About coding style

15-212 Spring 2004

February 9, 2004

Contents

A few students have asked what are the style guidelines for a perfect homework. First of all, the code in the homework is graded based on three main components: *correctness*, *specification*, and *style*.

1 Correctness

Correctness refers to the correctness of the function that is implemented, i.e., if the function computes the value requested in the homework handout.

2 Specification

Specification refers to the presence and correctness of the specification in a comment before the function: the comment must contain the identifier being declared, its type, an explanation of the meaning of the value computed, a list of invariants, and a list of possible effects. It is very important for the specification to be present for all functions, global, auxiliary and local functions: you can refer to the code distributed on the course webpage for some examples. Very simple helper functions and lambda expressions may not have a specification: if the function is very simple its meaning can be understood without the help of a specification. Use your judgment to decide if a specification is needed for a helper function, when in doubt, provide the specification: you will reduce the risk of losing specification points.

3 Style

Style refers to the way you implemented the requested function: most functions can be implemented in many different ways, each one of them satisfies the correctness requirement, but some of them have a *better style*. In specific, since the main focus of the course is functional programming, a "functional" style is to be preferred. Moreover, in general an elegant solution should be preferred to a more efficient one, if the more efficient solution is much more complicated. However do not write a very elegant $O(2^n)$ solution instead of a bit less elegant $O(n)$ one! Sometimes we might ask to implement a function following particular requirements (e.g., request a recursive definition): if there exists a more efficient solution that does not adhere to such requirements, please do not use such solution as it will not be considered a correct solution to the problem, as it does not satisfy the explicit requirements. Often exercises are meant to test the ability to use a particular programming technique (like the use of recursive function definitions) and if the solution does not follow the given requirement, it won't be useful in establishing the ability to use such technique.

In terms of style, one of the purposes of homework #1 was to get familiar with the functional programming style. The best sources for learning good programming style are the example code presented at lectures and recitation as well as the code distributed on the course webpage. The following contains some example of bad style that you can use to understand what you should avoid, and what to do instead:

3.1 `if-then-else` Expressions

3.1.1 Boolean `if-then-else` Expressions

```
if a = b then
  true
else
  false
```

It is bad style to have an `if-then-else` expression whose type is `bool`. The previous expression is equivalent to `a = b`. Other examples include:

```
if a = b then
  false
else
  true
```

is equivalent to:

```
not (a = b)
```

```
if a = b then
  if a > 0 then
    true
  else
    false
else
  false
```

is equivalent to:

```
(a = b) andalso (a > 0)
```

```
if a = b then
  true
else
  if a > 0 then
    true
  else
    false
```

is equivalent to:

```
(a = b) orelse (a > 0)
```

3.1.2 `if-then-else` instead of Pattern Matching

```
fun natpred (n: int) : int =
  if n = 0 then
    0
  else
    n - 1
```

In this case, the `if` statement is used to discriminate over the value of an argument that is matched against a constant. This should be done using pattern matching as follows:

```
fun natpred (0: int) : int = 0
  | natpred (n: int) : int = n - 1
```

3.2 Functions

3.2.1 Multiple Evaluations

```
if f (x) > 0 then
  f (x)
else
  0
```

In this case function `f` is evaluated twice. Since *f* does not have side effects (you are not allowed to use the imperative features of SML), the function will return the same value each time. If is better style to do the following:

```
let
  val y = f (x)
in
  if y > 0 then
    y
  else
    0
end
```

or:

```
let
  fun positiveorzero (n:int): int =
    if n > 0 then
      n
    else
      0
in
  positiveorzero (f (x))
end
```

The second solution is to be preferred especially if the function `positiveorzero` is reused in other parts of the code (in which case it has to be scoped accordingly).

3.2.2 Redefinition of Library Functions

```
(*
   val max: int * int -> int

   max (x,y) returns the maximum of two numbers.

   Invariants: none
   Effects: none

*)

fun max (x: int, y: int): int =
    if x > y then x else y

(*
   val maxlist : int list -> int

   maxlist (l) computes the maximum value contained in the list

   Invariants: l is not empty and every element of l is positive.
   Effects: none
*)

fun maxlist ([]: int list): int = 0
  | maxlist (x::xs: int list): int =
    max (x, maxlist (xs))
```

