# Function Specifications and Patterns

Johannes Borgström
johannes.borgstrom@it.uu.se

Program Design and Data Structures

Based on notes by Tjark Weber, Lars-Henrik Eriksson, Pierre Flener, Sven-Olof Nyström

# Today

- Function specifications
- Patterns

# Function Specifications

# Motivation: What Does a Function Do?

When you've written a (perhaps complicated) function, you want other programmers to be able to use this function without having to study its implementation. How can you tell them what your function does?

When someone wants you to implement a particular function, how can they tell you (without implementing it themselves) what the function should do?

We need a way to **specify** functions (i.e., to say precisely what a function does) that is independent of the function's implementation.

# Function Specifications

In this course, our function specifications will consist of

1. Function name and arguments
2. Function type — argument(s) and result
3. Description — what is the purpose of this function?
4. What the function assumes (precondition)
5. What the function computes (return value)
6. Side effects
7. Examples of function usage

Specifications may be given at different levels of detail.

Specifications may be given in natural language, or more formally.

# Function Specifications 1: Name

The name of a function should be a concise description of what it computes.

Example:

```
{- takeLast
   ...
 -}
```

When naming related functions, ensure that their names follow a pattern.
Use nouns or active verb forms (or adjectives for functions returning `Bool`.)

# Function Specifications 2: Description

Write a short description of the purpose of the function.

Example:

```
{- takeLast
   Get a suffix of a string.
   ...
 -}
```

Concise, intended to communicate to another programmer what the function is to be used for.

# Function Specifications 3: Examples of Usage

Show examples of what happens when the function is applied to specific arguments.

Example:

```
{- takeLast
   ...
   EXAMPLES: takeLast 5 "computer" == "puter"
             takeLast 0 "computer" == ""
 -}
```

Focus especially on arguments where the result is counterintuitive or difficult to understand. Good examples can be used as the basis for tests.

# Function Specifications 4: Argument Names

Give names to each of the parameters of the function.

Example:

```
{- takeLast n str
   Get a suffix of a string.
   ...
   EXAMPLES: takeLast 5 "computer" == "puter"
             takeLast 0 "computer" == ""
 -}
```

The names of the arguments do not have to be consistent between the specification and the code. These should be plain identifiers, not types or patterns.

# Function Specifications 5: Types of Arguments and Result

Example:

```
{- takeLast n str
   Get a suffix of a string.
   ...
   EXAMPLES: takeLast 5 "computer" == "puter"
             takeLast 0 "computer" == ""
 -}
takeLast :: Int -> String -> String
```

In Haskell, the function type is checked by the compiler.

# Function Specifications 6: Return Values

Example:

```
{- takeLast n str
   Get a suffix of a string.
   ...
   RETURNS: the last n characters of str
   EXAMPLES: takeLast 5 "computer" == "puter"
             takeLast 0 "computer" == ""
 -}
takeLast :: Int -> String -> String
```

What a function produces is called its **return value**.

# Return Values: What, Not How

The return value should state **what** the function returns, not **how** it computes that result.
The return value is **independent** of the concrete implementation (and should be written first).

For instance,

**WRONG:** n is subtracted from the length of str; that many characters are then dropped from the beginning of str and the remaining suffix is returned

(Note that this describes a particular algorithm to compute the last n characters of str.)

**CORRECT:** the last n characters of str

# Return Values: Considerations

- What, not how!
- Use clear and precise language or mathematical notation.
- Keep the description abstract: mention only what is relevant for callers, omit implementation details.
- Refer to arguments by name.
- If not all of the function's formal arguments are mentioned in the return value, it is almost certainly wrong. (Why?)

# Function Specifications 7: Side Effects

Functions can do other things than to just compute values: read from keyboard and files, write to terminal and files, ...

Example:

```
appendFile :: FilePath -> String -> IO ()
```

Evaluating `appendFile file str` appends the string `str` to the file `file`

```
{ - appendFile filePath str
   ...
    SIDE EFFECTS: appends str to the file found at filePath
-}
```

# Preconditions

Some functions do not return meaningful results for all possible input arguments.
For instance, `1 `div` 0` causes a run-time error.

The condition under which a function returns a meaningful result is called its **precondition**.

For instance, the precondition for `div` is that its second argument is not equal to `0`.

# Function Specifications 8: Preconditions

Example:

```
{- takeLast n str
   Get a suffix of a string.
   PRE: 0 <= n <= length str
   RETURNS: the last n characters of str
   EXAMPLES: takeLast 5 "computer" == "puter"
             takeLast 0 "computer" == ""
 -}
takeLast :: Int -> String -> String
```

The precondition does not need to say anything about argument types: these are already indicated in the type of the function.

