

## Named function definitions

Like any other expressions,  $\lambda$ -expressions can be named. Naming a  $\lambda$ -expression allows it to be used more than once.

For example, by defining

```
> double = \x -> x + x
```

with a name, we can apply it as often as we like:

```
Main> double 3+5 + double (2^4)
43
```

The standard Haskell function-definition form is syntactic sugar for named  $\lambda$ -expressions.

An example:

```
> double x = x + x
```

The syntax:

- the parameters are placed between the function's name and the '='
- the parameters' scope is the definition's right-hand side (including any attached **where** clause)
- the  $\lambda$ -expression's ' $\backslash$ ' and '->' are omitted

Applications involving functions defined this way are evaluated by parameter-argument substitution, just like  $\lambda$ -expressions. For example,

```
double (3*3)
--> let x = 3*3 in x + x
--> let x = 9 in x + x
--> 9 + 9
--> 18
```

## Recursive function definitions

Named functions and conditional expressions enable functions to be defined recursively.

The classic recursive-function example:

```
> fac n = if n==0 then 1 else n * fac(n-1)
```

The usual calculation method works:

```
fac 3
--> if 3==0 then 1 else 3*fac(3-1)
--> if False then 1 else 3*fac(3-1)
--> 3 * fac (3-1)
--> 3 * let n = 3-1 in if n==0 then 1 else n*fac(n-1)
--> 3 * let n = 2 in if n==0 then 1 else n*fac(n-1)
--> 3 * if 2==0 then 1 else 2*fac(2-1)
--> 3 * if False then 1 else 2*fac(2-1)
--> 3 * (2 * fac (2-1))
--> 3 * (2 * let n = 2-1 in if n==0 then 1 else n*fac(n-1))
--> 3 * (2 * let n = 1 in if n==0 then 1 else n*fac(n-1))
--> 3 * (2 * if 1==0 then 1 else 1*fac(1-1))
--> 3 * (2 * if False then 1 else 1*fac(1-1))
--> 3 * (2 * (1 * fac (1-1)))
--> 3 * (2 * (1 * let n = 1-1 in if n==0 then 1 else n*fac(n-1)))
--> 3 * (2 * (1 * let n = 0 in if n==0 then 1 else n*fac(n-1)))
--> 3 * (2 * (1 * if 0==0 then 1 else 0*fac(0-1)))
--> 3 * (2 * (1 * if True then 1 else 0*fac(0-1)))
--> 3 * (2 * (1 * 1))
--> 3 * (2 * 1)
--> 3 * 2
--> 6
```

Of course, not all recursions terminate.

Example:

```
let h x = 1 + h x in h 2
--> 1 + h 2
--> 1 + (1 + h 2)
--> 1 + (1 + (1 + h 2))
--> ...
"="  $\perp$  (this equality is not computable)
```

Function `fac` is an example of **linear recursion**

A recursive function  $f$  is *linear*-recursive if each application of  $f$  produces **at most one** new application of  $f$ .

An example of non-linear recursion is

```
> fib n = if n==0 || n==1 then 1
>         else fib (n-1) + fib (n-2)
fib 4
--> if 4==0 || 4==1 then 1 else fib (4-1) + fib (4-2)
--> ...
--> fib 3 + fib 2
--> ...
--> (fib 2 + fib 1) + fib 2
--> ...
--> ((fib 1 + fib 0) + fib 1) + (fib 1 + fib 0)
--> ...
--> 5
```

## Tail recursion

A linear-recursive function  $f$  is *tail*-recursive if what it returns **either**

1. contains *no* applications of  $f$
- or**
2. consists *entirely* of a *single* application of  $f$ .

See <http://haskell.org/hawiki/TailRecursive> .

Example:

```
> fact n a
> = if n==0
>     then a           -- contains no applications of fact
>     else fact (n-1) (n*a) -- consists of nothing but
-- a single application of fact
```

A calculation:

```
fact 4 1
--> fact 3 (4*1)           winding...
--> fact 2 (3*(4*1))       winding...
--> fact 1 (2*(3*(4*1)))   winding...
--> fact 0 (1*(2*(3*(4*1)))) base case...
--> 1*(2*(3*(4*1)))
--> 1*(2*(3*4))
--> 1*(2*12)
--> 1*24
--> 24
```

**Guards** (syntactic sugar)

An alternative to a function definition whose body consists of a conditional expression is one that uses *guarded equations*.

Another formulation of factorial, as a guarded equation:

```
> fac1 n
>   | n==0    = 1
>   | otherwise = n * fac1 (n-1)
```

When an application of a guarded function is evaluated, its guards are first evaluated (top-down); the first true guard determines the result.

Although you might guess that `otherwise` is a reserved word, it's merely a synonym for `True`.

If all guards are false, the result is an error message (so `fac2` is a better definition than `fac1`):

```
> fac2 n
>   | n==0 = 1
>   | n > 0 = n * fac2 (n-1)
```

Example:

```
Main> fac2 5
120
Main> fac2 0
1
Main> fac2 (-5)
Program error: {fac2 (-5)}
```

Why are the parentheses needed around `-5`?

**Pattern-matching function definitions**

Another form of function definition is composed of several equations. The equation to be used in a particular application is selected by comparing the argument with the equations' parameter patterns.

For example,

```
> daysIn "feb" = 28
> daysIn "sep" = 30
> daysIn "apr" = 30
> daysIn "jun" = 30
> daysIn "nov" = 30
> daysIn _     = 31
```

The patterns available are the same as in `case` expressions (slide 46).

A plain parameter name can be seen as a pattern that matches any value.

If a function application's argument matches *none* of the patterns in the function's definition, the application's value is  $\perp$ .

Patterns and guards can be combined:

```
> fac3 0      = 1
> fac3 n | n>0 = n * fac3 (n-1)
> fac3 _     = error "fac3 n is not defined for n<0"
```

When an expression

```
error s
```

is evaluated, execution halts and string `s` is printed on the output stream.

Patterns and guards can be combined in the same way in `case`-expressions. For example,

```
> fac4 n
> = case n of
>   0      -> 1
>   n | n > 0 -> n * fac4 (n-1)
>   _     -> error "fac4 n undefined for n<0"
```

**Lazy evaluation and Strictness**

Haskell's lazy-evaluation policy makes Haskell functions *non-strict* by default.

A *strict* function is one that always maps  $\perp$  to  $\perp$ . That is, if its argument is undefined, its result is undefined.

In most programming languages, all functions are strict, except for a few special built-in functions such as the C operators (`&&`), (`||`), and (`?:`).

Haskell's nonstrictness allows functions to map  $\perp$  to well-defined values. For example,

```
(\x -> 3) (1/0)
~> 3
```

Hence, for example, the Haskell operators (`&&`) and (`||`) can be defined as nonstrict functions using ordinary pattern-matching definitions:

```
> False && _ = False
> True  && x = x
> True  || _ = True
> False || x = x
```

Pattern-matching with a literal parameter (such as the above definition's `False` and `True`) **forces** evaluation of the corresponding argument.

So evaluation of

```
a && b
```

always requires the evaluation of `a`, but if `a == False`, then `b` is not evaluated.