General Computer Science I (320101) Fall 2014

September 25. 2014

Abstract

This document accompanies the traditional SML tutorial in GenCS. It contains a sequence of simple (but increasingly difficult) problems designed to practice the art of recursive programming.

The problems in this document are intended for self-study, they are supplied with solutions.

As most students have never programmed SML (or programmed at all), most students only manage to solve the first five to 10 problems. This is to be expected, and sufficient, since the purpose of the tutorial is to get students started at all and jointly remove the first roadblocks, so that they can continue alone (or in groups) after that.

The problems from the first three assignments should be doable after the first two lectures on SML, the later problems can be tackled as the lecture progresses.

Contents

Practice Problems 1: SML Basics

Problem 1.1 (Head and Tail)
Program the following elementary list functions in SML and test them on three examples each. Please note that your program must work on every input except the empty list.

1. head takes a list L as input and returns the first element of L
2. tail takes a list L as input and returns the list consisting of all elements of L in original order except the first one.

Solution:

```
- fun head(h::t)=h;
- fun tail(h::t)=t;
```

Problem 1.2 Define the member relation which checks whether an integer is member of a list of integers. The solution should be a function of type int ∗ int list −> bool, which evaluates to true on arguments n and l, iff n is an element of the list l.

Solution: The simplest solution is the following

```
fun member(n,nil) = false
  | member(n,h::r) = if n=h then true else member(n,r);
```

The intuition here is that a is a member of a list l, iff it is the first element, or it is a member of the rest list.

Note that we cannot just use member(n,n::r) to eliminate the conditional, since SML does not allow duplicate variables in matching. But we can simplify the conditional after all: we can make use of SML's **orelse** function which acts as a logical "or" and get the slightly more elegant program

```
fun member(n,nil) = false
  | member(n,h::r) = (n=h) orelse member(n,r);
```

Problem 1.3 Define the subset relation. Set T is a subset of S iff all elements of T are also elements of S. The empty set is subset of any set.

Hint: Use the member function from ?prob.member?

Solution: Here we make use of SML's **andalso** operator, which acts as a logical "and"

```
fun subset(nil,_) = true
  | subset(x::xs,m) = member(x,m) andalso subset(xs,m);
```

The intuition here is that $S \subseteq T$, iff for some $s \in S$ we have $s \in T$ and $S\backslash\{s\} \subseteq T$.

Problem 1.4 Define functions to zip and unzip lists. zip will take two lists as input and create pairs of elements, one from each list, as follows: zip [1,2,3] [0,2,4] $\leadsto$ [[1,0],[2,2],[3,4]]. unzip is the inverse function, taking one list of tuples as argument and outputing two separate lists. unzip [[1,4],[2,5],[3,6]] $\leadsto$ [1,2,3] [4,5,6].

Solution: Zipping is relatively simple, we will just define a recursive function by considering 4 cases:

```
fun zip nil nil = nil
  | zip nil l = l
  | zip l nil = l
  | zip (h::t) (k::l) = [h,k]::(zip t l)
```

Unzipping is slightly more difficult. We need map functions that select the first and second elements of a two-element list over the zipped list. Since the problem is somewhat under-specified by the example, we will put the rest of the longer list into the first list. To avoid problems with the empty tails for the shorter list, we use the mapcan function that appends the tail lists.

```
fun mapcan(f,nil) = nil | mapcan(f,h::t) = (f h)@(mapcan(f,t))
fun unzip (l) = if (l = nil) then nil
                else [(map head l),(mapcan tail l)]
```

Problem 1.5 Declare a (abstract) data type for natural numbers and one for lists of natural numbers in SML. Write an SML function that given two natural number n and m (as a constructor term) creates the list $[n, n+1, \ldots, m-1, m]$ if $n \leq m$ and raises an exception otherwise.

Problem 1.6 Write three SML functions nth, take, drop that take a list and an integer as arguments, such that

1. nth(xs,n) gives the n-th element of the list xs.
2. take(xs,n) returns the list of the first n elements of the list xs.
3. drop(xs,n) returns the list that is obtained from xs by deleting the first n elements.

In all cases, the functions should raise the exception Subscript, if n is negative or the list xs has less than n elements. We assume that list elements are numbered beginning with 0.