If a function does return a meaningful result for all possible inputs, its precondition is simply `True`, and shall be omitted.

# Preconditions: Considerations

The precondition describes what a function assumes about its arguments.

If the precondition is met, the function must return a meaningful result (according to its return value), and not produce a run-time error.

If the precondition is violated, the function may or may not produce a meaningful result.

# Design by Contract

Pre- and return values can be seen as a contract between the function and the caller.

The caller promises to fulfil the precondition and the function promises to return a result satisfying the description of the return value.

In particular, if the caller breaks his part of the contract (i.e., if the precondition is violated), the function may behave in any way whatsoever.

# When Preconditions Are Violated . . .

On its first test flight in June 1996, the European Ariane 5 space rocket (worth nearly US$ 400 million) exploded 37 seconds after launch because of a malfunction in its control software.

The software was originally written for the Ariane 4 and could not cope with the higher speed of the Ariane 5 rocket.

The program itself was working correctly. The error was caused because the program's *precondition* was violated.

*Don't break the contract!*

# Patterns

# Motivation: Conditional Computation

Often, functions need to perform different computations,
depending on the values of their arguments.

We have already seen one way of doing this in Haskell: conditional expressions.
For instance,

```
lucky x = if x==7 then "LUCKY NUMBER!" else "Sorry, pal!"
```

When there are many different cases to consider,
we need many nested `if-then-else` expressions.
These can be hard to read.

# Function Declarations with Patterns

A case distinction that tests whether the function argument is equal to a constant can be written more elegantly:

```
lucky 7 = "LUCKY NUMBER!"
lucky x = "Sorry, pal!"
```

Here, `7` and `x` are called **patterns**.

# Evaluation of Patterns

```
lucky 7 = "LUCKY NUMBER!"
lucky x = "Sorry, pal!"
```

When `lucky` is called with an argument, Haskell evaluates only the expression of the *first* clause where the pattern **matches** the actual argument.

A constant matches only itself. An identifier matches any value.

```
> lucky 7
"LUCKY NUMBER!"
> lucky 42
"Sorry, pal!"
```

# Patterns: Order is Important

The order of equations (within a declaration that uses patterns) is important! Consider

```
lucky x = "Sorry, pal!"
lucky 7 = "LUCKY NUMBER!"
```

Now `lucky 7` $\longrightarrow$ `...`?!

# The _ Pattern

The underscore `_` can be used as a pattern when we do not care about the value of the argument. Like an identifier pattern, it matches any value. But the underscore creates no binding.

Example:

```
lucky 7 = "LUCKY NUMBER!"
lucky _ = "Sorry, pal!"
```

# Patterns and Non-Strict Evaluation

As usual, function arguments are evaluated only when (and if) they are needed to determine whether a pattern matches.

```
lucky 7 = "LUCKY NUMBER!"
lucky x = "Sorry, pal!"

> lucky (1 `div` 0)
"*** Exception: divide by zero

fortytwo _ = "42"

> fortytwo (1 `div` 0)
"42"
```

# (Non-)Exhaustive Matching

It is (possible, but) usually a bad idea to provide clauses only for some of the possible argument values.

```
isZero 0 = True

> isZero 0
True

> isZero 42
*** Exception: ...: Non-exhaustive patterns in function isZero
```

There will be a runtime error when the function is called with an argument that does not match any given pattern.

# "Catch-All" Equations

It is easy to avoid non-exhaustive matches. One can simply specify a final equation whose pattern matches any value.

Example:

```
isZero 0 = True
isZero _ = False
```

# Matching Tuples

It is possible to match tuples, as well as inductive data types (we will see later what those are).

For instance, functions `fst` and `snd` (which are provided by the Prelude) could be declared as follows:

```
fst (x, _) = x

snd (_, y) = y
```

# Matching Tuples (cont.)

Tuple patterns (also data type patterns) can contain other patterns, e.g., constants, identifiers, _.

For instance,

```
numberOfZeros (0, 0) = 2
numberOfZeros (0, _) = 1
numberOfZeros (_, 0) = 1
numberOfZeros (_, _)_      = 0
```

# Function Declarations with Patterns

In general:

```
name pattern1 = expression1
name pattern2 = expression2
...
name patternN = expressionN
```

Every **pattern** is a constant, identifier, underscore (_) or a "skeleton" for a datatype (e.g., tuples) where the skeleton components are patterns. Note that function applications are not patterns!

