# Types

- ***primitive type*** *: int, real, bool, unit, string, char, ..., list type, record type, tuple type, function type;* ***type abbreviation****;* ***datatype definition****;*

***in ML, the type for each expression is inferred and checked for consistency at compile time !***

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

type king = {name    : string,
             born    : int,
             crowned : int,
             died    : int,
             quote   : string}

fun lifetime(k : king) = #died k - #born k

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






# Polymorphic Functions

- *polymorphic functions can be applied to arguments of different types, polymorphic functions usually do something simple !*

```
- fun ident x = x
val ident = fn : 'a -> 'a
- fun pairself x = (x,x)
val pairself = fn : 'a -> 'a * 'a
- fun pairint (x : int) = (x,x)
val pairint = fn : int -> int * int
- fun fst (x,y) = x
val fst = fn : 'a * 'b -> 'a
- fun snd (x,y) = y
val snd = fn : 'a * 'b -> 'b
- val foo = pairself 4.0;
val foo = (4.0,4.0) : real * real
- val bar = pairself "hello";
val bar = ("hello","hello") : string * string
- fst(foo);
val it = 4.0 : real
- pairint(4.0);
std_in:13.1-13.12 Error: operator and operand don't agree (tycon
    mismatch) operator domain: int, operand: real, in expression:
    pairint (4.0)
```






# Polymorphic Data Structures

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

fun rev nil = nil
  | rev (a::r) = (rev r)@[a]
val rev = fn : 'a list -> 'a list

datatype 'a tree = LEAF of 'a
                 | NODE of 'a * 'a tree * 'a tree
datatype 'a tree
  con LEAF : 'a -> 'a tree
  con NODE : 'a * 'a tree * 'a tree -> 'a tree

fun depth(LEAF _) = 1
  | depth(NODE(_,left,right) = 1+max(depth(left),depth(right))
val depth = fn : 'a tree -> int

val t = NODE(0, LEAF 1, LEAF 2)
val t = NODE (0,LEAF 1,LEAF 2) : int tree

- depth t;
val it = 2 : int
```






# More on Pattern Matching

- ***nested pattern --- use the "as" idiom***

```
(* example : merging two sorted list of integers *)
fun merge(x : int list, []) = x
  | merge([], y) = y
  | merge(x as (a::r), y as (b::z)) =
      if (a > b) then (b :: (merge(x, z)))
      else if (a < b) then (a :: (merge(r, y)))
           else (a::(merge(r, z)))
```

- ***partial record pattern --- must fully specify the record type !***

```
type king = {name : string, born : int, crowned : int,
             died : int, quote : string}

fun lifetime ({born, died, ...} : king) = died - born
```

# Higher-Order Functions

- *In ML, functions can be passed as arguments, returned as the result, and even stored in a data structure*

```
fun map f nil = nil
  | map f (a::r) = (f a)::(map f r)
val map = fn : ('a -> 'b) -> ('a list -> 'b list)

fun map2 f =
  (let fun m nil = nil
         | m (a::r) = (f a)::(m r)
    in m
   end)
val map2 = fn : ('a -> 'b) -> ('a list -> 'b list)

(* composing two functions *)
fun comp (f,g) = (fn x => g(f(x)))
val comp = fn : ('a -> 'b) * ('b -> 'c) -> ('a -> 'c)
```






# Exceptions

```
exception con
or exception con of ty

5 div 0;
uncaught exception Div

exception NotFound of string;
type dictionary = (string * string) list
fun lookup ([],s)= raise (NotFound s)
  | lookup ((a,b)::r,s : string) =
        if (a=s) then b else lookup(r,s)

val sampleDict = [("foo", "a sample name"),
                  ("bar", "another sample name")]

val x = lookup(sampleDict, "foo");
val x = "a sample name" : string

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

val z = lookup(sampleDict, "moo") handle NotFound s =>
          (print ("cannot find "^s^" in the dict"); "a word")
val z = "a word" : string
```






# Input and Output

