CAS CS 320
Lec 2

Introduction to SML (cont.)

1. Reusing names
    a. val x = 1 (* declaration *)
       val y = x + x
       val y = y + 1
2. First-order
    a. val f = fn x => x + x
       val a = f(2) (* function call *)
       val b = f 2 (* same function call *)
       val rec f = fn x => if x>0 then x * f(x-1) else 1 (* if the function is recursive, need to have the word "rec" when declaring variable *)
       OR
       fun f(x) = if …
3. Exception (raising exception)
    a. exception MyErrorExn
       fun g(x) = raiseMyErrorExn
       val x = f(5) handle Overflow => 0
4. Second problem
    a. fun isPrime (n0: int): bool = raise MyNotImplementedExn
       fun isPrime_helper(n0: int, i0: int): bool =
            if i0 >= n0
                then true
            else if (n0 mod i0) <> 0 then isPrime_helper(n0, i0 + 1)
            else false
       fun isPrime(n0: int) =
            if n0 <= 1
                then false
            else isPrime_helper(n0, 2)

b. fun isPrime(n0: int): bool =
    if n0 <= 1
        then false
    else isPrime_helper(n0, 2)
   and (* for mutual recursion *) → compiler checks all the name first
   isPrime_helper(n0: int, i0: int): bool =
    if i0 >= n0
        then true
    else if (n0 mod i0) <> 0 then isPrime_helper(n0, i0 + 1)
    (* <> is != *)
    else false
c. Check TypeCheck → use the "use" command to upload
    i. Example
        1. stdIn: 19.8 - 19.26 (line number) → goto-line 24
           Error: operator and operand don't agree
           operator domain: int * int
           operand: int
           in expression:
               isPrime_helper n0
d. Go to the testing directory → run

5. Hiding certain information (making low code) → due to possible changes in the future
    a. Use "let" and "in", and "end"(all the names and code will be local and only be visible to the programmer) → only the code written between "let" and "in" is not visible to the outside world, code between "in" and "end" will be shown to the outside world
        i. let
           __
           in
           __
           end
        ii. fun isPrime(n0: int): bool =
            let
            fun
            helper(i0: int): bool =
            if i0 >= n0
                then true
            else if (n0 mod i0) <> 0
                then helper(i0 + 1)
            else false
            in

```
if n0 <= 1
        then false
else false
end
```

6. Third problem
    a. exception MyAssertExn
    fun assert(claim: bool) =
    if not(claim)
        then raise MyAssertExn
    else () (* if "if" is used, else needs to exist as well *)

    fun int2str (x: int): string =
    if x < 10
        then String.str(Char.chr(Char.ord(#"0") + x mod 10))
    else (* recursion *)
        int2str(x div 10) ^
        String.str(Char.chr(Char.ord(48) + x mod 10)))
        (* ^ is a string concatenation, 48 is the ASCII code for 0 *)
        (* Char.ord → char to int, Char.chr → int to char *)
        (* String.str → char to string *)

    OR

    fun int2str (x: int): string =
    let
    val _ = assert (x >= 0) (* check whether the integer given is greater than 0 *)
    in

    if x < 10
        then String.str(Char.chr((Char.ord(#"0") + x mod 10)))
    else (* recursion *)
        int2str(x div 10) ^
        String.str(Char.chr((Char.ord(#"0") + x mod 10)))

    b. Negative integer → ~1, ~12345
    c. Defensive programming → assert that the arguments are good (stop errors as soon as possible in case of partial function)

7. Fourth problem
    a. fun str2int (cs: string): int = raise MyNotImplementedExn
    fun str2int (cs: string): int =
    let
    val n0 = String.size(cs)
    fun helper(i0: int): int =
    if i0 <= 0
        then 0
    else
        10 * helper(i0 - 1) + Char.ord(String.sub(cs, i0 - 1)) - Char.ord(#“0”)
    in
        helper(n0)
    end

    val myans1 = str2int(“12345”)