All patterns in a function declaration must have the same type (namely the argument type of the function).

All expressions on the right-hand side of a function declaration must have the same type (namely the return type of the function).

# Patterns in Value Declarations

We have seen value declarations where the left-hand side is an identifier.

```
> x = ("PKD", 2019)
> x
("PKD",2019)
```

In general, the left-hand side may be a pattern.

```
> (course, year) = ("PKD", 2019)
> course
"PKD"
> year
2019
```

# Refutable Patterns

The pattern in a value declaration must match the value of the right-hand side. Otherwise, an exception is thrown when (some part of) the pattern is evaluated.

```
> let (course, 2015) = ("PKD", 2019) -- no error yet
> course
*** Exception: <interactive>:...: Irrefutable pattern failed for pattern (course
    , 2015)
```

A pattern that may fail to match is called **refutable**. A pattern that must never fail to match is called **irrefutable**.

# Patterns: Linearity

Patterns in Haskell must be **linear**: each identifier can occur at most once.

Linear:

```
allEqual (x, y, z) = x==y && y==z
```

Not linear:

```
allEqual (x, x, x) = True
```

```
Conflicting definitions for `x'
Bound at: ...
In an equation for `allEqual'
```

# Guarded Patterns

The patterns that we have seen so far are not very flexible: it is only possible to compare (some part of) an argument to some constant.

For instance, consider the sign function.

Without patterns:

```
sign x = if x<0 then -1 else if x==0 then 0 else 1
```

With patterns:

```
sign 0 = 0
sign x = if x<0 then -1 else 1
```

# Guarded Patterns (cont.)

With **guarded patterns**, we can get rid of the remaining `if-then-else`:

```
sign 0 = 0
sign x | x < 0 = -1
       | otherwise = 1
```

Thus, guarded patterns are of the form `pattern | expr`. Here, `expr` must have type `Bool` and will usually refer to names from `pattern`.

Patterns are tested in order. When a pattern matches, its guard is evaluated. If the guard evaluates to `True`, the equation's right-hand side is evaluated. If the guard evaluates to `False`, the next equation is tried.

# Case Expressions

We've seen how to use patterns to distinguish different cases when we define a function. Haskell's `case` expression allows us to distinguish different cases *without* defining a function.

```
case 42 `mod` 2 of
  0 -> "even"
  1 -> "odd"
```

# Case Expressions (cont.)

In general:

```
case expr of
  pat1 -> expr1
  pat2 -> expr2
  ...
  patN -> exprN
```

- `case ... of ...` is an expression.
- `expr`, `pat1`, ..., `patN` must be of the same type.
- `expr1`, ..., `exprN` must be of the same type.
- Patterns are tested in order. If `pati` matches `expr`, then *only* `expri` is evaluated.

# Case Expressions: (Non-)Exhaustive Patterns

Our earlier remarks about redundant and non-exhaustive patterns in function declarations equally apply to `case` expressions.

```
> case 42 `mod` 2 of _ -> "odd" ; 0 -> "even"

<interactive>:...: Warning:
    Pattern match(es) are overlapped
    In a case alternative: 0 -> ...
"odd"


> case 42 `mod` 2 of 1 -> "odd"
"*** Exception: <interactive>:...: Non-exhaustive patterns in case
```

# Pattern Matching and Shadowing

When a pattern matches a value (and only then), identifiers in the pattern are bound to corresponding parts of the matching value.

These bindings can cause shadowing.

Example:

```
multiply x y =
  case x * y of
    1 -> "one"
    2 -> "two"
    x -> show x
```

What is the value of `multiply 3 4`?

# Syntactic Sugar

Often, there is more than one way to say something.

**Syntactic sugar** refers to syntax that allows things to be expressed more clearly or more concisely, but that could (in principle) be expressed with other syntax.

Note that there is a tradeoff: with syntactic sugar, programs become more readable, but the programming language becomes more complicated.

# Syntactic Sugar in Haskell

`if-then-else` is redundant:

```
if condition
  then true_value
  else false_value
```

```
case condition of
  True -> true_value
  False -> false_value
```

# Syntactic Sugar in Haskell (cont.)

Patterns in function declarations and case matching are interchangeable:

```
f pat1 = expression1
f pat2 = expression2
...
f patN = expressionN
```

```
f x =
  case x of
    pat1 -> expression1
    pat2 -> expression2
    ...
    patN -> expressionN
```