Programming Language Concepts

Standard ML Tutorial[1]

[1] Adapted from slides and notes by John Reppy & Matthias Blume and Dan Grossman

What is Standard ML?

SML is a general-purpose functional programming language with

- strict evaluation
- strong and static typing
- polymorphic types
- type inference
- datatypes and pattern matching
- functional impurities (mutable objects, side-effects, exceptions)
- a sophisticated module system
- a rigorous formal definition

What is Standard ML?

SML is a general-purpose functional programming language with

- strict evaluation
- strong and static typing
- polymorphic types
- type inference
- datatypes and pattern matching
- functional impurities (mutable objects, side-effects, exceptions)
- a sophisticated module system
-

History of Standard ML

- 1978: ML (*meta language*)
  - designed and implemented by Robin Milner et. al.
  - a programming language for finding and performing proofs in a formal logical system (LCF)
  - features to support writing proof tactics: higher-order functions, polymorphic types, exceptions
- 1978: Hindley-Milner, Damas-Milner type inference
  - a.k.a., polymorphic type checking or Algorithm W
  - automatically determine the (most general) types of variables

```
fun map f l =
  case l of nil => nil
          | (h::t) => (f h)::(map f t)
(*
val map :
  ('a -> 'b) -> 'a list -> 'b list
*)
```

History of Standard ML

- 1980: HOPE
  - designed and implemented by Rod Burstal et. al.
  - pattern matching, early module systems
- 1981: MacQueen modules
  - parametric module system for HOPE, inspired by CLEAR's parameterized specifications
  - extended with novel method of specifying sharing of components among the structure parameters of a functor
- 1983: Standard ML
  - Robert Harper, David MacQueen, Robin Milner
  - ML polymorphism, HOPE patterns, Cardelli records, Mycroft et. al. exceptions (generalizing ML exceptions), MacQueen modules

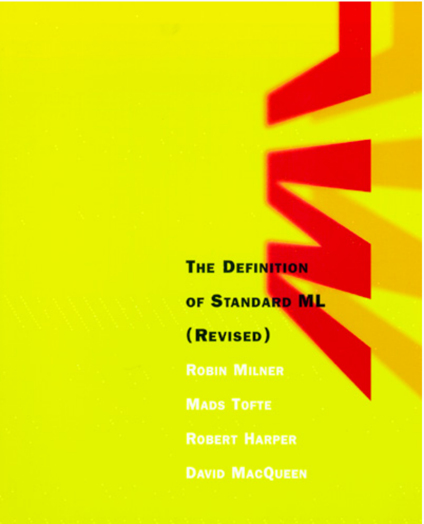
History of Standard ML

- 1990: The Definition of Standard ML
  - Robin Milner, Mads Tofte, and Robert Harper
  - "A precise description of a programming language is a prerequisite for its implemention and for its use."
  - formalization of syntax, static semantics, and dynamic semantics
- 1991: Commentary on Standard ML
  - Robin Milner and Mads Tofte

History of Standard ML

- 1997: The Definition of Standard ML (Revised)
  - Robin Milner, Mads Tofte, Robert Harper, and David MacQueen
  - "A precise description of a programming language is a prerequisite for its implemention and for its use."
  - formalization of syntax, static semantics, and dynamic semantics
  - $< \mathbf{120}$ pages (incl. contents, appendicies, bibliography, index)

The Definition of Standard ML (Revised) — SML'97



The Definition of Standard ML (Revised) — SML'97

Match Rules $\boxed{C \vdash mrule \Rightarrow \tau}$

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C + VE \vdash exp \Rightarrow \tau' \qquad \mathrm{tynames}\, VE \subseteq T \text{ of } C}{C \vdash pat \texttt{ => } exp \Rightarrow \tau \to \tau'} \tag{14}$$

Comment: This rule allows new free type variables to enter the context. These new type variables will be chosen, in effect, during the elaboration of *pat* (i.e., in the inference of the first hypothesis). In particular, their choice may have to be made to agree with type variables present in any explicit type expression occurring within *exp* (see rule 9).

