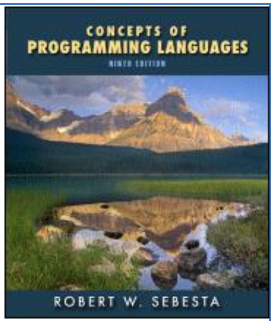
Standard ML



1. Ch15 – Functional Programming Languages
   - 15.7 ML
2. Standard ML of New Jersey
3. References
   - Programming in Standard ML, Robert Harper
   - Notes on Programming Standard ML of New Jersey, Riccardo Pucella

Read-eval-print loop

- REPL

```
– 2;                          – val x = 2;      (* value binding *)
val it = 2 : int              val x = 2 : int
– 2                           – x;
= ;                           val it = 2 : int
val it = 2 : int              – it;
– 2; 3;                       val it = 2 : int
val it = 2 : int              – ^Z              (* ^D in Unix *)
val it = 3 : int
```

Note: – exp; is equivalent to – val it=exp;

Value bindings

- Value bindings

```
– val x = 2;            (* value; not variable *)
val x = 2 : int
– fun f y = x+y;
val f = fn : int -> int
– val x = 3;
val x = 3 : int
– f x;
val it = 5 : int
```

However, SML is an impurely functional language.
It has variables and assignments.

Value bindings

- Value bindings
  - val x = 7; val y = x ;   (* sequence; ; is optional here*)

    val x = 7 : int

    val y = 7 : int
  - val x = 8 and y = x ;   (* simultaneous *)

    val x = 8 : int

    val y = 7 : int
  - val x : int = 9   (* explicitly typed style *)

    val x = 9 : int
  - val x = 9 : int

    val x = 9 : int

Loading program

- Loading file
  - File demo.ml

```
val x = 2;
fun f y = x+y;
```

  - Suppose the file is located in the working directory of sml
    – use "demo.ml";
    [opening demo.ml]
    val x = 2 : int
    val f = fn : int -> int
    val it = () : unit   (* value of the use call *)
  - If the file isn't in the working directory, then
    – use "c:\...*path*...\demo.ml";   (* / in Unix *)

Basic data types

- Basic data types

```
– 2; 2.3; true; #"c"; "snoopy";
val it = 2 : int
val it = 2.3 : real
val it = true : bool
val it = #"c" : char
val it = "snoopy" : string

– ();
val it = () : unit
```

Note: The unit type has the 0-tuple () as its only member.

Basic data types

- Operators and functions
  - int ~, +, -, *, div, mod, =, <>, >, >=, <, <=

    real ~, +, -, *, /, =, <>, >, >=, <, <=

    bool not, andalso, orelse, =, <>

    string ^
  - – 3 - ~5; (* space between - and ~ *)

    val it = 8 : int

    – 5 mod 3; (* infix *)

    – op mod(5,3); (* prefix *)

    val it = 2 : int

    – "snoopy" ^ "pluto"; (* op^("snoopy" ,"pluto"); *)

    val it = "snoopypluto" : string

Type constraints

- Type constraints
  - Operands must be of the same type

```
– 3 + 4.5;
stdIn:1.1-1.6  Error: operator and operand don't agree
    operator domain        int * int
    operand:               int * real
in expression:
    3 + 4.5
```

```
– real 3 + 4.5;
– 3 + round 4.5;
– 3 + trunc 4.5;
```

Type constraints

- Arguments must agree with parameters on types

```
– real;
val it = fn : int -> real
– real 3.4;
Error: operator and operand don't agree
    operator domain: int
    operand:         real
```

- Both branches of if expression must have the same type.

```
– if x < y then 2 else 3.4;
Error: types of if branches do not agree
    then branch: int
    else branch: real
```

Compound data types

- Tuples: $A \times B = \{ (a,b) \mid a \in A, b \in B \}$

```
– (2,3);                              (* 2-tuple *)
val it = (2,3) : int * int
– (2,3.4,"Pluto");                    (* 3-tuple *)
val it = (2,3.4,"Pluto") : int * real * string
– ((2,3.4),"Pluto");                  (* 2-tuple; *)
val it = ((2,3.4),"Pluto") : (int * real) * string
– (2);                                (* No 1-tuple *)
val it = 2 : int
– #1(2,3);  #2(2,3);                  (* #1, #2 are selectors *)
val it = 2 : int
val it = 3 : int
```