Solution:

```
exception Subscript
fun nth (nil,_) = raise Subscript
  | nth (h::t,n) = if n < 0 then raise Subscript
                   else if n=0 then h else nth(t,n−1)
fun take (l,0) = nil
  | take (nil,_) = raise Subscript
  | take (h::t,n) = if n < 0 then raise Subscript
                    else h::take(t,n−1)
fun drop (l,0) = l
  | drop (nil,_) = raise Subscript
  | drop (h::t,n) = if n < 0 then raise Subscript
                    else drop(t,n−1)
```

Problem 1.7 Write three SML functions rev, take, drop operating on lists (lst) and integers (n), such that

1. rev(lst) returns the list lst in reversed order.
2. last(lst,n) returns the list of the last n elements of the list lst.

Make only use of the :: and @ built-in operators or those functions you have defined yourself. Note that n is always a positive integer. Assume that list elements are numbered beginning with 0.

Solution:

```
fun rev [] = []
  | rev (x::xs) = (rev lst) @ [x];
fun take (l,0) = nil
  | take (nil,_) = raise Subscript
  | take (h::t,n) = if n < 0 then raise Subscript
                    else h::take(t,n-1)
fun last (l,n) = rev(take(rev(l),n))
```

Problem 1.8 Write an SML function that tabulates lists, i.e tabulate(f,n) evaluates to the list [f(0),...,f(n−1)]. As en example if f(x)=2∗x and n=[0,1,2] then tabulate(f,n)=[0,2,4]

Practice Problems 2: Arithmetics

Problem 2.1 (Factorial)
Write a recursive procedure fact : int −> real, that computes the factorial function $n! = 1 \cdot 2 \cdot \cdots \cdot n$, where $0! = 1$. We want the procedure fact to diverge for negative arguments (on an abstract interpreter without resource limitations).

What is the largest number n you can compute $n!$ for on your system?

Solution:

```
- fun fact(0)=1.0| fact(n)=real(n)*fact(n-1)
- fact(170)
val it = 7.25741561531E306 : real
- fact(171);
val it = inf : real
```

The largest value is ∞ (infinity)! But seriously: the largest factorial that can be computed is 170!.

Problem 2.2 Define a function mymax over SML lists of integers. Here, we take the maximum of an empty list to be the "default value" 0. You might want to consider a notation of type mymax(x::y::ys) and work with the first two elements of the list in the step case.

Solution: In this program, we have to consider three cases,

- the empty list, there we give the default value
- the singleton list where the maximum is the unique value of the list
- lists of length 2 or more: Here the intuitio for the recursion is that

$$\max([x,y,\ldots]) = \begin{cases} \max([y,\ldots]) & \text{if } x > y \\ \max([x,\ldots]) & \text{if } y \leq x \end{cases}$$

So we naturally get the following implementation:

```
fun mymax (nil) = 0
  | mymax ([x]) = x
  | mymax (x::y::xs) = if x > y then maxint(x::xs) else maxint (y::xs)
```

Note that with this recursion the first case is never reached, exept if an empty list was given as the original argument.

Problem 2.3 Generalize the mymax function from ?prob.maxint? to general lists with an ordering function (i.e. a function of type ('a * 'a) −>bool) and a default value of type 'a, which are passed as arguments. Given a list l an ordering relation ord, and a default value d, calling gmax(l,ord,d) evaluates to the ord-maximal element of l, or d, if l is empty.

Test this on lists of digits with the ordering relation given by $1 < 3 < 2 < 5 < 7 < 8 < 4 < 9 < 6$.

Solution: Generalizing the function is rather easy, we only add two arguments ord and default to the argument pattern of the function, and use them instead of the fixed values in mymax.

```
fun gmax (nil,_,default) = default
  | gmax ([x],_,_) = x
  | gmax (x::y::y::xs,ord,default) = if ord(x,y)
                                       then gmax(x::xs,ord,default)
                                       else gmax(y::xs,ord,default)
```

For the test, we need to come up with an predicate for the ordering relation given by $1 < 3 < 2 < 5 < 7 < 8 < 4 < 9 < 6$. We make use of the fact that SML looks at overlapping clauses in top-to-bottom order.

```
fun myord (1,_) = true | myord (_,1) = false
  | myord (3,_) = true | myord (_,3) = false
  | myord (2,_) = true | myord (_,2) = false
  | myord (5,_) = true | myord (_,5) = false
  | myord (7,_) = true | myord (_,7) = false
  | myord (8,_) = true | myord (_,8) = false
  | myord (4,_) = true | myord (_,4) = false
  | myord (9,_) = true | myord (_,9) = false
  | myord (6,_) = true | myord (_,6) = false
```

of course we get a warning that the match is non-exhaustive (we have not dealt with all integers) but for the moment we do not care.