Declarations $\boxed{C \vdash dec \Rightarrow E}$

$$\frac{\begin{array}{l} U = \mathrm{tyvars}(tyvarseq) \\ C + U \vdash valbind \Rightarrow VE \qquad VE' = \mathrm{Clos}_{C,valbind} VE \qquad U \cap \mathrm{tyvars}\, VE' = \emptyset \end{array}}{C \vdash \texttt{val}\ tyvarseq\ valbind \Rightarrow VE' \text{ in Env}} \tag{15}$$

$$\frac{C \vdash typbind \Rightarrow TE}{C \vdash \texttt{type}\ typbind \Rightarrow TE \text{ in Env}} \tag{16}$$

$$\frac{\begin{array}{c} C \oplus TE \vdash datbind \Rightarrow VE, TE \qquad \forall (t, VE') \in \mathrm{Ran}\, TE,\ t \notin (T \text{ of } C) \\ TE \text{ maximises equality} \end{array}}{C \vdash \texttt{datatype}\ datbind \Rightarrow (VE, TE) \text{ in Env}} \tag{17}$$

$$\frac{C(longtycon) = (\theta, VE) \qquad TE = \{tycon \mapsto (\theta, VE)\}}{C \vdash \texttt{datatype}\ tycon \texttt{ = datatype}\ longtycon \Rightarrow (VE, TE) \text{ in Env}} \tag{18}$$

$$\frac{\begin{array}{ll} C \oplus TE \vdash datbind \Rightarrow VE, TE & \forall (t, VE') \in \mathrm{Ran}\, TE,\ t \notin (T \text{ of } C) \\ C \oplus (VE, TE) \vdash dec \Rightarrow E & TE \text{ maximises equality} \end{array}}{C \vdash \texttt{abstype}\ datbind\ \texttt{with}\ dec\ \texttt{end} \Rightarrow \mathrm{Abs}(TE, E)} \tag{19}$$

$$\frac{C \vdash exbind \Rightarrow VE}{C \vdash \texttt{exception}\ exbind \Rightarrow VE \text{ in Env}} \tag{20}$$

$$\frac{C \vdash dec_1 \Rightarrow E_1 \qquad C \oplus E_1 \vdash dec_2 \Rightarrow E_2}{C \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \Rightarrow E_2} \tag{21}$$

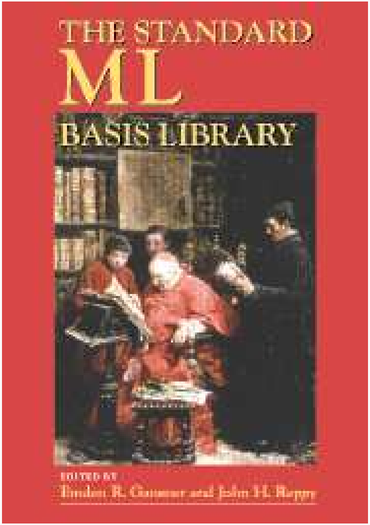
$$\frac{C(longstrid_1) = E_1 \quad \cdots \quad C(longstrid_n) = E_n}{C \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_n} \tag{22}$$

$$\frac{}{C \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{23}$$

History of Standard ML

- 2004: The Standard ML Basis Library
  - Emden Gasner and John Reppy (eds)
  - "the fundamentals: primitive types such as integers and floating-point numbers, operations requiring runtime system or compiler support, such as I/O and arrays; and ubiquitous utility types such as booleans and lists. ... does not cover higher-level types, such as collection types, or application-oriented APIs, such as regular expression matching."

The Standard ML Basis Library



http://www.standardml.org/Basis/

The Standard ML Basis Library

11.38 The PACK_WORD signature

The `PackWordNBig` and `PackWordNLittle` structures provide facilities for packing and unpacking N-bit word elements into `Word8` vectors. This mechanism allows word values to be transmitted in binary format over networks. The `PackWordNBig` structures perform big-endian packing and unpacking, while the `PackWordNLittle` structures perform little-endian packing and unpacking.

Synopsis

```
signature PACK_WORD
structure PackWordNBig :> PACK_WORD
structure PackWordNLittle :> PACK_WORD
```

Interface

```
val bytesPerElem : int
val isBigEndian : bool
val subVec  : Word8Vector.vector * int -> LargeWord.word
val subVecX : Word8Vector.vector * int -> LargeWord.word
val subArr  : Word8Array.array * int -> LargeWord.word
val subArrX : Word8Array.array * int -> LargeWord.word
val update : Word8Array.array * int * LargeWord.word
               -> unit
```

Description

```
val bytesPerElem : int
```

The number of bytes per element. Most implementations will provide several structures with values of `bytesPerElem` that are small powers of two (e.g., 1, 2, 4, and 8, corresponding to N of 8, 16, 32, 64, respectively).

```
val isBigEndian : bool
```

True if the structure implements a big-endian view of the data (most-significant byte first). Otherwise, the structure implements a little-endian view (least-significant byte first).

```
val subVec : Word8Vector.vector * int -> LargeWord.word
val subVecX : Word8Vector.vector * int -> LargeWord.word
```