Compound data types

- Records
  - {age=20,name="Snoopy"} ;

  val it = {age=20,name="Snoopy"} : {age:int, name:string}
  - val r = it;

  val r = {age=20,name="Snoopy"} : {age:int, name:string}
  - #name r; #age r;

  val it = "Snoopy" : string

  val it = 20 : int

  - {1=2,2=3}; (* Tuples are a special case of records. *)

  val it = (2,3) : int * int

Functions

- Lambda expressions (e.g. $\lambda x.x+1$ in lambda calculus)

```
– fn x => x+1;
val it = fn : int -> int
– (fn x => x+1) 2;
val it = 3 : int
```

```
– fn x : int => x+1 : int;
val it = fn : int -> int
```

- Named functions

```
– val f = fn x => x+1;
– fun f x = x+1;
val f = fn : int -> int
– f 2*3;
val it = 9 : int
```

```
– val f : int->int = fn x => x+1
– fun f (x : int) : int = x+1;
val f = fn : int -> int
– f (2*3);
val it = 7 : int
```

Functions

- Recursive functions
  - fun f n = if n=0 then 1 else n*f(n-1);
  - val rec f = fn n => if n=0 then 1 else n*f(n-1);

    val f = fn : int -> int
  - f 5;

    val it = 120 : int
  - fun f x = x+1;

    val f = fn : int -> int
  - val f = fn n => if n=0 then 1 else n*f(n-1);

    val f = fn : int -> int
  - f 5;

    val it = 25 : int

Functions

- val rec x=x;

Error: fn expression required on rhs of val rec

- Mutual recursive functions

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

Functions

- Curried functions

  A curried function takes one argument at a time.

```
– fun f x y = x+y;                    (* λx.λy.x+y *)
– fun f x = fn y => x+y;
– val f = fn x => fn y => x+y;        (* default is int *)
val f = fn : int -> int -> int        (* -> is right associative *)
– f 2 3;                              (* application is left associative *)
val it = 5 : int;
– f 2;
val it = fn : int -> int
– fun f (x : real) y = x+y;           (* fun f x y = x+y : real; *)
val f = fn : real -> real -> real     (* fun f x y = x+(y : real); *)
```

Functions

- Uncurried functions
  - fun f (x,y) = x+y;
  - val f = fn (x,y) => x+y;

  val f = fn : int * int -> int

  In effect, this is a two-parameter function, but is interpreted in ML as an one-parameter function – the parameter being a 2-tuple.

Pattern matching

- Pattern matching: val pattern=expression;

```
– val a = 2;                        (* variable pattern *)
val a = 2 : int
– val (a,b) = (2,3);                (* constructor pattern *)
val a = 2 : int                     (* (,) is a 2-tuple constructor *)
val b = 3 : int
– val (a,_) = (2,3);                (* wildcard pattern *)
val a = 2 : int
– val x as (a,b) = (2,3);           (* layered pattern *)
val x = (2,3) : int * int
val a = 2 : int
val b = 3 : int
```

Pattern matching

- Pattern matching subsumes parameter passing
  - fun f (x,y) = x*y;
  - val f = fn (x,y) => x*y;

  val f = fn : int * int -> int

  - f (2,3) ;　　　　– f (0,3) ;　　　　(* 0*3 is redundant *)

  val it = 6 : int　　　　val it = 0 : int

  constant pattern

```
- fun f (0,_) = 0                 - val f = fn (0,_) => 0
=     | f (_,0) = 0               =            | (_,0) => 0
=     | f (x, y) = x*y;           =            | (x, y) => x*y;
val f = fn : int*int -> int
```