Problem 2.4 (Infinite Precision Arithmetics)
We represent natural numbers of arbitrary length by non-empty lists of digits. Write SML functions for the arithmetical operations of addition, multiplication, integer division and modulo.

Problem 2.5 (Binary and Decimal Addition)
Define binary and decimal addition, and multiplication on lists of digits.

Hint: This is just a sneaky way of getting you to practice with lists.

For the decimal addition function, we represent natural numbers as lists of of digits, e.g. [5,3,4] for the number 534. Now the function badd works as follows: badd([2,8],[1,3]) evaluates to [4,1].

The binary addition function is similar, only have it operates on lists of binary digits, i.e. the numbers 1 and 0.

Problem 2.6 Program the function f with $f(x) = x^2$ on unary natural numbers without using the multiplication function.

Solution: We will use the abstract data type mynat

```
datatype mynat = zero | s of mynat
fun add(n,zero) = n | add(n,s(m))=s(add(n,m))
fun sq(zero)=zero|sq(s(n))=s(add(add(sq(n),n),n))
```

Problem 2.7 (Floating Point Powers)
Write a recursive procedure power : real ∗ int −> real, that computes the power x^n for a real number x and a natural number n by floating point operations. What is the result of $power(3.0, 100)$, is this really the number 3^{100} (discuss)?

Solution:

```
− fun power(x:real,0) = 1.0 | power(x,n) = x ∗ power(x,n−1);
− power(3.0,100)
val it = 5.15377520732E47 : real
```

This is not the number 3^{100}, since it is a floating point approximation; The result has 36 zeros at the end, but 3^{100} does not.

Practice Problems 3: Sorting

Problem 3.1 (Sorting a list)
Write a function that takes a list of strings and sorts it in ascending order like in dictionary.

Solution:

```
fun selmax(nil:string list) = nil
| selmax(a::nil) = a::nil
| selmax(a::b::t) =
          if a>b then b::selmax(a::t)
          else a::selmax(b::t);
fun descend(nil: string list) = nil
| descend(h::nil) = h::nil
| descend(l) = let
       val max = rev(selmax(l))
       in hd(max)::descend(tl(max))
     end;
fun ascend(l) = rev(descend(l));
```

Problem 3.2 Write a function split that takes a string (a sentence) and returns a list of all pairs of strings one can get by splitting the sentence between any two words. A word is defined as a sequence of symbols between two spaces or a space and nothing. For example,

```
split "We really love SML!";
val it = [("We", "really love SML!"), ("We really", "love SML!"),
("We really love", "SML!")]
```

Solution:

```
    fun split(s) = let
        fun go(sack, nil) = nil |
        go(sack, #" "::tl) = (implode(sack), implode(tl))::go(sack@[#" "], tl) |
        go(sack, w::tl) = go(sack@[w],tl)
    in
        go([], explode(s))
  end
```

Problem 3.3 (Ordering Function)
Write an SML function that takes an int list and sorts it in ascending order. For example:

```
sort [7,4,1,3];
val it = [1,3,4,7];
```

Solution:

```
fun put(x,nil) = [x] |
    put(x,h::t) = if (x<h) then x::h::t else h::put(x,t);
fun sort(nil) = nil |
    sort(h::t) = put(h,sort(t));
```

Practice Problems 4: Data Types

Problem 4.1 Using the abstract data type of truth functions, give the defining equations for a function myif that takes three arguments, such that myif(X,Y,Z) behaves like "if X then Y, else Z".

Hint: There is a control structure **if**. . . **then**. . . **else** in SML, of course you are not supposed to use that.

Problem 4.2 Write three variants of the member function in SML, where member(x,xs) returns true, iff x is an element in xs.