`subVec (vec, i)`
`subVecX (vec, i)`
These extract the subvector

`vec[bytesPerElem*i..bytesPerElem*(i+1)-1]`

of the vector *vec* and convert it into a word according to the endianness of the structure. The `subVecX` version extends the sign bit (most significant bit) when converting the subvector to a word. The functions raise the `Subscript` exception if $i < 0$ or if $|vec| <$ `bytesPerElem` $* (i + 1)$.

http://www.standardml.org/Basis/

History of Standard ML

- 2007: Defects in the Revised Definition of Standard ML
  - Andreas Rossberg
  - 14 pages

SML Implementations

- Standard ML of New Jersey
  - `http://www.smlnj.org`
  - continuation-passing style; incremental compilation and REPL
- MLton
  - `http://www.mlton.org`
  - whole-program optimization
- Poly/ML
  - `http://www.polyml.org`
  - very fast compilation; REPL
- ML Kit
  - `http://www.itu.dk/research/mlkit/index.php/Main_Page`
  - region-based memory managment
- HaMLet
  - `http://www.mpi-sws.org/~rossberg/hamlet/`
  - reference interpreter (written in SML, following the Definition)
-
  - `http://www.itu.dk/people/sestoft/mosml.html`

Using the MoscowML

- interactive REPL (read-eval-print-loop)
  - Type `mosml` to run the MoscowML interactive REPL
  - `Ctrl-d` exits the REPL; `Ctrl-c` interrupts execution.
  - Some ways to run ML programs:
    - type in code in the interactive read-eval-print loop

      ```
      - 1 + 1;
      ```

    - load ML code from a file (e.g., `foo.sml`)

      ```
      - use "foo.sml";
      ```

- batch compiler
  - Type `mosmlc` to run the MoscowML batch compiler

Hello, World!

```
(* first program *)
val x = print "Hello, World!\n"
```

- A *program* is a sequence of *bindings*
- One kind of binding is a *variable binding*
- Execution evaluates bindings in order
- To evaluate a variable binding:
  - Evaluate the expression (to the right of =) in the environment created by the *previous* bindings.
  - This produces a value.
  - Extend the (top-level) environment, binding the variable to the value.

Theory Break

Some terminology and pedantry:

- Expressions are *evaluated* in an environment
- An *environment* maps variables to values
- Expressions are *type-checked* in a context
- A *context* maps variables to types

- *Values* are integers, strings, function-closures, ...
  - (“things already evaluated”)
- Expressions have evaluation rules and type-checking rules

Simple expressions

- Integers: `3`, `54`, `~3`, `~54`
- Reals[2]: `3.0`, `3.14159`, `~3.6E00`
- Booleans: `true`, `false`, `not`
- Strings: `"abc"`, `"hello world\n"`, `x ^ ".sml"`
- Chars: `#"a"`, `#"\n"`,
- Overloaded operators: `+`, `-`, `*`, `<`, `<=`
- Lists: `[]`, `[1,2,3]`, `["x","sml"]`, `1::2::nil`
- Tuples: `()`, `(1,true)`, `(3,"abc",false)`
- Records: `{a=1,b=true}`, `{name="bob",age=8}`
- conditionals, functions, function applications

[2] floating-point numbers

Value Declarations

Binding a value to a variable.

- syntax

  val *var* = *exp*

- examples

```
val x = 3

val y = x + 1

val z = y - x
```

Thus, variables are identifiers that *name* values.

Once a binding for a variable is established,
the variable names the *same* value until it goes out of scope.

Standard ML variables are *immutable*.

Function Declarations

Binding a function (which is a value) to a variable.

- syntax (simplified)

$$\texttt{fun}\ \mathit{var_f}\ \mathit{var_a} = \mathit{exp}$$

- examples

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

fun fact2_loop (n, f) =
  if n = 0 then f
  else fact2_loop (n - 1, n * f)

fun fact2 n = fact2_loop (n, 1)
```

Let expressions

Limit the scope of variables from declarations.

- syntax

  let *decl* **in** *exp* **end**

- example

```
let
  val x = let val y = 1
          in y + y
          end
  fun f z = (z, x * z)
in
  f (4 + x)
end
```

Function expressions

Introduce a function from one argument to one result.
Such an *anonymous* function has no name, but is a value,
so it can be bound to a variable.

- syntax (simplified)

  fn *var* => *exp*

- example

```
val double = fn z => 2.0 * z

val inc = fn x => x + 1
```

The last is equivalent to

```
fun inc x = x + 1
```

Function expressions (cont.)

Because functions are *first-class*,
one function can return another function as a result.

- example

