Functional Programming I, 5.0 c

Course code: 1DL330, Report code: 11006, 33%, DAG, NML, week: 35 - 43 Semester: Autumn 2017

CODING CONVENTION

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

Coding Convention

Good software development companies have coding conventions, hence so can programming courses and software development courses! 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 call 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, shall be commented with its specification, as follows:

```
(* function_identifier argument
   TYPE: argument_type(s) -> result_type
   PRE:  ... pre-condition on the arguments ...
   POST: ... post-condition on the result ...
   SIDE EFFECTS: ... if any, including exceptions ...
   EXAMPLES: ... especially if useful to highlight delicate issues; also consider including counter-examples ...
*)
```

where the function identifier is unnecessary for anonymous functions. The only exception to the necessity of specifications is for an anonymous function that is passed as an argument to another function: you need not specify it if it is very simple.

The names of the components of the argument need *not* be consistent between the specification and the program.

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.

Under PRE and POST, there is *no* need to repeat the types of the argument and result, as they are already indicated under TYPE.

Under POST, 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 side effects.

Specifications shall 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.

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 either (i) separate each word with an underscore ('_'), or (ii) glue them together, *without* using the underscore character, and start each new word with an uppercase letter. (This is known as *camel case*.) Be consistent.

Examples:

```
max_value
end_of_the_game
datatype this_is_a_datatype = This_is_a_constructor

maxValue
endOfTheGame
datatype thisIsADatatype = ThisIsAConstructor
```

RECURSIVE FUNCTIONS

Every recursive function shall be commented with the chosen numeric variant, as follows:

(* VARIANT: ... *)

Like the pre- and post-conditions, the variant *must* refer to the parameter names in the specification, but *not* to the possibly different parameter names in the program. The reason for this is that the variant reflects a decision made *before* the program was written, according to the design methodology. Also, the variant is *not* part of the specification, but rather part of the verification outline of why the program is believed correct. For a given specification, many programs can be written, and the recursive ones usually have different variants.

DATATYPE REPRESENTATION

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

```
(* REPRESENTATION CONVENTION: ... description of how the datatype represents data ...
   REPRESENTATION INVARIANT:  ... requirements on elements of the datatype that the code preserves at all times ...
 *)
```

INDENTATION

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

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

Alternatively, expressions can be written on a separate line:

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

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

```
if boolean_expression then
   expression1
else
   expression2
```

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

```
if boolean_expression1 then
   expression1
else if boolean_expression2 then
   expression2
...
else
   expressionN
```

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
end
```

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

```
(* fac n
   TYPE: int -> int
   PRE:  n  0
   POST: n! (i.e., the factorial of n)
   EXAMPLES: fac 0 = 1
             fac 3 = 6
*)
(* VARIANT: n *)
fun fac 0 = 1
  | fac n = n * fac (n-1)


(* select n xs
   TYPE: int -> int list -> int list
   PRE:  true
   POST: 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 []          = []
*)
(* VARIANT: the length of xs *)
fun select _ [] = []
  | select n (x::xs) =
    let
      val xs' = select n xs
    in
      if x>n then x::xs' else xs'
    end
```

(This coding convention is based on material by Lars-Henrik Eriksson, Pierre Flener and Tjark Weber.)