```
structure TextIO (* read the basis manual on the web *)


type instream                        (* the input stream *)
type outstream                       (* the output stream *)

val stdIn : instream                (* the standard input stream *)
val stdOut : outstream              (* the standard output stream *)
val stdErr : outstream         (* the standard error output stream *)

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

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

val output : outstream * string -> unit

val input : instream -> string
val inputLine : instream -> string
............
```






# Assignment via References

- *ML supports updatable refernce cells*

```
(* assignment operator ":=", dereference "!" *)
let val lineNum = ref 0      (*  has type   int ref *)
 in lineNum := !lineNum + 1;
    lineNum := !lineNum + 1;
    lineNum
end
```

- *Assignement is different from value binding*

*just a value !*

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

*a pointer to a memory cell !*

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

# ML Module --- "Structure"

```
structure Ford =
  struct
    type car = {make : string, built : int}
    val first = {make = "Ford",  built = "1904"}
    fun mutate (c : car) year =
               {make = #make c, built = year}
    fun built (c : car) = #built c
    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 new_year(y : year) = y+1
    fun show(y : int) = makestring(y)
  end

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

***A structure is an encapsulated collection of declarations !***






# Module Interface --- "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 new_year : year -> year
    val show : year -> string
  end

signature MSIG =
  sig
    structure C : MANUFACTUER
    structure Y : YEAR
  end
```

***A signature is a collection of specifications for types, values and structures ...***






# Structure Matching

- *a structure S <u>matches</u> a signature SIG if every component specification in SIG is matched by a component in S.*
- *S can contain more components than SIG !!!*

```
structure Year1 : YEAR =
  struct
    type year = int
    val first = 1900
    val second = 2000
    fun new_year(y : year) = y+1
    fun decade y = (y - 1900) div 10
    fun show(y : int) =
         if y < 1910 orelse y >= final
         then Int.toString(y)
         else ("the '"^(Int.toString (decade y))^"0s")
  end

val long_gone = Year1.show 1968

structure MCar : MSIG = MutableCar
val long_gone2 = MCar.Y.show 1968
```

***use "long identifier" to refer to the structure component.***

*OR use the identifier directly after the structure is "open-ed"*






# Functors

- *A functor is a parametrized module. It takes a structure as argument and return another structure as the result !*

```
functor ProductLine(M : MANUFACTURER) =
  struct
    fun line(y,c) =
      if y = 2000 then ()
      else (output(std_out, ("\n" ^ (Int.toString y)
                             ^ "\t" ^ M.show c));
            line(y+1, M.mutate c (y+1))
           )
    fun show() = line(M.built M.first, M.first)
  end

structure FordLine = ProductLine(Ford)
val _ = FordLine.show();
```

# How to Use CM

- ***CM*** *inside* ***sml*** *is just like "****make****".*
- *the standard* ***makefile*** *is* ***sources.cm***

```
(* sources.cm for assignment 2 *)
Group is

driver.sml
errormsg.sml
tokens.sig
tokens.sml
tiger.lex
/c/cs421/lib/smlnj-lib.cm
```

**.lex** *ML-Lex source* **.grm** *ML-Yacc source* **.cm** *library inclusion* **.sml**, **.sig** *SML source*

- *after enter* ***sml****, type* `CM.make "sources.cm";`






# "tiger.lex" skeleton

```
type pos = int
type lexresult = Tokens.token

val lineNum = ErrorMsg.lineNum
val linePos = ErrorMsg.linePos
fun err(p1,p2) = ErrorMsg.error p1

fun eof() = let val pos = hd(!linePos)
             in Tokens.EOF(pos,pos)
            end

%%
%%
\n     => (inc lineNum; linePos := yypos :: !linePos;
           continue());
","    => (Tokens.COMMA(yypos,yypos+1));
var    => (Tokens.VAR(yypos,yypos+3));
"123"  => (Tokens.INT(123,yypos,yypos+3));
```






# "tokens.sig"

