Why Standard ML?

A language particularly suited to compiler implementation.

- Efficiency
- Safety
- Simplicity
- Higher-order functions
- Static type checking with type inference
- Polymorphism
- Algebraic types and pattern matching
- Modularity
- Garbage collection
- Exception handling
- Libraries and tools
Using the SML/NJ Compiler

- **Type “sml” to run the SML/NJ compiler**

  Normally installed in /usr/local/bin, which should be in your PATH.

- **Cntl-d exits the compiler, Cntl-c interrupts execution.**

- **Three ways to run ML programs:**

  1. type in code in the interactive read-eval-print loop
  2. edit ML code in a file, say foo.sml, then type command

      use “foo.sml”;

  3. use Compilation Manager (CM):

      CM.make “sources.cm”;
ML Tutorial 1

Expressions

- **Integers:** 3, 54, ~3, ~54
- **Reals:** 3.0, 3.14159, ~3.2E2
- **Overloaded arithmetic operators:** +, -, *, /, <, <=
- **Booleans:** true, false, not, orelse, andalso
- **Strings:** "abc", "hello world\n", x^".sml"
- **Lists:** [], [1,2,3], ["x","str"], 1::2::nil
- **Tuples:** (), (1,true), (3,"abc",true)
- **Records:** {a=1,b=true}, {name="fred",age=21}
- *conditionals, function applications, let expressions, functions*
ML Tutorial 2

**Declarations**: binding a name to a value

value bindings

```ml
val x = 3
val y = x + 1
```

function bindings

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

**Let expressions**: local definitions

```ml
let decl in expr end

let val x = 3
  fun f y = (y, x*y)
  in f(4+x)
end
```
Function expressions

The expression “fn var => exp” denotes a function with formal parameter var and body exp.

val inc = fn x => x + 1

is equivalent to

fun inc x = x + 1
Compound values

Tuples: $\langle \exp_1, \ldots, \exp_n \rangle$

$(3, 4.5)$

val x = ("foo", x*1.5, true)

val first = #1(x)
val third = #3(x)

Records: $\{ \text{lab}_1 = \exp_1, \ldots, \text{lab}_n = \exp_n \}$

val car = {make = "Ford", year = 1910}

val mk = #make car
val yr = #year car
ML Tutorial 5

**Patterns**

*a form to decompose compound values, commonly used in value bindings and function arguments*

```ml
val pat = exp
fun f(pat) = exp
```

**variable patterns:**

```
val x = 3
⇒ x = 3
fun f(x) = x+2
```

**tuple and record patterns:**

```
val pair = (3,4.0)
val (x,y) = pair
⇒ x = 3, y = 4.0

val {make=mk, year=yr} = car
⇒ mk = “Ford”, yr = 1910
```
Patterns

wildcard pattern: _ (underscore)

constant patterns: 3, “a”

    fun iszero(0) = true
    | iszero(_) = false

constructor patterns:

    val list = [1,2,3]
    val fst:::rest = list
⇒ fst = 1, rest = [2,3]

    val [x,_,y] = list
⇒ x = 1, y = 3
Pattern matching

match rule: \( \text{pat} \Rightarrow \text{exp} \)

match: \( \text{pat}_1 \Rightarrow \text{exp}_1 \mid \ldots \mid \text{pat}_n \Rightarrow \text{exp}_n \)

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

case expression: \( \text{case } \text{exp} \of \text{match} \)

function expression: \( \text{fn } \text{match} \)

clausal functional defn: \( \text{fun } f \text{ pat}_1 = \text{exp}_1 \mid f \text{ pat}_2 = \text{exp}_2 \mid \ldots \mid f \text{ pat}_2 = \text{exp}_2 \)
Pattern matching examples (function definitions)

fun length l = (case l
  of [] => 0
     | [a] => 1
     | _ :: r => 1 + length r
  (* end case *))

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

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

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


ML Tutorial 9

**Types**

*basic types:* int, real, string, bool
  3 : int, true : bool, “abc“ : string

*function types:* \( t_1 \rightarrow t_2 \)
  even : int \rightarrow bool

*product types:* \( t_1 \times t_2 \), unit
  (3, true): int \times bool, (): unit

*record types:* \( \{ lab_1 : t_1, \ldots, lab_n : t_n \} \)
  car: \{make : string, year : int\}

*type operators:* \( t \) list \ (for example)
  [1,2,3] : int list
Type abbreviations

\texttt{type tycon = ty}

\textbf{examples:}

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

\texttt{type tyvar tycon = ty}

\textbf{examples:}

\texttt{type 'a pair = 'a * 'a}
\texttt{type point = real pair}
Datatypes

**datatype** \( tycon = con_1 \text{ of } ty_1 \mid \ldots \mid con_n \text{ of } ty_n \)

This is a tagged union of variant types \( ty_1 \) through \( ty_n \). The tags are the data constructors \( con_1 \) through \( con_n \).

The data constructors can be used both in expressions to build values, and in patterns to deconstruct values and discriminate variants.