There is another additional improvement that should be done to this code. The function `max` is actually equivalent to the library function `Int.max`. In general, you should not define your own function when an equivalent function is present in the library. Library functions are described in the SML base library documentation, moreover, some of the most commonly used functions are also summarized in the appendix of the textbook. Try to avoid "re-inventing the wheel", and make full use of the library functions. This is especially true in the case of library functions for lists, like `foldl`, `foldr`, and `map`, which can often be used to produce a very simple and elegant function that accomplishes the given task.

```
(*
   val maxlist : int list -> int

   maxlist (l) computes the maximum value contained in the list
   Invariants: l is not empty and every element of l is positive.
   Effects: none
*)

fun maxlist ([]: int list): int = 0
  | maxlist (x::xs: int list): int =
    Int.max (x, maxlist (xs))
```

3.2.3 Overly Complex Implementations

```
(*
   val divisible : int * int -> bool
   divisible (n, m) returns true if n is divisible by m

   Invariants: n >= 0 and m > 0
   Effects: none
*)
fun divisible(n: int, m: int): bool =
    if n = m then
      true
    else
      if n < m then
        false
      else
        divisible (n - m, m)
```

In this case, while the function is correct, the implementation is overly complex: a natural number `n` is divisible by a positive natural number `m`, if the remainder of the division of `n` by `m` is equal to zero. The remainder can be obtained by using the `mod` operator. Moreover, the previous implementation has complexity $O(n/m)$, while the `mod` operation can be done in constant time. A better solution would be:

```
(*
   val divisible : int * int -> bool
   divisible (n, m) returns true if n is divisible by m

   Invariants: x and y are strictly positive
   Effects: none
*)
fun divisible(n: int, m: int): bool = (n mod m) = 0
```

3.3 Helper Functions

3.3.1 Non-local Helper Functions

```
(*
   val maxlisthelper : int * int list -> int

   maxlisthelper (max, list) computes the maximum of max and each
                             value in the list

   Invariants: none
   Effects: none
*)

fun maxlisthelper (max: int, []: int list): int =
    max
  | maxlisthelper (max: int, x::xs: int list): int =
    let
      val newmax: int = if max > x then max else x
    in
      maxlisthelper (newmax, xs)
    end

(*
   val maxlist : int list -> int

   maxlist (l) computes the maximum value contained in the list

   Invariants: l is not empty
   Effects: raises Invalid_argument if the list is empty
*)

fun maxlist ([]: int list): int =
    raise Invalid_argument
  | maxlist (x::xs: int list): int =
    maxlisthelper (x, xs)
```

In this case the `maxlisthelper` function is an auxiliary function that is used in the definition of `maxlist`. Since it has no specific use besides within `maxlist`, it should be defined as a local function.

```
(*
   val maxlist : int list -> int

   maxlist (l) computes the maximum value contained in the list

   Invariants: l is not empty
   Effects: raises Invalid_argument if the list is empty

  The function maxlist uses the local helper function:
   val maxlisthelper : int * int list -> int

   maxlisthelper (max, list) computes the maximum of max and each
                             value in the list

   Invariants: none
   Effects: none
*)

fun maxlist ([]: int list): int =
    raise Invalid_argument
  | maxlist (x::xs: int list): int =
    let
      fun maxlisthelper (max: int, []: int list): int =
          max
        | maxlisthelper (max: int, x::xs: int list): int =
          let
            val newmax: int = if max > x then max else x
          in
            maxlisthelper (newmax, xs)
          end
    in
      maxlisthelper (x, xs)
    end
```

3.3.2 Multiple Definitions of the Same Local Helper Function

```
(*
   val maxlist : int list -> int

   maxlist (l) computes the maximum value contained in the list

   Invariants: l is not empty and every element of l is positive.
   Effects: none

  The function maxlist uses the local helper function:

   val max: int * int -> int

   max (x,y) returns the maximum of two numbers.

   Invariants: none
   Effects: none

*)
```

```
fun maxlist ([]: int list): int = 0
  | maxlist (x::xs: int list): int =
    let
      fun max (x: int, y: int): int =
        if x > y then x else y
    in
      max (x, maxlist (xs))
    end

(*
   val maxfun int * (int -> int) * (int -> int) -> int

   maxfun (x, f, g) returns the maximum of f(x) and g(x).

   Invariants: f and g are defined at x.
   Effects: none

  The function maxfun uses the local helper function:

   val max: int * int -> int

   max (x,y) returns the maximum of two numbers.

   Invariants: none
   Effects: none

*)
fun maxfun (x: int, f: int -> int, g: int -> int): int =
    let
      fun max (x: int, y: int): int =
        if x > y then x else y
    in
      max (f (x), g (x))
    end
```

