# Program Design and Data Structures, 20.0 c

Course code: 1DL201, Report code: DL201, 67%, DAG, NML,
week: 44 - 02 Semester: Autumn 2017
week: 03 - 11 Semester: Spring 2018

## CODING CONVENTION

This information is not available in English. Now showing the Swedish version.

# Coding Convention

Good software development companies have coding conventions to ensure that code is written in a uniform way, to enhance readability for whoever will maintain the code. The requirements in this coding convention have been devised to be *tools* to guide and facilitate your thinking *before and while* programming, but *not* additional obstacles after the already difficult task of programming.

They also ease the communication of your programs to others, such as the instructors and assistants who grade them.

Shedding these good habits after this course would be doing yourself a great disservice. A competent programmer can of course do the specification and verification (or verification outline, as in this course) in her brain only, but that will be of no help if she herself reconsiders the program a few weeks later, or if somebody else has to use or modify or verify her code. No engineer takes on a well-defined task without a specification.

The real value of following coding conventions like this one becomes apparent when you work in a team of developers, and you have to read (and understand, use and modify) code written by someone else.

## FUNCTION SPECIFICATIONS

Every function, whether declared globally or locally, whether named or anonymous, whether curried or not, whether commissioned in the exercise or invented by you as a helper function, could be commented with its specification, as follows:

```
{- functionIdentifier arguments
   A brief human-readable description of the purpose of the function.
   PRE:  ... pre-condition on the arguments, if any ...
   RETURNS: ... description of the result, in terms of the arguments ...
   SIDE EFFECTS: ... side effects, if any, including exceptions ...
   EXAMPLES: ... especially if useful to highlight delicate issues; also consider including counter-examples ...
-}
functionIdentifier :: argumentType -> resultType
```

Top level functions that are exported from a module **must** have the complete comment. Helper/auxiliary (whether defined at the top level or using let or where clauses) and nested functions that are of *significant complexity* **must** have the complete comment. Trivial functions can have an abbreviated comment, or even no comment when the function is completely clear.

The names of the arguments need *not* be consistent between the specification and the program. These should be plain variables, not types or patterns.

Under PRE, be careful *not* to require any accumulator to be equal to some constant. Indeed, this may be the case at the first call, but any recursive call will almost certainly be for a value different from that constant, and would thus violate such a pre-condition. If there is no pre-condition (i.e., true), this row may be omitted. Similarly, if there are no side effects, the row SIDE EFFECTS can be omitted.

Under PRE and RETURNS, there is *no* need to repeat the types of the argument and result, as they are already indicated in the type of the function.

Similarly, assume that the representation invariants of all data types (see below) hold -- this means representation invariants should not be restated in pre- and post-conditions.

Under RETURNS, it is *sufficient* to give the returned expression *e*, so that the actual post-condition *implicitly* is "the returned expression is equal to *e*". Under this convention, writing "this function returns *e*" is also unnecessarily long.

A function specification is almost always wrong unless *all* of the function's arguments appear in the post-condition or return value.

Specifications should be in English. Specifications are usually written independently of the programming language, to the extent possible. A suitable combination of natural language and rather standard mathematical notation is usually best.

Typically, one does not describe how the function behaves for invalid inputs, namely, those not satisfying the representation convention and pre-condition.

# IDENTIFIERS

*Every* function identifier shall be descriptive of the performed function. *Every* value identifier shall be descriptive of the provided value.

*Every* function, value and type identifier shall begin with a lowercase letter. *Every* constructor name shall begin with an uppercase letter.

If there are several words in an identifier, then glue them together, *without* using the underscore character ('_'), and start each new word with an uppercase letter. (This is known as *camel case*.)

Examples:

```
maxValue
endOfTheGame
data ThisIsADatatype = ThisIsAConstructor
```

# DATATYPE REPRESENTATION

*Every* datatype definition (cf. the data or type keywords) shall be commented with an explanation of the type's representation convention and invariant:

```
{- ... description of how the datatype represents data ...
  INVARIANT:  ... requirements on elements of the datatype that the code preserves at all times ...
-)
```

Data types can be supplied with valid and invalid examples to help illustrate how they represent data, and which instances of the data type are invalid. If there are no invalid instances (i.e., the invariant is "true"), this row may be omitted.

# INDENTATION

Layout and indentation of *function declarations* shall be as follows:

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

Alternatively, expressions can be written on a separate line:

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

Layout and indentation of *if-then-else* expressions shall be as follows:

```
if booleanExpression
  then expression1
  else expression2
```

If further *if-then-else* expressions follow after *else* the layout and indentation shall be as follows

```
if booleanExpression1
  then expression1
  else if booleanExpression2
         then expression2
         else ...
```

Layout and indentation of *case* expressions shall be as follows:

```
case expression of
  pattern1 -> expression1
  ...
  patternN -> expressionN
```

Layout and indentation of *let* expressions shall be as follows:

```
let
  declaration1
  ...
  declarationN
in
  expression
```

Simple let expression can omit the newline after the `let` and in keywords:

```
let declaration1
in expression
```

Layout and indentation of *where* clauses shall be as follows:

```
  declaration
   where
    declaration1
    ...
    declarationN
```

Simple *where* clauses can be formatted as follows:

```
   declaration
     where declaration1
```

The exact number of spaces is not important, but all indentation must be consistent, that is, each expression within "..." above must begin equally far from the left side (except where the indentation is changed according to the rules above).

Indentation of terms other than those described above shall be made in good faith. Please look at programs on the lecture slides for ideas.

# EXAMPLES

Functions should include examples of how they behave. These examples can also be used as test cases.

```
{- fac n
  PRE:  n >= 0
  RETURNS: n! (i.e., the factorial of n)
  EXAMPLES: fac 0 = 1
            fac 3 = 6
-}

fac :: Integer -> Integer

-- VARIANT: n
fac 0 = 1
fac n = n * fac (n-1)


{- select n xs
  PRE:  true
  RETURNS: the list of the elements in xs that are larger than n,
       in the same order as they occur in xs
  EXAMPLES: select 2 [1,3,2,4,3] = [3,4,3]
            select 3 [1,3,1]     = []
            select 0 []          = []
-}

select :: Integer -> [Integer] -> [Integer]

-- VARIANT: length xs
select _ [] = []
select n (x:xs) =
 let
   xs' = select n xs
 in
   if x>n then x:xs' else xs'
```

```
{- MyRational
   Rational numbers based on the Int type.  "Rat nom denom" denotes nom/denom.
   INVARIANT: denom <> 0 in Rat nom denom.
 -}
data MyRational = Rat Int Int
```

Function examples involving abstract data types as input or output of functions may be represented by describing the data structure in words. For instance, "a table containing elements ('a', 10) and ('b',11)."

Examples for functions involving complex input or output may be described in words where feasible. For example, if the type models a chess board, then a brief description of the board is likely to be more informative than a detailed representative of the board.

(This coding convention is based on earlier material by Lars-Henrik Eriksson and Pierre Flener, translated to Haskell by Dave Clarke.)