Pattern matching

- Pattern matching subsumes parameter passing

```
– fun f 0 = 1
      | f n = n*f(n-1);
– val rec f = fn 0 => 1
               | n => n*f(n-1);
val f = fn : int -> int

– fun f () = 5;          (* constant pattern *)
– val f = fn () => 5;
val f = fn : unit -> int
– f ();
val it = 5 : int
```

Pattern matching

- Pattern matching constraints
  - Reduandant pattern

    – fun f n = n*f(n-1) | f 0 = 1;   (* wrong order *)

    Error: match reduandant
  - Nonexhaustive pattern

    – fun f 2 = 3;

    Warning: match nonexhaustive

    – f 3;

    Error: nonexhaustive match failure
  - Nonlinear pattern

    – fun f x x = true

    Error: duplicate variable in pattern(s)

Case expression

- Case expression
  - case exp of pat_1 => exp_1 | … | pat_n => exp_n
    ≡ (fn pat_1 => exp_1 | … | pat_n => exp_n) exp
  - Example
    – fun f n = case n of 0 => 1 | n => n*f(n-1);
    – fun f n = (fn 0 =>1 | n => n*f(n-1)) n;
  - Example
    – fun f 0 y = y | f x y = 1+f (x-1) y;
    – fun f 0 = (fn y => y) | f x = fn y => 1+f (x-1) y;
    val f = fn : int -> int -> int
    – fun f 0 = fn y => y | f x = fn y => 1+f (x-1) y;
    Error: syntax error found at EQUALOP

Let expression

- Let expression
  - let *declarations* in *expression* end
  - Example

```
– val x = 2;
– let val x = 3 val y = x in x+y end;          (* sequential *)
val it = 6 : int
– let val x = 3 and y = x in x+y end;          (* simultaneous *)
val it = 5 : int
– let fun f 0 = 1 | f n = n*f(n-1) in f 10 end;
val it = 3628800 : int
– let val rec f = fn 0 => 1 | n => n*f(n-1) in f 10 end;
val it = 3628800 : int
```

Polymorphic functions

- Polymorphic function cf. Monomorphic function

```
– fun f x = 3;
val f = fn : 'a -> int

– fun f x = x+1;
val f = fn : int -> int

– fun id x = x;
– fun id (x : 'a) : 'a = x;
val id = fn : 'a -> 'a
– id 2;          (* instance: int -> int *)
– id (2,3);      (* instance: int*int -> int*int *)
– id id 2;       (* instance of 1st id: ('a -> 'a) -> 'a -> 'a *)
                 (* instance of 2nd id (or, id id): int -> int *)
```

Polymorphic functions

- Value polymorphism (or value restriction)

  To ensure the consistency of type system, SML 97 imposes the value polymorphism rule: Variables introduced by a val binding are allowed to be polymorphic only if the right-hand side is a value, i.e. needs no computation.

  – fun id x = x; (* OK *)

  – val id = fn x => x; (* OK *)

  – val iden = id id; (* NO *)

  Warning: type vars not generalized because of ....

  To satisfy the value polymorphism rule, do this:

  – val iden = fn x => id id x; (* OK *) N.B. f = g iff f x = g x $\forall$x

  – fun iden x = id id x; (* OK *)

Polymorphic functions

- Polymorphic function: an example
  - fun c f g x = f (g x);

    val c = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
  - c Math.sqrt (c real ord) #"a";
  - c (c Math.sqrt real) ord #"a";

    val it = 9.8488578018 : real
  - (Math.sqrt o real o ord) #"a";
  - op o (op o (Math.sqrt,real),ord) #"a";

    val it = 9.8488578018 : real
  - op o;

    val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

> – ord; real; Math.sqrt;
> val it = fn : char -> int
> val it = fn : int -> real
> val it = fn : real -> real

Polymorphic functions

- Equality
  - Equality is defined for many types (for some definition of "same"), but not all.
  - Types that admit equality

    integers, booleans, strings and characters

    Tuples and records admit equality if all their component types admit equality.