In this case the function `max: int * int -> int` is defined as a local helper function in both `maxlist` and `maxfun`. It is better style to define `max` as a non local helper function, especially given the general nature of the function `max`.

```
(*
   val max: int * int -> int

   max (x,y) returns the maximum of two numbers.

   Invariants: none
   Effects: none

*)

fun max (x: int, y: int): int =
    if x > y then x else y

(*
   val maxlist : int list -> int

   maxlist (l) computes the maximum value contained in the list

   Invariants: l is not empty and every element of l is positive.
   Effects: none
*)

fun maxlist ([]: int list): int = 0
  | maxlist (x::xs: int list): int =
    max (x, maxlist (xs))

(*
   val maxfun int * (int -> int) * (int -> int) -> int

   maxfun (x, f, g) returns the maximum of f(x) and g(x).

   Invariants: f and g are defined at x.
   Effects: none
*)
fun maxfun (x: int, f: int -> int, g: int -> int): int =
    max (f (x), g (x))
```

3.4 Pattern Matching

3.4.1 Redundant Cases

```
fun fib (0: int): int = 1
  | fib (1: int): int = 1
  | fib (2: int): int = 2
  | fib (n: int): int = fib(n-1) + fib(n-2)
```

In this case the additional base case for `n = 2` is not necessary. A better definition for the function `fib` would be:

```
fun fib (0: int): int = 1
  | fib (1: int): int = 1
  | fib (n: int): int = fib(n-1) + fib(n-2)
```

3.4.2 Pattern Matching Besides Function Arguments

```
fun second (nil: int list) : int option = NONE
  | second (x::nil: int list) : int option = NONE
  | second (x::xs: int list) : int option =
    let
      val second::rest : int list = xs
    in
      SOME second
    end
```

In this case, the `let` local definition is used to bind the second element of the list of the local variable `second`. While the code is correct and sometimes it is necessary to use pattern matching besides function argument, in this case this is not good style: the pattern matching should have been done in the argument pattern instead, making it evident the intention. Moreover this approach works correctly only because the pattern `(x::nil)` appears before the pattern `(x::xs)`. A better solution would be:

```
fun second (nil: int list) : int option = NONE
  | second (x::nil: int list) : int option = NONE
  | second (x::second::rest: int list) : int option =
    SOME second
```

3.4.3 Bindings to Identifiers and the Wildcard Pattern _

```
fun second (nil: int list) : int option = NONE
  | second (x::nil: int list) : int option = NONE
  | second (x::second::rest: int list) : int option =
    SOME second
```

The previous function declaration can be further improved by replacing bindings in the patterns to identifiers that are never used with the wildcard pattern _.

```
fun second (nil: int list) : int option = NONE
  | second (_::nil: int list) : int option = NONE
  | second (_::second::_: int list) : int option =
    SOME second
```

3.4.4 Multiple Patterns for Function Arguments Corresponding to the Same Expression

```
fun second (nil: int list) : int option = NONE
  | second (_::nil: int list) : int option = NONE
  | second (_::second::_: int list) : int option =
    SOME second
```

The pattern in the previous definition is used only to discriminate between a list with at least 2 elements, and a list with fewer than 2 elements. The same result can be obtained by the more compact:

```
fun second (_::second::_: int list) : int option =
    SOME second
  | second (_: int list) : int option = NONE
```

Beware, however, that the use of the wildcard pattern _ as well as overlapping patterns (multiple pattern that can match the same value), can be sometimes misleading. For instance, in the last example _ and `_::second::_` both match the list `[1,2,3]`: however the first pattern will be the one chosen because it appears first.

3.5 Shadowing of Previous Bindings

```
fun convolve (1: int, f: int -> int, g: int -> int): int =
    f(1) * g(1)
  | convolve (k: int, f: int -> int, g: int -> int): int =
    let
      val f = fn (n: int) => f(n+1)
    in
      f(0) * g(k) + convolve (k-1, f, g)
    end
```

The function `convolve` defines a binding for `f` that overrides the previous binding of `f`. In general, it is bad style to define new bindings for an identifier which shadow previous bindings unless strictly necessary. In this case, it would have been better to create a new binding with a different name, or use the lambda expression directly.

```
fun convolve (1: int, f: int -> int, g: int -> int): int =
    f(1) * g(1)
  | convolve (k: int, f: int -> int, g: int -> int): int =
    let
      val f' = fn (n: int) => f(n+1)
    in
      f(1) * g(k) + convolve (k-1, f', g)
    end
```

or:

```
fun convolve (1: int, f: int -> int, g: int -> int): int =
    f(1) * g(1)
  | convolve (k: int, f: int -> int, g: int -> int): int =
    f(1) * g(k) + convolve (k-1, fn (n: int) => f(n+1), g)
```