  ```
  val add = fn x => fn y => x + y
  val inc = add 1  (* == fn y => 1 + y *)
  val three = inc 2
  ```

  The first is equivalent to

  ```
  fun add x y = x + y
  ```

This is one "solution" to functions taking multiple arguments;
such functions are called *curried* functions.

Another "solution" is to take a value that is a data structure
containing multiple values.

Tuple and record expressions

Create collections of values.

- tuples, syntax

$$(\mathit{exp}_1 , \ldots , \mathit{exp}_n)$$

- tuples, examples

```
val x = ("foo", 1.0 / 2.0, false)
val y = (x, x)
```

- records, syntax

$$\{ \mathit{lab}_1 = \mathit{exp}_1 , \ldots , \mathit{lab}_n = \mathit{exp}_n \}$$

- records, examples

```
val car = {make = "Toyota", year = 2001}
```

List expressions

Finite sequences of values.

- syntax

$$\texttt{nil} \qquad \mathit{exp}_x \;\texttt{::}\; \mathit{exp}_l$$

$$\texttt{[}\; \mathit{exp}_1 \;\texttt{,}\; \ldots \;\texttt{,}\; \mathit{exp}_n \;\texttt{]}$$

- examples

```
val l0 = nil
val l1 = 1.0 :: 2.0 :: 3.0 :: nil
val l2 = [1.0, 2.0, 3.0]
val l3 = 1.0 :: 2.0 :: [3.0]
```

All of `l1`, `l2`, and `l3` are equivalent.

Patterns

Decompose compound values;
commonly used in value bindings and function arguments.

- revized syntax for declarations and function expressions

  val *pat* = *exp*   **fun** var_f pat_a = *exp*

  fn *pat* => *exp*

- variable patterns

```
val z = 3
val pair = (z, true)
```

  ⇒ `z` = 3, `pair` = `(3, true)`

- tuple and record patterns

```
val (x,y) = pair
```

  ⇒ `x` = 3, `y` = `true`

```
val {make=mk, year=yr} = car
```

Patterns (cont.)

- wildcard patterns

```
val _ = 4 * 3 * 2 * 1
```

$\Rightarrow$

- constant patterns

```
val 3 = 1 + 2
val true = 1 < 3
```

- constructor patterns

```
val l = [1,2,3]
val fst::rest = l
val [x,_,z] = l
```

$\Rightarrow$ `fst` = 1, `rest` = `[2,3]`, `x` = 1, `z` = 3

Patterns (cont.)

- nested patterns

```
val ((x,y),z) = ((1,2),3)
val (a,b)::_ = [(3.0,true),(5.0,false)]
```

$\Rightarrow$ `x` = 1, `y` = 2, `z` = 3

$\Rightarrow$ `a` = `3.0`, `b` = `true`

- as patterns

```
val l as (a,b)::_ = [(3.0,true),(5.0,false)]
val t as (p as (x,y),z) = ((1,2),3)
```

$\Rightarrow$ `l` = `[(3.0,true),(5.0,false)]`,

$\Rightarrow$ `a` = `3.0`, `b` = `true`,

$\Rightarrow$ `t` = `((1,2),3)`, `p` = `(1,2)`, `x` = 1,

$\Rightarrow$ `y` = 2, `z` = 3

Pattern matching

What to do when there is more than one way to decompose a value?
Use *pattern matching* to consider each possible way.

- match rule, syntax

$$\mathit{pat} \texttt{ => } \mathit{exp}$$

- match, syntax

$$\mathit{pat}_1 \texttt{ => } \mathit{exp}_1 \texttt{ | } \cdots \texttt{ | } \mathit{pat}_n \texttt{ => } \mathit{exp}_n$$

  When a match is applied to a value *value*,
  we try the rules from left to right,
  looking for the first rule whose pattern matches *value*.
  We then bind the variables in the pattern and evaluate the expression.

Pattern matching (cont.)

Pattern matching is used in a number of expression and declaration forms.

- case expression, syntax

  $$\texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{match}$$

- function expression, syntax

  $$\texttt{fn}\ \mathit{match}$$

- clausal function declaration, syntax

  $$\texttt{fun}\ \mathit{var_f}\ \mathit{pat_1}\ \texttt{=}\ \mathit{exp_1}\ \texttt{|}\ \cdots\ \texttt{|}\ \mathit{var_f}\ \mathit{pat_n}\ \texttt{=}\ \mathit{exp_n}$$

  The function name ($\mathit{var_f}$) is the same in all branches.

Pattern matching examples