    N.B. Two tuples (or records) are equal if their components are equal.
  - Types that don’t admit equality

    reals, e.g. 0.3 = 0.1+0.1+0.1 ???

    functions, e.g. (fn x => x+x) = (fn x => 2*x) ??? undecidable

Polymorphic functions

- Equality
  - Type variables ''a, ''b, etc, range only over types which admit equality.
  - – op =;
    Warning: calling polyEqual
    val it = fn : ''a * ''a -> bool
    – 2.0 = 2.0;
    Error: equality type required
  - – fun mem x (y, z) = x=y orelse x=z;
    Warning: calling polyEqual
    Warning: calling polyEqual
    val mem = fn : ''a -> ''a * ''a -> bool

Type expressions

- Type expressions
  - A type expression denotes a type.
  - Type expressions are defined as
    - Type constants, e.g. int, real, bool, char, string, and unit, are type expressions
    - Type variables, e.g. 'a, ''a, etc are type expressions.
    - If $t, u, t_1, t_2, \ldots, t_n$ are type expressions, so are

      t -> u

      t list

      $t_1 * t_2 * \ldots * t_n$

      $\{label_1 = t_1,\ label_2 = t_2, \ldots,\ label_n = t_n\}$

Type expressions

- Type operators
  - Precedence

{...}	highest
list	
*	
->	lowest

  - Associativity

*	non-associative
->	right associative

  - Example

    'a * 'b * real -> 'a * int -> 'b list

    ('a * 'b * real -> 'a * int) -> 'b list

    ('a * 'b) * real -> 'a * int -> 'b list

    'a * 'b * real -> 'a * (int -> 'b list)

Type expressions

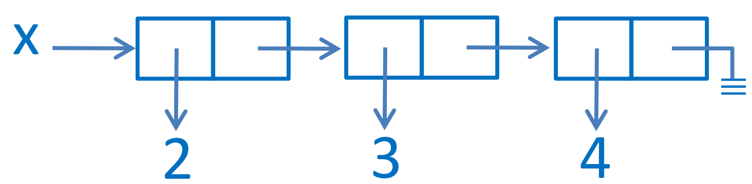
- Polymorphism and monomorphism
  - Polytype (polymorphic types)
    Types that contain type variables
  - Monotypes (monomorphic types)
    Types that don't contain type variables
  - Polymorphic functions – Functions of polytypes
    Monomorphic functions – Functions of monotypes

List processing

- Lists
  - [2,3,4];
  val it = [2,3,4] : int list
  - [];
  val it = [] : 'a list
  - val x = 2::3::4::[];
  val x = [2,3,4] : int list
  - hd x;
  val it = 2 : int
  - tl x;
  val it = [3,4] : int list

  - [2,3.4];
  Error: elements not the same type

  (* or, 2::3::[4], 2::[3,4], [2,3,4] *)
  (* List constructors: [] and :: *)

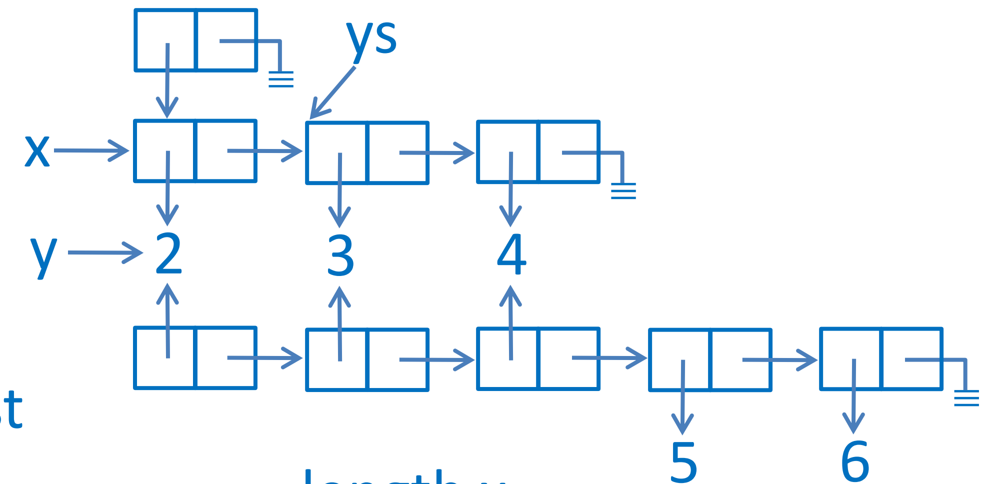


  - hd []; tl [];
  Both are erroneous.