1. the first variant should not use another function.
2. the second variant should be non-recursive, using the function myexists (write that as well) that takes a property p and a list l as arguments and returns true, iff there is an element a in l such that p(a) evaluates to true.
3. the third variant should be non-recursive using the foldl function.

Solution:

```
fun member1 (x,nil) = false
  | member1 (x,h::t) = if (h=x) then true else member1(x,t)
fun myexists (_,nil) = false
  | myexists (p,h::t) = p(h) orelse myexists(p,t)
fun member2 (x,xs) = myexists (fn (y) => x=y) xs
fun member3 (x,xs) = foldl (fn (y,b) => b orelse x=y) false
```

Problem 4.3 (Your own lists)
Define a data type mylist of lists of integers with constructors mycons and mynil. Write translators tosml and tomy to and from SML lists, respectively.

Solution: The data type declaration is very simple

```
datatype mylist = mynil | mycons of int * mylist;
```

it declares three symbols: the base type mylist, the individual constructor mynil, and the constructor function mycons.

The translator function tosml takes a term of type mylist and gives back the corresponding SML list; the translator function tomy does the opposite.

```
fun tosml mynil = nil
  | tosml mycons(n,l) = n::tosml(l)
fun tomy nil = mynil
  | tomy (h::t) = mycons(h,tomy(t))
```

Problem 4.4 Declare a data type myNat for unary natural numbers and NatList for lists of natural numbers in SML syntax, and define a function that computes the length of a list (as a unary natural number in mynat). Furthermore, define a function nms that takes two unary natural numbers n and m and generates a list of length n which contains only ms, i.e. nms(s(s(zero)),s(zero)) evaluates to construct(s(zero),construct(s(zero),elist)).

Solution:

```
datatype mynat = zero | s of mynat;
datatype natlist = elist | construct of mynat * natlist;
fun length (nil) = zero | length (construct (n,l)) = s(length(l));
fun nms(zero,m) = elist | nms(s(n),m) = construct(m,nms(n));
```

Problem 4.5 (Unary natural numbers)
Define a **datatype** nat of unary natural numbers and implement the functions

- add = **fn** : nat * nat −> nat (adds two numbers)
- mul = **fn** : nat * nat −> nat (multiplies two numbers)

Solution:

```
datatype nat = zero | s of nat;
fun add(zero:nat,n2:nat) = n2
  | add(n1,zero) = n1
  | add(s(n1),s(n2)) = s(add(n1,s(n2)));
fun mult(zero:nat,_) = zero
  | mult(_,zero) = zero
  | mult(n1,s(zero)) = n1
  | mult(s(zero),n2) = n2
  | mult(n1,s(n2)) = add(n1,mult(n1,n2));
```

Problem 4.6 (Nary Multiplication)
By defining a new datatype for n-tuples of unary natural numbers, implement an n-ary multiplications using the function mul from ?prob.natoper?. For $n = 1$, an n-tuple should

be constructed by using a constructor named first; for $n > 1$, further elements should be prepended to the first by using a constructor named next. The multiplication function nmul should return the product of all elements of a given tuple.

For example,

```
nmul(next(s(s(zero)),
        next(s(s(zero)),
          first(s(s(s(zero)))))))
```

should output s(s(s(s(s(s(s(s(s(s(s(s(zero)))))))))))) since $223 = 12$.

Solution:

```
datatype tuple = first of nat | next of nat*tuple;
fun nmult(first(num)) = num |
     nmult(next(num, rest)) = mult(num, nmult(rest));
```

Practice Problems 5: Higher-Order Functions

Problem 5.1 Write a recursive higher-order SML function mapcan that maps a list-valued function f over a list and appends all the result lists to a single list. What is the SML type of this function (explain).

Write versions of map and mapcan that map a binary function over two lists. What are the SML types of these functions (explain).