```
signature Toy_TOKENS =
  sig
    type linenum (* = int *)
    type token
    val TYPE:  linenum * linenum -> token
    val VAR:  linenum * linenum -> token
    val FUNCTION:  linenum * linenum -> token
    val BREAK:  linenum * linenum -> token
    ............
    val DOT:  linenum * linenum -> token
    val RBRACE:  linenum * linenum -> token
    val LBRACE:  linenum * linenum -> token
    val RBRACK:  linenum * linenum -> token
    val LBRACK:  linenum * linenum -> token
    val RPAREN:  linenum * linenum -> token
    val LPAREN:  linenum * linenum -> token
    val SEMICOLON:  linenum * linenum -> token
    val COLON:  linenum * linenum -> token
    val COMMA:  linenum * linenum -> token
    val STRING: (string) *  linenum * linenum -> token
    val INT: (int) *  linenum * linenum -> token
    val ID: (string) *  linenum * linenum -> token
    val EOF:  linenum * linenum -> token
   end
```






# "tokens.sml"

```
structure Tokens : Toy_TOKENS =
struct
  (* A "scaffold" structure for debugging lexers. *)
  val makestring = Int.toString
  type linenum = int
  type token = string
  fun TYPE(i,j) = "TYPE   " ^ makestring(i:int)
  fun VAR(i,j) = "VAR   " ^ makestring(i:int)
  fun FUNCTION(i,j) = "FUNCTION   " ^ makestring(i:int)
  fun BREAK(i,j) = "BREAK   " ^ makestring(i:int)
  fun OF(i,j) = "OF   " ^ makestring(i:int)
  fun END(i,j) = "END   " ^ makestring(i:int)
  fun IN(i,j) = "IN   " ^ makestring(i:int)
  fun NIL(i,j) = "NIL   " ^ makestring(i:int)
  fun LET(i,j) = "LET   " ^ makestring(i:int)
  fun DO(i,j) = "DO   " ^ makestring(i:int)
  fun TO(i,j) = "TO   " ^ makestring(i:int)
  fun FOR(i,j) = "FOR   " ^ makestring(i:int)
  ..............
  ..............
  fun STRING(s,i,j) = "STRING("^s^ ")     " ^ makestring(i:int)
  fun ID(s,i,j) = "ID("^s^")     " ^ makestring(i:int)
  fun EOF(i,j) = "EOF   " ^ makestring(i:int)
end
```

# "errormsg.sml"

```
signature ERRORMSG =
sig
    val anyErrors : bool ref
    val fileName : string ref
    val lineNum : int ref
    val linePos : int list ref
    val sourceStream : TextIO.instream ref
    val error : int -> string -> unit
    exception Error
    val impossible : string -> 'a   (* raises Error *)
    val reset : unit -> unit
end

structure ErrorMsg : ERRORMSG =
struct
  val anyErrors = ref false
  val fileName = ref ""
  val lineNum = ref 1
  val linePos = ref [1]
  val sourceStream = ref std_in

  fun reset() = ...
  exception Error
  ........
```






# "errormsg.sml" (cont'd)

```
structure ErrorMsg : ERRORMSG =
struct
  ........
  exception Error
  val makestring = Int.toString

  fun error pos (msg:string) =
      let fun look(p:int,a::rest,n) =
                  if a<p then app print [":",makestring n,
                                         ".",makestring (p-a)]
                  else look(p,rest,n-1)
            | look _ = print "0.0"
       in anyErrors := true;
          print (!fileName);
          look(pos,!linePos,!lineNum);
          print ":";
          print msg;
          print "\n"
      end

  fun impossible msg = ......

end  (* structure ErrorMsg *)
```






# "driver.sml"

```
structure Parse =
struct
  structure Lex = Mlex

  fun parse filename =
    let val file = TextIO.openIn filename
        fun get _ = TextIO.input file
        val lexer = Lex.makeLexer get

        fun do_it() =
          let val t = lexer()
           in print t; print "\n";
              if substring(t,0,3)="EOF" then () else do_it()
          end

     in do_it();
        TextIO.closeIn file
    end
end
```






# Assignment 2

*Writing a lexical analyzer for Tiger using ML-Lex*

- *how to handle nested comments ?*
- *how to handle string literals, integer literals, identifiers ?*
- *how to do the error handing especially for unclosed comments or strings (at the end of the file) ?*