The “of \( ty \)” can be omitted, giving a nullary constructor.

Datatypes can be recursive.

**datatype** intlist = Nil | Cons of int * intlist
Datatype example

```
datatype btree = LEAF
               | NODE of int * btree * btree

fun depth LEAF = 0
                | depth (NODE(_,t1,t2)) =
                              max(depth t1, depth t2) + 1

fun insert(LEAF,k) = NODE(k,LEAF,LEAF)
                     | insert(NODE(i,t1,t2),k) =
                                      if k > i then NODE(i,t1,insert(t2,k))
                                      else if k < i then NODE(i,insert(t1,k),t2)
                                      else NODE(i,t1,t2)

(* in-order traversal of btrees *)
fun inord LEAF = []
                | inord(NODE(i,t1,t2)) =
                               inord(t1) @ (i :: inord(t2))
```
ML Tutorial 13

Representing programs as datatypes

```plaintext
type id = string

datatype binop = PLUS | MINUS | TIMES | DIV

datatype stm = SEQ of stm * stm
  | ASSIGN of id * exp
  | PRINT of exp list

and exp = VAR of id
  | CONST of int
  | BINOP of binop * exp * exp
  | ESEQ of stm * exp

val prog =
  SEQ(ASSIGN(“a”,BINOP(PLUS,CONST 5,CONST 3)),
       PRINT[VAR “a”])
```
ML Tutorial 14

Computing properties of programs: size

fun sizeS (SEQ(s1,s2)) = sizeS s1 + sizeS s2
  | sizeS (ASSIGN(i,e)) = 2 + sizeE e
  | sizeS (PRINT es) = 1 + sizeEL es

and sizeE (BINOP(_,e1,e2)) = sizeE e1 + sizeE e2 + 2
  | sizeE (ESEQ(s,e)) = sizeS s + sizeE e
  | sizeE _ = 1

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

sizeS prog ⇒ 8
Types Review

**Primitive types**
- unit, int, real, char, string, ..., instream, outstream, ...

**Composite types**
- unit, tuples, records
- function types

**Datatypes**
- types and n-ary type operators, tagged unions, recursive nominal type equality
- bool, list
- user defined: trees, expressions, etc.

**Type Abbreviations**
- types and n-ary type operators
- structural type equality
- type ‘a pair = ‘a * ‘a
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`

Inconsistencies will be detected as type errors.

- `if 1<2 then 3 else 4.0;`

  `stdIn:2.1-2.23 Error: types of rules don't agree`
  `earlier rule(s): bool -> int`
  `this rule: bool -> real`
  `in rule:`
  `false => 4.0`
Type Inference

In some cases involving record field selections, explicit type annotations (called ascriptions) may be required.

```
- datatype king = {name: string, 
                  born: int, 
                  crowned: int, 
                  died: int, 
                  country: string} 

- fun lifetime(k: king) = 
  =    #died k - #born k; 
val lifetime = fn : king -> int 

- fun lifetime({born,died,...}: king) = 
  =    died - born; 
val lifetime = fn : king -> int 
```

*partial record pattern*
Polymorphic Types

The most general type is inferred, which may be *polymorphic*

- `fun ident x = x;
  val ident = fn : 'a -> 'a`

- `fun pair x = (x, x);
  val ident = fn : 'a -> 'a * 'a`

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

- `val foo = pair 4.0;
  val foo : real * real`

- `fst foo;
  val it = 4.0: real`
Polymorphic Types

The most general type is inferred, which may be polymorphic

- `fun ident x = x;`
  `val ident = fn : 'a -> 'a`

- `fun pair x = (x, x);`
  `val ident = fn : 'a -> 'a * 'a`

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

- `val foo = pair 4.0;`
  `val foo : real * real`

- `fst foo;`
  `val it = 4.0 : real`
Polymorphic Data Structures

- `infixr 5 ::`
- `datatype 'a list = nil |
  :: of 'a * 'a list`

- `fun hd nil = raise Empty = | hd (x::_) = x;`
  `val hd = fn : 'a list -> 'a`

- `fun length nil = 0 = | length (_::xs) = 1 + length xs;`
  `val length = fn : 'a list -> int`

- `fun map f nil = nil = | map f (x::xs) = f x :: map f xs;`
  `val map = fn : ('a -> 'b) -> 'a list -> 'b list`
More Pattern Matching

Layered Patterns: $x \text{ as pat}$

(* merging two sorted lists of ints *)

fun merge(x, nil) = x
| merge(nil, y) = y
| merge(l as x::xs, m as y::ys) =
  if x < y then x :: merge(xs,m)
  else if y < x then y :: merge(l,m)
  else x :: merge(xs,ys);
val merge = fn : int list * int list -> int list

Note: although $<$ is overloaded, this definition is unambiguously typed with the lists assumed to be int lists because the $<$ operator defaults to the int version (of type int*int->bool).
- 5 div 0;               (* primitive failure *)
uncaught exception Div

exception NotFound of string;   (* control structure *)
type 'a dict = (string * 'a) list
fun lookup (s,nil) = raise (NotFound s)
   | lookup (s,(a,b)::rest) =
       if s = a then b else lookup (s,rest)
val lookup: string * 'a dict -> 'a

