      return p(n-1) * n * p(n-1);
}
```

Likewise, for the following:

$$\begin{cases} \text{sine} : \mathbb{R} \times \mathbb{N} & \mapsto & \mathbb{R} \\ \text{sine}(x, 1) & = & x \\ \text{sine}(x, n) & = & \text{sine}(x, n-1) + (-1)^n \frac{x^{(2n-1)}}{(2n-1)!} \end{cases}$$

The resulting code is:

```
double sine(double x, int n)
{
   if (n == 1)
      return x;
   else
      return sine(x, n-1) + sign(n) * pow(x, 2*n-1) / fact(2*n-1);
}

// Here we need some helper functions because
// Java does not have a native operator for
// exponentiation:

int sign(n)
{
   if (n % 2 == 0)
      return 1;
   else
      return -1;
}

// Notice how pow() is itself a recursive function.
double pow(double x, int n)
{
    if (n == 0)
       return 1;
    else
       return x * pow(n-1);
}
```

Important concepts

- *inductive definition* in maths;
- standard format (presentation) for inductive definitions;
- *base case* and *general case*;
- process to recognize an inductive definition;
- process to translate an inductive definition in math into a recursive function in Java;

Further reading

- Think Java, sections 4.8 & 4.9 (pp. 44-46)
- Introduction to Programming, section 9.1 (pp. 423-435)
- Absolute Java, chapter 11 (pp. 648-687)

Copyright and licensing