```
fun length l =
  case l of [] => 0
          | _ :: r => 1 + length r

fun length [] = 0
  | length (_ :: r) = 1 + length r


val isZero = fn 0 => true | _ => false


fun even 0 = true
  | even n = odd (n - 1)
and odd 0 = false
  | odd n = even (n - 1)
```

Types

Every expression has a *type*.

- primitive types: `int`, `string`, `bool`

```
3 : int      true : bool      "abc" : string
```

- function types: ty_1 `->` ty_2

```
even : int -> bool
```

- product types: $ty_1 * \cdots * ty_n$, `unit`

```
(3, true) : int * bool      () : unit
```

- record types: { lab_1: ty_1 , $\cdots$, lab_n: ty_n }

```
car : {make: string, year: int}
```

- type operators: ty `list` (for example)

```
[1,2,3] : int list
```

Type abbreviations

Introduce a new name for a type.

- syntax

$$\texttt{type}\ tycon = ty$$

- examples

```
type point = real * real
type line = point * point
type car = {make: string, year: int}
```

- syntax

$$\texttt{type}\ tyvar\ tycon = ty$$

- examples

```
type 'a pair = 'a * 'a
type point = real pair
```

Datatypes

Algebraic datatypes are one of the most useful and convenient features
of Standard ML (and other functional programming languages).

They introduce a (brand) new type that is a *tagged union*
of some number of variant types.

- syntax

  $$\texttt{datatype}\ \mathit{tycon} = \mathit{con}_1\ \texttt{of}\ \mathit{ty}_1 \mid \cdots \mid \mathit{con}_n\ \texttt{of}\ \mathit{ty}_n$$

- example

```
datatype color = Red | Green | Blue
datatype shape =
    Circle of color * real
  | Rectangle of color * real * real
```

Datatypes (cont.)

The data constructors can be used both
in expressions to create values of the new type and
in patterns to discriminate variants and to decompose values.

- example

```
fun area s =
  case s of
       Circle (_, r) = Math.pi * r * r
     | Rectangle (_, l1, l2) => l1 * l2

val c = Circle (Red, 2.0)

val a = area c
```

Datatypes can be *recursive*.

- example

Datatype example

```
datatype int_btree = Leaf
                   | Node of int_btree * int * int_btree

fun depth t =
  case t of
     Leaf => 0
   | Node (l, _, r) => 1 + max (depth l, depth r)

fun insert t i =
  case t of
     Leaf => Node (Leaf, i, Leaf)
   | Node (l,j,r) =>
       if i=j then t
       else if i < j
               then Node(insert l i,j,r)
               else Node(l,j,insert r i)
```

Datatype example

```
datatype int_btree = Leaf
                   | Node of int_btree * int * int_btree

(* in-order traversal of trees *)
fun inttreeToList t =
  case t of
     Leaf => []
   | Node (l, i, r) =>
       (inttreeToList l) @ [i] @ (inttreeToList r)
```

Representing programs as datatypes

```
type var = string

datatype exp  = Var of var            (* x *)
              | Num of int            (* 1 *)
              | Plus of exp * exp     (* e1 + e2 *)
              | Times of exp * exp    (* e1 * e2 *)
datatype stmt = Seq of stmt * stmt    (* s1 ; s2 *)
              | Assign of var * exp   (* x := e *)
              | Print of exp list     (* print (e1,...) *)

val prog =
  Seq (Assign ("a", Plus (Num 5, Num 3)),
       Print [Var "a"])
(* a := 5 + 5 ; print (a) *)
```

Computing properties of programs: size

```
fun sizeE (Var _) = 1
  | sizeE (Num _) = 1
  | sizeE (Plus (e1, e2)) = sizeE e1 + sizeE e2 + 1
  | sizeE (Times (e1, e2)) = sizeE e1 + sizeE e2 + 1

fun sizeEL [] = 0
  | sizeEL (e::es) = sizeE e + sizeEL es

fun sizeS (Seq (s1,s2)) = sizeS s1 + sizeS s2 + 1
  | sizeS (Assign (_,e)) = 2 + sizeE e
  | sizeS (Print es) = 1 + sizeEL es
```

sizeS prog $\Rightarrow$ 8

Type inference

When defining values (including functions),
types do not need to be declared
— they will be *inferred* by the compiler:

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

```
- fun isPos n = n > 0
val isPos = fn : int -> bool
```

Any inconsistencies will be detected as type errors.

```
- if 1 < 2 then 3 else "four";
stdIn:1.1-1.25 Error: types of if branches do not agre
  then branch: int
  else branch: string
  in expression:
```

Type inference (cont.)

Type inference works with *all* types in the language.