List processing

- Lists

```
– val y :: ys = x;                     (* constructor pattern *)
Warning: binding not exhaustive        (* missing [] *)
val y = 2 : int
val ys = [3,4] : int list
– null x;
val it = false : bool
– x @ [5,6];
val it = [2,3,4,5,6] : int list
– [x];
val it = [[2,3,4]] : int list list
```




```
– length x;
val it = 3 : int
```

List processing

- Examples
  - – fun length xs = if null xs then 0 else 1+length (tl xs);

    – fun length [] = 0

    = | length (x::xs) = 1+length xs;

    val length = fn : 'a list -> int

    - length x::xs means (length x)::xs
    - – fun length xs = if xs=[] then 0 else 1+length (tl xs);

      val length = fn : ''a list -> int

      – length [2.0,3.0]; (* error *)
  - – fun member x [] = false

    = | member x (y::ys) = x=y orelse member x ys;

    val member = fn : ''a -> ''a list -> bool

List processing

- Appending lists
  – infixr 6 ++;
  infixr 6 ++
  – fun [] ++ ys = ys
  = | (x::xs) ++ ys = x :: xs ++ ys;
  val ++ = fn : 'a list * 'a list -> 'a list
  - xs ++ ys takes O(|xs|) time and space
  - xs ++ ys ++ zs
    Left-associative ++ takes O(2|xs|+|ys|) time and space.
    Right-associative ++ takes O(|xs|+|ys|) time and space.

  – [1,2] ++ [3,4,5];
  – op++([1,2],[3,4,5]);

precedence 0~9
infix, infixr, nonfix
infix 8 * / div mod
infix 7 + - ^
infixr 6 :: @
infix 5 = <> > >= < <=
infix 3 o

List processing

```
– nonfix ++;
nonfix ++
– [1,2] ++ [3,4,5];
Error : operator is not a function
– ++([1,2],[3,4,5]);
```

- ```
  – fun rev [] = []
  =     | rev (x::xs) = rev xs @ [x];
  – fun rev xs =
  =     let fun rev [] ys = ys
  =             | rev (x::xs) ys = rev xs (x::ys)
  =     in rev xs [] end;
  val rev = fn : 'a list -> 'a list
  ```
  ```

List processing

- Insertion sort

```
– fun isort [] = []
      | isort (x::xs) =
          let fun insert x [] = [x]
              (* | insert x (y::ys) = if x<=y then x::y::ys *)
                | insert x (zs as y::ys) = if x<=y then x::zs
                                           else y::insert x ys
          in insert x (isort xs)
          end;
val isort = fn : int list -> int list
```

Higher-order functions

- Higher-order functions

```
– map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
– map (fn x=>x+1) [1,2,3];
val it = [2,3,4] : int list
– map (fn (x,y)=>x+y) [(1,2),(3,4)];
val it = [3,7] : int list

– fun powerset [] = [[]]
=     | powerset (x::xs) = let val ys=powerset xs
                           in  ys @ map (fn s=>x::s) ys end;
val powerset = fn : 'a list -> 'a list list
–  powerset [1,2];
val it = [[],[2],[1],[1,2]] : int list list
```

Concrete data type

- Datatype declaration
  - datatype *typename*

    = *constructor*$_1$ [of argument-type$_1$]

    | *constructor*$_2$ [of argument-type$_2$]

    | . . .
  - Constructor constants – those without arguments

    Constructor functions – those with arguments
  - datatype declarations introduce union types.
  - Convention

    Constructor names begin with an uppercase letter.

Concrete data type

- Enumeration type
  - Enumeration types are union types with only constructor constants.
  - – datatype logic3 = True | False | Undef;
    datatype logic3 = False | True | Undef
  - In math, logic3 = {True} ∪ {False} ∪ {Undef}
    In C/C++, enum logic3 {True, False, Undef};
  -
```
– fun and3 True True = True          – and3 True Undef;
=      | and3 False _ = False         val it = Undef : logic3
=      | and3 _ False = False
=      | and3 _ _ = Undef;
val and3 = fn : logic3 -> logic3 -> logic3
```

Concrete data type

- Union type
  - Union types, except for enumeration types, have at least one constructor function.
  - – datatype length = Meter of int | Millimeter of int;
  - In math, length = { Meter x | x∈int } ∪ { Millimeter x | x∈int }
    In C/C++, struct length {

