1 Introductory Examples

- Function Syntax

```
fun increment (n:int) : int = n + 1;


fun sumRange (n:int, m:int) : int =
   (n + m) * (m - n + 1) div 2
;


fun fact (n:int) : int =
   if n <= 0
   then 1
   else n * fact (n - 1)
;
```

2 Bindings

- Non-local

```
   - val x = 2;
```

- Local

```
fun sumRange (n:int, m:int) : int =
   let
      val numValues = m - n + 1;
      val each = m + n;
   in
      numValues * each div 2
   end
;
```

3 Data Types

- Tuples

```
   - (1,2);
   val it = (1,2) : int * int
   - val x = (1, true, 7.2);
   - #2 x;
   - #3 x;
   - #4 x;
   stdIn:34.1-34.5 Error: operator and operand don't agree [record labels]
     operator domain: {4:'Y; 'Z}
     operand:         int * bool * real
     in expression:
       (fn {4=4,...} => 4) x
```

```
- fun sumRange (t:int * int) : int =
   (#1 t + #2 t) * (#2 t - #1 t + 1) div 2
;
```

- Records

```
- {foo = 1, bar = 2};
- #foo it;

- fun translate ({x:real,y:real}, dx:real, dy:real) : {x:real, y:real} =
   {x=x + dx, y=y + dy}
;
```

- Lists

```
- [1,2,3];
val it = [1,2,3] : int list
- [];
val it = [] : 'a list
- val x = [1,2,3];
- 0::x;
- hd x;
- tl x;
- hd x;
- tl x;
- x @ [4,5];

- fun length (L:'a list) : int =
   if null L
   then 0
   else 1 + length (tl L)
;
```

- Strings

```
- "abc" ^ "def";
- str(#"a");       (* favor str over Char.toString *)
- implode (explode "abc");
```

4 Decision Making and Pattern Matching

- Conditional - already seen — an expression!!

```
- if true
= then 1
= else 2;
- 7 + (if true then 1 else 2);
```

- Case

```
- fun length (L:'a list) : int =
   (case L of
      [] => 0
    | (x::xs) => 1 + length xs
   )
;

(* can promote case/pattern-matching to parameter list *)
- fun length ([]:'a list) : int = 0
    | length (x::xs:'a list) : int = 1 + length xs
;

(* write sum with patterns *)
- fun sum [] = 0
    | sum (x::xs) = x + (sum xs)
;
```

5 Example

```
fun unzip ([] :('a * 'b) list) : 'a list * 'b list = ([], [])
  | unzip ((x,y)::zs :('a * 'b) list) : 'a list * 'b list =
      let
         val (xs, ys) = unzip zs;
      in
         (x::xs, y::ys)
      end
;
```

6 More about Functions

- Currying

```
- fun add1 (x:int) : int = x + 1;
- add1 2;

- fun add(x:int,y:int) : int = x + y;
- add(2,3);

- fun add (x:int) (y:int) : int = x + y;
- add 2 3;
- add (2,3);
- add 1;
```

- Mutual Definition

```
(* useful later this quarter *)
- fun odd 0 = false
=    | odd n = even (n - 1)
= and
=     even 0 = true
=   | even n = odd (n - 1)
= ;
```

- Anonymous

```
- fn x => x + 1;
- (fn x => x + 1) 2;
```

7 Higher-Order Functions

```
- fun square x = x * x;

- fun apply f x = f x;
- apply square 3;

- fun sumsquares [] = 0
    | sumsquares (x::xs) = square x + sumsquares xs
;
- sumsquares [1,2,3];

- fun sumF [] f = 0
    | sumF (x::xs) f = f x + sumF xs f
;
- sumF [1,2,3] square;

- fun map f [] = []
    | map f (x::xs) = (f x) :: map f xs
;
- mymap square [1,2,3];

- fun fake_right f [] comb base = base
    | fake_right f (x::xs) comb base = comb (f x, fake_right f xs comb base)
;

- fun fold_right f b [] = b
    | fold_right f b (x::xs) = f (x, fold_right f b xs)
;
- fold_right (op +) 0 [1,2,3];
- foldr (op +) 0 [1,2,3];

- fun unzip L = foldr (fn ((x,y), (xs,ys)) => (x::xs, y::ys)) ([], []) L;
```

8 Option Type

- There is a rather useful type for writing what we might consider partial functions (i.e., those for which some inputs do not map to meaningful results). This is the `'a option` type.

```
- SOME 1;
val it = SOME 1 : int option
```

```
- val opt = SOME 1;
val opt = SOME 1 : int option
- isSome opt;
val it = true : bool
- valOf opt;
val it = 1 : int
- isSome NONE;
val it = false : bool
- valOf (NONE : int option);

uncaught exception Option
  raised at: smlnj/init/pre-perv.sml:21.27-21.33
- case opt of
=    SOME _ => true
=  | NONE => false
= ;
val it = true : bool
```

9 Algebraic Data Types

- Basic

```
- datatype foo = Bar | Baz;
- val x = Bar;
- case x of
        Bar => 1
      | Baz => 2
  ;
- datatype union =
        INT of int
      | BOOL of bool
      | TUPLE of (int * int)
  ;
- val y = TUPLE (3,2);
- case y of
        (INT x) => x
      | (BOOL true) => 1
      | (BOOL false) => 0
      | (TUPLE (a,b)) => a + b
  ;
```

- Parameterized

```
- datatype 'a pair = PAIR of 'a;
- PAIR(1,2);
- datatype 'a tree = nil | node of ('a * 'a tree * 'a tree);
fun insert x nil = node (x , nil, nil)
  | insert x (node(y, l, r)) =
         if x < y
         then node(y, insert x l, r)
         else node(y, l, insert x r)
```

```
;     (* darn, only for int trees because of the < *)


(* generalized *)
fun insert comp x nil = node (x , nil, nil)
  | insert comp x (node(y, l, r)) =
         if comp(x,y)
         then node(y, insert comp x l, r)
         else node(y, l, insert comp x r)
;

- insert (op <);
```

10 Exceptions

```
exception Foo;
exception Bar of int;
raise Foo;
raise Bar 2;
fun f true = raise Foo
  | f false = raise Bar 2;

f true
   handle Foo => 1
        | Bar x => x
;
```

11 I/O

```
open TextIO;
print "Hello, world!\n";
val fstr = TextIO.openIn("file");
TextIO.input1 fstr;
TextIO.lookahead fstr;

(* sequence expressions *)
(output (stdErr, "Hello ");
 output (stdErr, "-- this is an error\n"));
```