```
- fun area (Circle (_,r)) = Math.pi * r * r
=   | area (Rectangle (_,l1,l2)) = l1 * l2;
val area = fn : shape -> real
```

Overloaded operators default to `int`;
use type annotations (called *ascriptions*) to be explicit.

```
- fun add (x, y) = x + y;
val add = fn : int * int -> int
- fun addR (x: real, y) = x + y;
val addR = fn : real * real -> real
```

Polymorphic type inference

Type inference produces the *most general* type, which may be *polymorphic*.

```
- fun ident x = x;
val ident = fn : 'a -> 'a
- fun pair x = (x, x);
val pair = fn : 'a -> 'a * 'a
- val fst = fn (x, y) => x
val fst = fn : 'a * 'b -> 'a
- val foo = pair 4.0;
val foo = (4.0,4.0) : real * real
```

`pair` was used at the type `real -> real * real`.

```
- val z = fst foo;
val z = 4.0 : real
```

`fst` was used at the type `real * real -> real`.

Polymorphic datatypes

```
datatype 'a btree = Leaf
                  | Node of 'a btree * 'a * 'a btree

fun depth t =
  case t of
     Leaf => 0
   | Node (l, _, r) => 1 + max (depth l, depth r)
val depth = fn : 'a btree -> int

fun btreeToList t =
  case t of
     Leaf => []
   | Node (l, x, r) =>
      (btreeToList l) @ [x] @ (btreeToList r)
val btreeToList = fn : 'a btree -> 'a list

fun btreeMap f Leaf = Leaf
  | btreeMap f (Node (l, x, r)) =
```

Closure idioms

Closure: Function plus environment where function was defined

- Environment matters when function has free variables

1. Create similar functions
2. Combine functions
3. Pass functions with private data to iterators
4. Provide an abstract data type
5. Currying and partial application

Create similar functions

```
fun addn m n = m + n

val add_one = addn 1

val add_two = addn 2

fun mkAddList m =
  if m = 0
    then []
    else (addn m)::(mkAddList (m-1))

val lst65432 = map (fn add => add 1) (mkAddList 5)
```

Combine functions

```
fun f1 g h = (fn x => g (h x))  (* compose *)

datatype 'a option = NONE | SOME of 'a  (* predefined *)

fun f2 g h x =
  case g x of
      NONE => h x
    | SOME y => y

val printInt = f1 print Int.toString

fun truncate1 lim f = f1 (fn x => Real.min (lim, x)) f
```

Private data for iterators

```
fun map f lst =
  case lst of
     [] => []
   | h::t => (f h) :: (map f t)

fun incr lst = map (fn x => x+1) lst
val incr = map (fn x => x + 1)

fun mul i lst = map (fn x => x * i) lst
fun mul i = map (fn x => x * i)
```

A more powerful iterator

```
fun foldl f acc lst =
  case lst of
     [] => acc
   | h::t => foldl f (f (h, acc)) t

val f1 = foldl (fn (x, y) => x + y) 0
val f2 = foldl (fn (x, y) => y andalso x > 0) true

fun f3 lo hi lst =
  foldl (fn (x, y) => if x>lo andalso x<hi
                      then y+1 else y)
        0
        lst
```

Thoughts on fold

- Functions like `foldl` decouple recursive traversal (“walking”) from data processing
- No unecessary type restrictions
- Similar to visitor pattern in OOP
  - Private fields of visitor like free variables

Provide an ADT

This is difficult stuff.

```
datatype intset = ISET of { add : int -> intset,
                            member : int -> bool}

val empty_set =
  let
    fun exists (lst: int list) j =
      let fun iter rest =
            case rest of
               [] => false
             | h::t => j=h orelse iter t
      in  iter lst
      end
    fun make_set lst =
      ISET {add = fn i => (make_set(i::lst)),
            member = exists lst }
  in
```

Thoughts on ADT example

- By "hiding the list" behind the functions, we know clients do not assume anything about the representation
- Why? All you can do with a function is apply it
  - No other primitives on functions
  - No reflection
  - No aspects
  - ...

Currying

- We've been using currying and partial application a lot
  - Efficient and convenient in SML
    - (efficiency depends upon compiler; most are very good)

- Remember: the semantics is to build closures.

