SML

CSE 216 –PROGRAMMING ABSTRACTIONS

HTTPS://PPAWAR.GITHUB.IO/CSE216-S19/INDEX.HTML

SLIDES COURTESY:

DR. PAUL FODOR

STONY BROOK UNIVERSITY

Definition by Patterns

In SML functions can also be defined via patterns.

The general form of such definitions is:

```
fun <identifier>(<pattern1>) = <expression1>
| <identifier>(<pattern2>) = <expression2>
| ...
| <identifier>(<patternK>) = <expressionK>;
```

where the identifiers, which name the function, are all the same, all patterns are of the same type, and all expressions are of the same type.

The patterns are inspected in order and the first match determines the value of the function.

Example:

```
- fun reverse(nil) = nil
  | reverse(x::xs) = reverse(xs) @ [x];
val reverse = fn : 'a list -> 'a list
```

Sets with lists in SML

```
fun member(X,L) =
        if L=[] then false
        else if X=hd(L) then true
        else member(X,tl(L));
                        OR with patterns:
fun member(X,[]) = false
        | member(X,Y::Ys) =
                if (X=Y) then true
                else member(X,Ys);
member(1,[1,2]); (* true *)
member(1,[2,1]); (* true *)
member(1,[2,3]); (* false *)
```

Sets - UNION

```
fun union(L1,L2) =
      if L1=[] then L2
      else if member(hd(L1),L2)
            then union(tl(L1),L2)
            else hd(L1)::union(tl(L1),L2);
union([1,5,7,9],[2,3,5,10]);
       (* [1,7,9,2,3,5,10] *)
union([],[1,2]);
       (* [1,2] *)
union([1,2],[]);
       (* [1,2] *)
```

Sets - UNION patterns

```
fun union([],L2) = L2
      | union(X::Xs,L2) =
          if member(X,L2) then union(Xs,L2)
          else X::union(Xs,L2);
union([1,5,7,9],[2,3,5,10]);
       (* [1,7,9,2,3,5,10] *)
union([],[1,2]);
       (* [1,2] *)
union([1,2],[]);
       (* [1,2] *)
```

Sets - Intersection ∩

```
fun intersection(L1,L2) =
     if L1=[] then []
     else if member(hd(L1),L2)
     then hd(L1)::intersection(tl(L1),L2)
     else intersection(tl(L1),L2);

intersection([1,5,7,9],[2,3,5,10]);
     (* [5] *)
```

Sets - ∩ patterns

```
fun intersection([],L2) = []
      | intersection(L1,[]) = []
      | intersection(X::Xs,L2) =
            if member(X,L2)
            then X::intersection(Xs,L2)
            else intersection(Xs,L2);

intersection([1,5,7,9],[2,3,5,10]);
      (* [5] *)
```

Sets – subset

```
fun subset(L1,L2) = if L1=[] then true
     else if L2=[] then false
     else if member(hd(L1),L2)
          then subset(tl(L1),L2)
          else false;
subset([1,5,7,9],[2,3,5,10]);
     (* false *)
subset([5],[2,3,5,10]);
     (* true *)
```

Sets – subset patterns

```
fun subset([],L2) = true
        | subset(L1,[]) = if(L1=[])
                then true
                else false
        | subset(X::Xs,L2) =
                if member(X,L2)
                        then subset(Xs,L2)
                        else false;
subset([1,5,7,9],[2,3,5,10]);
        (* false *)
subset([5],[2,3,5,10]);
        (* true *)
```

Sets – equals

```
fun setEqual(L1,L2) =
    subset(L1,L2) andalso
subset(L2,L1);
setEqual([1,5,7],[7,5,1,2]);
    (* false *)
setEqual([1,5,7],[7,5,1]);
    (* true *)
```

Sets – minus patterns

```
fun minus([],L2) = []
    | minus(X::Xs,L2) =
        if member(X,L2)
            then minus(Xs,L2)
            else X::minus(Xs,L2);

minus([1,5,7,9],[2,3,5,10]);
    (* [1,7,9] *)
```

Sets - Cartesian product

```
fun product_one(X,[]) = []
        | product_one(X,Y::Ys) =
                (X,Y)::product_one(X,Ys);
product_one(1,[2,3]);
        (* [(1,2),(1,3)] *)
fun product([],L2) = []
        | product(X::Xs,L2) =
                union(product_one(X,L2),
                        product(Xs,L2));
product([1,5,7,9],[2,3,5,10]);
        (* [(1,2),(1,3),(1,5),(1,10),(5,2),
   (5,3),(5,5),(5,10),(7,2),(7,3),...] *)
```