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

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

val y = lookup("moo",dict);
uncaught exception NotFound

val z = lookup("moo",dict) handle NotFound s =>
       (print ("can't find " ^ s ^ \
             "\n")); 0)
can't find moo
val z = 0 : int
References and Assignment

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

val linenum = ref 0; (* create updatable ref cell *)
val linenum = ref 0 : int ref

fun newline () = linenum := !linenum + 1; (* increment it *)
val newline = fn : unit -> unit

fun lineCount () = !linenum; (* access ref cell *)
val lineCount = fn : unit -> int

local val x = 1
    in fun new1 () = let val x = x + 1 in x end
end (* new1 always returns 2 *)

local val x = ref 1
    in fun new2 () = (x := !x + 1; !x)
end (* new2 returns 2, 3, 4, ... on successive calls *)
Simple Modules -- Structure

```ocaml
structure Ford =
struct
  type car = {make: string, built: int}
  val first = {make = "Ford", built: 1904}
  fun mutate ({make,built}: car) year =
    {make = make, built = year}
  fun built ({built,...}: car) = built
  fun show (c) = if built c < built first then " - "
    else "(generic Ford)"
end

structure Year =
struct
  type year = int
  val first = 1900
  val second = 2000
  fun newYear(y: year) = y+1
  fun show(y: year) = Int.toString y
end

structure MutableCar =
struct
  structure C = Ford
  structure Y = Year
end
```

A structure is an encapsulated, named, collection of declarations
Module Interfaces -- Signature

signature MANUFACTURER =
sig
  type car
  val first : car
  val built : car -> int
  val mutate : car -> int -> car
  val show : car -> string
end

signature YEAR =
sig
  eqtype year
  val first : year
  val second : year
  val newYear : year -> year
  val show : year -> string
end

signature MCSIG =
sig
  structure C : MANUFACTURER
  structure Y : YEAR
end

A signature is a collection of specifications for module components -- types, values, structures
Signature Matching

```ocaml
structure Year1 : YEAR =
  struct
    type year = int
    type decade = string
    val first = 1900
    val second = 2000
    fun newYear(y: year) = y+1
    fun leap(y: year) = y mod 4 = 0
    fun show(y: year) = Int.toString y
  end

structure MCar : MCSIG = MutableCar

val classic = Year1.show 1968
val antique = MCar.Y.show 1930
val x = Year1.leap(Year1.first)
```

Structure S matches SIG if S if every spec in SIG is matched by a component of S. S can have more components than are specified in SIG.

Use the dot notation to access components of structures.

Can’t access components not specified in signature.
Module Functions -- *Functors*

```ml
signature ORD =
sig
  type t
  val less : t * t -> bool
end

functor Sort(X: ORD) =
struct
  fun insert(x,nil) = [x] |
   insert(x,l as y::ys) =
     if X.less(x,y) then x::l
     else y::insert(x,ys)
  fun sort (m : X.t list) = foldl insert nil m
end

structure IntOrd : ORD =
structure IntSort = Sort(IntOrd)
```

Sort is a *parameterized module*, with parameter X: ORD

*functor application*
structure TextIO : sig

val stdIn : instream (* standard input *)
val stdout : outstream (* standard output *)
val stdErr : outstream (* standard error *)

val openIn: string -> instream (* open file for input *)
val openOut: string -> outstream (* open file for input *)
val openAppend: string -> outstream (* open file for appending *)

val closeIn: instream -> unit (* close input stream *)
val closeOut: outstream -> unit (* close output stream *)

val output: outstream * string -> unit (* output a string *)

val input: instream -> string (* input a string *)
val inputLine: instream -> string (* input a line *)

end
The Unix structure

signature UNIX =
  sig
    type proc
    val execute : (string * string list) -> proc
    val streamsOf : proc -> (TextIO.instream * TextIO.outstream)
    val reap : proc -> Posix.Process.exit_status
...
end

structure Unix : UNIX = ...

**Note**: the Unix structure in SML/NJ has a different signature from that given in the SML’97 Basis documentation.
An example of using the Unix structure

```ocaml
structure U = Unix

fun date () = let
  val pid = U.execute("/bin/date", [])
  val (inS, outS) = U.streamsOf pid
  val _ = TextIO.closeOut outS
  val now = TextIO.inputLine inS
  in
    TextIO.closeIn inS;
    U.reap pid;
    now
  end
```
Consider the problem of providing unique identifiers.

```
signature UID =

  sig
    type uid
    val same : (uid * uid) -> bool
    val compare : (uid * uid) -> order
    val gensym : unit -> uid

  end
```
structure UID => UID =
  struct
    type uid = int (* abstract *)
    fun same (a : uid, b) = (a = b)
    val compare = Int.compare
  val count = ref 0 (* hidden *)
  fun gensym () = let
    val id = !count
    in
      count := id + 1;
      id
    end
end