```
val f = fn x => (fn y => (fn z => ...))
val a = ((f 1) 2) 3
```

Exceptions

```
- 5 div 0;  (* primitive failure *)
uncaught exception Div

exception NotFound of string  (* declare new exception *)
type 'a dict = (string * 'a) list
fun lookup (s, nil) = raise (NotFound s)
  | lookup (s, (k,v)::rest) =
      if s = k then v else lookup (s, rest)
val lookup : string * 'a dict -> 'a

val d = [("foo",2), ("bar",~1)]
val d : (string * int) list  (* == int dict *)

val x = lookup ("foo", d)
val x = 2 : int

val y = lookup ("baz", d)
uncaught exception NotFound

val y = lookup ("baz", d) handle NotFound s =>
        (print ("NotFound: " ^ s ^ "\n") ; 0)
NotFound: baz
val y = 0 : int
```

References and Assignments

Although SML variables are immutable,
SML provides a type of mutable cells.

```
type 'a ref
val ref : 'a -> 'a ref
val ! : 'a ref -> 'a
val := : 'a ref * 'a -> unit
```

```
- val lineNum = ref 0;  (* create mutable cell *)
val lineNum = ref 0 : int ref

- fun lineCount () = !lineNum;  (* access mutable cell *)
fun lineCount = fn : unit -> int

- fun newLine () = lineNum := !lineNum + 1;  (* increment the cell *)
fun newLine = fn : unit -> unit

- val lineNum = ref 0;  (* create mutable cell *)
val lineNum = ref 0 : int ref
```

References and Assignments (cont.)

SML variables are immutable:

```
local
  val x = 1
in
  fun new1 () = let val x = x + 1 in x end
end
```

new1 always returns 2.

SML references are mutable:

```
local
  val x = ref 1
in
  fun new2 () = (x := !x + 1; !x)
end
```

Standard ML = Core Language + Module Language

SML is made up of two sub-languages

- core language:
  - expressing types and computations
- module language
  - packaging elements of core language into units for modularity and reuse

The module language is a *language*:
it has non-trivial static and dynamic semantics.

It is not simply a namespace management veneer.

Standard ML: Module Language

- Structures
  - an encapsulated, named, collection of (type and value) declarations
- Signatures
  - an encapsulated, named, collection of specifications
  - classify structures
- Functors
  - an encapsulated, named, function from structures to structures

To a rough approximation, the Standard ML module language is a first-order language[3] with no conditionals or recursion.

- *not* Turing complete
- evaluate module language progam at compile-time (MLton, MLKit)

[3] Proposals for higher-order functors; still strongly normalizing.

Structures

A structure collects type and value declarations into a nameable module.

```
structure UniqueId = struct
  type id = int
  val ctr = ref 0
  fun new() = let
    val i = !ctr
    val () = ctr := i + 1
  in
    i
  end
  fun toString i = "id" ^ (Int.toString i)
  fun compare (i1, i2) = Int.compare (i1, i2)
end
```

Structures

Access structure components via *dot* notation:

```
val a = UniqueId.new ()
val b = UniqueId.new ()
val aStr = UniqueId.toString a
val bStr = UniqueId.toString b
```

Structures

A structure collects type and value *and structure* declarations into a nameable module.

```
structure UniqueId = struct
  structure Counter = struct
    type ctr = int ref
    fun new() = ref 0
    fun next(ctr) = let
      val i = !ctr
      val () = ctr := i
    in
      i
    end
  end
  type id = int
  val ctr = Counter.new ()
  fun new() = Counter.next ctr
  fun toString i = "id" ^ (Int.toString i)
  fun compare (i1, i2) = Int.compare (i1, i2)
```

Signatures

A signature is the "type" of a structure:

- specification of types in structure
- type of values in structure
- signature of sub-structures in structure

```
signature UNIQUE_ID = sig
  type id = int
  val ctr : int ref
  val new : unit -> id
  val toString : id -> string
  val compare : id * id -> order
end
```

Signatures

A signature is the "type" of a structure:

- specification of types in structure
- type of values in structure
- signature of sub-structures in structure

```
signature UNIQUE_ID = sig
  structure Counter : sig
    type ctr = int ref
    val new : unit -> ctr
    val next : ctr -> int
  end
  type id = int
  val ctr : Counter.ctr
  val new : unit -> id
  val toString : id -> string
  val compare : id * id -> order
end
```

Signatures

A signature is the "type" of a structure:

- specification of types in structure
- type of values in structure
- signature of sub-structures in structure

```
signature UNIQUE_ID = sig
  structure Counter : sig
    type ctr = int ref
    val new : unit -> ctr
    val next : ctr -> int
  end
  type id = int
  val ctr : Counter.ctr
  val new : unit -> id
  val toString : id -> string
  val compare : id * id -> order
end
```

Signatures

A signature is the “type” of a structure:

- specification of types in structure
- type of values in structure
- signature of sub-structures in structure

```
signature UNIQUE_ID = sig
  type id
  val new : unit -> id
  val toString : id -> string
  val compare : id * id -> order
end
```

Signature matching

A structure matches a signature if every specification in the signature
is satisfied by a component of the structure.
After matching, only specifications in the signature
are available in the structure.