Sets – Powerset

```
fun insert_all(E,L) =
        if L=[] then []
        else (E::hd(L)) :: insert_all(E,tl(L));
insert_all(1,[[],[2],[3],[2,3]]);
 (* [ [1], [1,2], [1,3], [1,2,3] ] *)
fun powerSet(L) =
        if L=[] then [[]]
        else powerSet(tl(L)) @
                insert_all(hd(L),powerSet(tl(L)));

powerSet([]);
powerSet([1,2,3]);
powerSet([2,3]);
```

Records

Records are structured data types of heterogeneous elements that are labeled

```
- {x=2, y=3};
```

The order does not matter:

```
- {make="Toyota", model="Corolla", year=2017, color="silver"}
= {model="Corolla", make="Toyota", color="silver", year=2017};
val it = true : bool


- fun full_name{first:string,last:string,
 age:int,balance:real}:string =
  first ^ " " ^ last;
       (* ^ is the string concatenation operator *)
val full_name=fn:{age:int, balance:real, first:string,
 last:string} -> string
```

Higher-Order Functions

In functional programming languages functions can be used in definitions of other, so-called higher-order, functions.

- The following function, **map**, applies its first argument (a function) to all elements in its second argument (a list of suitable type):

```
- fun map(f,L) =   if (L=[]) then []
 else f(hd(L))::(map(f,tl(L)));
val map = fn : (''a -> 'b) * ''a list -> 'b list
```

- We may apply **map** with any function as argument:

```
- fun square(x) = (x:int)*x;
val square = fn : int -> int
- map(square,[2,3,4]);
val it = [4,9,16] : int list
```

Higher-Order Functions

More map examples

- Anonymous functions:

```
- map(fn x=>x+1, [1,2,3,4,5]);

val it = [2,3,4,5,6] : int list

- fun incr(list) = map (fn x=>x+1, list);

val incr = fn : int list -> int list

- incr[1,2,3,4,5];

val it = [2,3,4,5,6] : int list
```

McCarthy's 91 function

McCarthy's 91 function:

```
- fun mc91(n) = if n>100 then n-10
  else mc91(mc91(n+11));

val mc91 = fn : int -> int

- map mc91 [101, 100, 99, 98, 97, 96];

val it = [91,91,91,91,91,91] : int list
```

Filter

Filter: keep in a list only the values that satisfy some logical condition/boolean function

```
- fun filter(f,l) =
    if l=[] then []
      else if f(hd l)
        then (hd l)::(filter (f, tl l))
        else filter(f, tl l);

val filter = fn : ('a -> bool) * 'a list -> 'a list

- filter((fn x => x>0), [~1,0,1]);

val it = [1] : int list
```

Currying

```
- fun f(a)(b)(c) = a+b+c;
val f = fn : int -> int -> int -> int
val f = fn : int -> (int -> (int -> int))
 OR
- fun f a b c = a+b+c;
- val inc1 = f(1);
val inc1 = fn : int -> int -> int
val inc1 = fn : int -> (int -> int)
- val inc12 = inc1(2);
val inc12 = fn : int -> int
- inc12(3);
val it = 6 : int
```

Composition

Composition is another example of a higher-order function:

```
- fun comp(f,g)(x) = f(g(x));
val comp = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
- val f = comp(Math.sin, Math.cos);
val f = fn : real -> real
      SAME WITH:
- val g = Math.sin o Math.cos;
      (*  Composition "o" is predefined  *)
val g = fn : real -> real
- f(0.25);
val it = 0.824270418114 : real
- g(0.25);
val it = 0.824270418114 : real
```

Mutually recursive function definitions

```
- fun odd(n) = if n=0 then false
                else even(n-1)
 and
        even(n) = if n=0 then true
                  else odd(n-1);
val odd = fn : int -> bool
val even = fn : int -> bool

- even(1);
val it = false : bool
- odd(1);
val it = true : bool
```

string and char

```
- "a";
val it = "a" : string
- #"a";
val it = #"a" : char
- explode("ab");
val it = [#"a",#"b"] : char list
- implode([#"a",#"b"]);
val it = "ab" : string
- "abc" ^ "def" = "abcdef";
val it = true : bool
- size ("abcd");
val it = 4 : int
```

string and char

```
- String.sub("abcde",2);
val it = #"c" : char
- substring("abcdefghij",3,4);
val it = "defg" : string
- concat ["AB"," ","CD"];
val it = "AB CD" : string
- str(#"x");
val it = "x" : string
```