```
enum { Meter, Milimeter } m;
union { int me; int mi; } x;
};
```

  - Representation of union values

$\boxed{\text{Meter 3}}$ $\boxed{\text{Millimeter 5}}$

Concrete data type

- Union type
  - – datatype volume = SqMeter of int | SqMillimeter of int;
    – fun area (Meter x) (Meter y) = SqMeter (x*y)
    = | area (Meter x) (Millimeter y) = SqMillimeter (100*x*y)
    = | area (Millimeter x) (Meter y) = SqMillimeter (100*x*y)
    = | area (Millimeter x) (Millimeter y) = SqMillimeter (x*y);
    val area = fn : length -> length -> volume
    – area (Meter 3) (Millimeter 5);
    val it = SqMillimeter 1500 : volume

Concrete data type

- Recursive data type
  - – datatype btree = Empty | Node of int*btree*btree;

```
– fun bst [] = Empty
=     | bst (x::xs) =
=         let
=            fun insert x Empty = Node(x,Empty,Empty)
=               | insert x (Node(y,l,r))
=                 = if x < y then Node(y,insert x l,r)
=                               else Node(y,l,insert x r)
=         in insert x (bst xs) end;
val bst = fn : int list -> btree
```

Concrete data type

- Recursive data type
  - – bst [3,1,2];

    val it = Node(2,Node(1,Empty,Empty), Node(3,Empty, Empty)) : btree

    – fun inorder Empty = []

    = | inorder (Node(n,l,r)) = inorder l @ [n] @ inorder r;

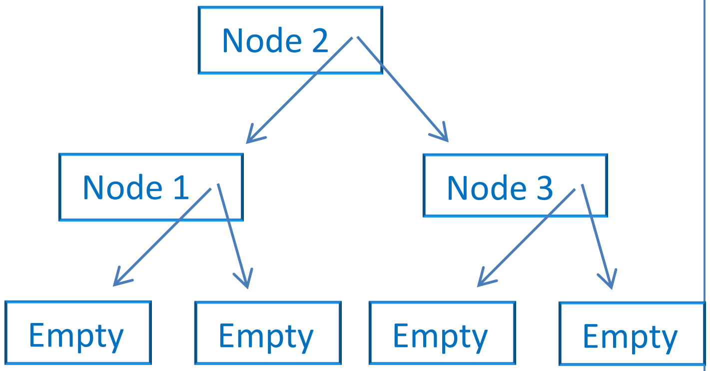
    val inorder = fn : btree -> int list

    – val sort = inorder o bst;

    val sort = fn : int list -> int list

    – sort [8,5,7,3,6,1,4,9,2];

    val it = [1,2,3,4,5,6,7,8,9] : int list



Reference

- Reference and assignment
  - ```
    - val x = ref 2;          (* ref is a constructor function *)
    val x = ref 2 : int ref
    - x;
    val it = ref 2 : int ref
    - !x;                     x  ref 2 3 7
    val it = 2 : int
    - x := !x+1;
    val it = () : unit
    - !x;                     - x := 7; !x;   (* sequence: ; is a must *)
    val it = 3 : int          val it = 7 : int
    ```
    ```

Reference

- Reference and assignment
  -
    ```
    - fun f n = let val r = ref 1
    =                 fun loop 0 = !r
    =                    | loop n = (r := !r*n; loop (n-1))
    =             in
    =                 loop n
    =             end;
    - f 10;
    val it = 3628800 : int
    ```