Solution:

```
fun mapcan(f,nil) = nil | mapcan(f,h::t) = (f h)@(mapcan(f,t))
```

The SML type of mapcan is ('a −> 'b list) ∗ 'a list −> 'b list. The function mapcan takes two arguments: a list valued function, which has the type ('a −> 'b list) and a list of some type 'a list. The output of the function is the concatenation of the resulting lists so it is again a list 'b list.

```
fun map2(f,nil,_) = nil
  | map2(f,_,nil) = nil
  | map2(f,h1::t1,h2::t2) = f(h1,h2)::map2(f,t1,t2)
```

map2 has the type ('a ∗ 'b −> 'c) ∗ 'a list −> 'c list because the function map takes two arguments: a binary function (type 'a ∗ 'b −> 'c) and a list (type 'a list). The result is again a list of some type 'c list).

```
fun mapcan2(f,nil,_) = nil
  | mapcan2(f,_,nil) = nil
  | mapcan2(f,h1::t1,h2::t2) = f(h1,h2)@mapcan2(f,t1,t2)
```

mapcan2 has type ('a ∗ 'b −> 'c) ∗ 'a list −> 'c list which is the same as above because the function mapcan takes the same arguments and returns a list. the difference is (not obvious from type) that the mapcan function does not return 'a list list type as map does if function f is a list valued function.

Note that in the above 'a, 'b, 'c define any SML type. Most of the above given functions work with all SML types, so a defined type like int for example is not required.

Problem 5.2 Write a non-recursive variant of the member function from ?prob.member? using the foldl function.

Solution:

```
fun member (x,xs) = foldl (fn (y,b) => b orelse x=y) false
```

Problem 5.3 (Higher-Order Functions)
Write three higher-order functions that take a predicate p (a function with result type bool) and a list l.

- myfilter that returns the list of all members a of l where $p(a)$ evaluates to true.
- myexists that returns true if there is at least one element a in l, such that $p(a)$ evaluates to true.
- myforall that returns true if $p(a)$ evaluates to true on all elements of l.

Hint: If you are in need of a test predicate, you can work on a list l of ints and use the "even number" predicate:

```
fun even n = n mod 2 = 0;
```

Hint: We expect a different solution here than the solution for the next problem.

Solution:

```
fun myfilter p nil = nil
  | myfilter p h::t =
    if (p h) then h :: myfilter p t else myfilter p t

fun myexists p nil = false
  | myexists p h::t =
    if (p h) then true else myexists p t

fun myforall p nil = true
  | myforall p h::t =
    if (p h) then myforall p t else false
```

For the last two, we can make use of the predefined functions **orelse** and **andalso**, which are useful abbreviations: e_1 **andalso** e_2 abbreviates **if** e_1 **then** e_2 **else** false and e_1 **orelse** e_2 abbreviates **if** e_1 **then** true **else** e_2. Using this, we can write

```
fun myforall f nil = true
  | myforall f (x::xr) = f x andalso myforall f xr

fun myexists f nil = false
  | myexists f (x::xr) = f x orelse myexists f xr
```

Problem 5.4 (List functions via folding)
Write the following procedures using foldl or foldr

1. length which computes the length of a list
2. concat, which gets a list of lists and concatenates them to a list.

3. map, which maps a function over a list
4. myfilter, myexists, and myforall from the previous problem.

Solution:

```
fun length xs = foldl (fn (x,n) => n+1) 0 xs
fun concat xs = foldr op@ nil xs
fun map f = foldr (fn (x,yr) => (f x)::yr) nil
fun myfilter f =
    foldr (fn (x,ys) => if f x then x::ys else ys) nil
fun myexists f = foldl (fn (x,b) => b orelse f x) false
fun myall f = foldl (fn (x,b) => b andalso f x) true
```

Problem 5.5 (Understaning map)
Given the SML higher-order function **fun** f x = **fn** (y) => y::x and the list **val** l = [1,2,3].

- Determine the types of f and map (f l)
- evaluate the exression map (f l) l

Solution:

```
- fun f x = fn (y) => y::x;
val f = fn : 'a list -> 'a -> 'a list
- map (f l);
val it = fn : int list -> int list list

- map (f l) l;
val it = [[1,1,2,3],[2,1,2,3],[3,1,2,3]] : int list list
```

Problem 5.6 Write a function napply that takes a function f:int−>int and an integer n, and returns the result of n applications of f to itself, starting with number n as its argument. What does napply(**fn** x=>x+1, n) numerically compute? Is there any n for which napply(**fn** x=>x div 2, n) returns a non-zero value?

Solution: fun napply (f, n) = let fun loop(0) = n — loop(a) = f(loop(a-1)) in loop(n) end

---