```
signature UNIQUE_ID = sig
  type id
  val new : unit -> id
  val toString : id -> string
  val compare : id * id -> order
end
structure UniqueID : UNIQUE_ID = struct
  ...
end
val ctr = UniqueId.ctr  (* ERROR *)
```

Transparent signature matching

A transparent signature match (`:`) *reveals* the implementation of types,
even if their implementation is not specified in the signature.

```
signature UNIQUE_ID = sig
  type id
  val new : unit -> id
  val toString : id -> string
  val compare : id * id -> order
end
structure UniqueID : UNIQUE_ID = struct
  ...
end
val aId = UniqueID.new ()
val z = aId + aId
```

`UniqueId.id` is considered equivalent to `int`.

Opaque signature matching

An opaque signature match (`:>`) *does not reveal* the implementation.

```
signature UNIQUE_ID = sig
  type id
  val new : unit -> id
  val toString : id -> string
  val compare : id * id -> order
end
structure UniqueID :> UNIQUE_ID = struct
  ...
end
val aId = UniqueID.new ()
val z = aId + aId  (* ERROR *)
```

`UniqueId.id` is considered a new type, distinct from *all* other types (including `int`).

Functors

A functor parameterizes a structure with respect to an input signature.

```
functor TestUniqueId(structure UId : UNIQUE_ID) = struct
  val aId = UId.new ()
  val bId = UId.new ()
  val cId = UId.new ()
  val result =
    (UId.compare (aId, aId) = EQUAL) andalso
    (UId.compare (bId, bId) = EQUAL) andalso
    (UId.compare (cId, cId) = EQUAL) andalso
    (UId.compare (aId, bId) <> EQUAL) andalso
    (UId.compare (bId, aId) <> EQUAL) andalso
    (UId.compare (aId, cId) <> EQUAL) andalso
    (UId.compare (cId, aId) <> EQUAL) andalso
    (UId.compare (bId, cId) <> EQUAL) andalso
    (UId.compare (cId, bId) <> EQUAL)
end
```

Functors

A functor parameterizes a structure with respect to an input signature.

```
signature ORDER = sig
  type t
  val compare : t * t -> order
end
signature DICTIONARY = sig
  type key
  type 'a t
  exception DictExn
  val empty : 'a t
  val lookup : 'a t * key -> 'a t
  val insert : 'a t * key * 'a -> 'a t
  val update : 'a t * key * 'a -> 'a t
end
functor ListDictionary(struct Key: ORDER)
```

Functors

A functor parameterizes a structure with respect to an input signature.

```
signature ORDER = sig
  type t
  val compare : t * t -> order
end
signature DICTIONARY = sig
  type key
  type 'a t
  exception DictExn
  val empty : 'a t
  val lookup : 'a t * key -> 'a t
  val insert : 'a t * key * 'a -> 'a t
  val update : 'a t * key * 'a -> 'a t
end
functor BTreeDictionary(struct Key: ORDER)
```

Functors

A functor parameterizes a structure with respect to an input signature.

```
signature ORDER = sig
  type t
  val compare : t * t -> order
end
signature DICTIONARY = sig
  type key
  type 'a t
  exception DictExn
  val empty : 'a t
  val lookup : 'a t * key -> 'a t
  val insert : 'a t * key * 'a -> 'a t
  val update : 'a t * key * 'a -> 'a t
end
functor RBTreeDictionary(struct Key: ORDER)
```

Functors

A functor parameterizes a structure with respect to an input signature.
Sophisticated type refinement machinery to express relationships between types in input signature and output structure.

```
functor RBTreeDictionary(struct Key: ORDER)
        :> DICTIONARY where type key = Key.t = struct
...
end
```

A dictionary is an abstract type,
so want to hide the implementation of `'a t`
using an opaque signature constraint.
But, that would also hide the implementation of `key`,
making the resulting structure unusable.
We add a constraint to the output signature

Functors

Fully-functorial programming

- code almost entirely with functors
- functors and signatures are self-contained,
  refer only to other signatures and to pervasive components
  (e.g., the Standard Basis Library)
- all non-trivial program units coded as functors
  that can be written and separately compiled
- one link structure: applies functors
  to produce one structure containing the executable program