Introduction to ML

- Standard ML of New Jersey

http://www.smlnj.org/

- Tutorials

http://www.smlnj.org/doc/literature.html#tutorials
http://octweb.emich.edu/~lizhang/courses/341/resources/sml

  - A Gentle Introduction to ML
  - Tips for Computer Scientists on Standard ML
  - Programming in Standard ML '97: An On-line Tutorial

Running SML

- On Windows
  - Add **PATH** to the environment
  - Run from MS-DOS prompt
  - **sml**
  - Use **control-z** to exit interpreter
- On Mac
  - Run from Terminal **./sml**
  - Use **control-d** to exit interpreter
  - Run from MLEditor

A First Look at ML

```
Standard ML of New Jersey
- 1+2*3;
val it = 7 : int
- 1+2*3
= ;
val it = 7 : int
```

Type an expression after **-** prompt;
ML replies with value and type

After the expression put a **;**.
If you forget, the next prompt will be **=**, meaning that ML expects more input. (You can then type the **;** it needs.)

Variable **it** is a special variable that is bound to the value of the expression you type

Constants (1)

```
- 1234;
val it = 1234 : int
- 123.4;
val it = 123.4 : real
```

Integer constants: standard decimal , but use tilde for unary negation (like **~1**)

Real constants: standard decimal notation

Note the type names: **int, real**

Constants (2)

```
- true;
val it = true : bool
- false;
val it = false : bool
```

Boolean constants **true** and **false**

ML is case-sensitive: use **true**, not **True** or **TRUE**

Note type name: **bool**

Constants (3)

```
- "fred";
val it = "fred" : string
- "H";
val it = "H" : string
- #"H";
val it = #"H" : char
```

String constants: text inside double quotes

Can use C-style escapes: **\n, \t, \\, \"**, etc.

Character constants: put # before a 1-character string, Note type names: **string** and **char**

Operators (1)

```
- ~ 1 + 2 - 3 * 4 div 5 mod 6;
val it = ~1 : int
- ~ 1.0 + 2.0 - 3.0 * 4.0 / 5.0;
val it = ~1.4 : real
```

Standard operators for integers, using **~** for unary negation and **-** for binary subtraction

Same operators for reals, but use **/** for division

Left associative, precedence is
{+,-} < **{*,/,div,mod}** < **{~}**.

Operators (2)

```
- "bibity" ^ "bobity" ^ "boo";
val it = "bibitybobityboo" : string
- 2 < 3;
val it = true : bool
- 1.0 <= 1.0;
val it = true : bool
- #"d" > #"c";
val it = true : bool
- "abce" >= "abd";
val it = false : bool
```

String concatenation: **^** operator

Ordering comparisons: **<**, **>**, **<=**, **>=**, apply to **string**, **char**, **int** and **real**

Order on strings and characters is lexicographic

Operators (3)

```
- 1 = 2;
val it = false : bool
- true <> false;
val it = true : bool
- 1.3 = 1.3;
Error: operator and operand don't agree
              [equality type required]
  operator domain: ''Z * ''Z
  operand:         real * real
  in expression:
    1.3 = 1.3
```

Equality comparisons: **=** and **<>**

Most types are equality testable: these are equality types

Type **real** is not an equality type

Operators (4)

```
- 1 < 2 orelse 3 > 4;
val it = true : bool
- 1 < 2 andalso not (3 < 4);
val it = false : bool
```

Boolean operators: **andalso**, **orelse**, **not**. (And we can also use **=** for equivalence and **<>** for exclusive or.)

Precedence so far
{orelse} < **{andalso}** < **{=,<>,<,>,<=,>=}** < **{+,-,^}** < **{*,/,div,mod}** < **{~,not}**

Operators (5)

```
- true orelse 1 div 0 = 0;
val it = true : bool
```

Note: **andalso** and **orelse** are short-circuiting operators: if the first operand of **orelse** is true, the second is not evaluated; likewise if the first operand of **andalso** is false

Technically, they are not ML operators, but keywords

All true ML operators evaluate all operands

Operators (6)

```
- if 1 < 2 then #"x" else #"y";
val it = #"x" : char
- if 1 > 2 then 34 else 56;
val it = 56 : int
- (if 1 < 2 then 34 else 56) + 1;
val it = 35 : int
```

Conditional expression (not statement) using
if … **then** … **else** …

Similar to C's ternary operator: **(1<2) ? 'x' : 'y'**

Value of the expression is the value of the **then** part, if the test part is true, or the value of the **else** part otherwise

There is no **if** … **then** construct

Let's Practice

- What is the value and ML type for each of these expressions?

```
1 * 2 + 3 * 4;
0x12 -  ~0x34;
"abc" ^ "def";
if (1e~3 < 1e~2) then 3.0 else 4.0;
1 < 2 orelse (1 div 0) = 0;
```

- What is wrong with each of these expressions?

```
10 / 5;
#"a" = #"b" or 1 = 2;
1.0 = 1.0;
not 2 > 1;
if (1<2) then 3;
```

Operators (7)

```
- 1 * 2;
val it = 2 : int
- 1.0 * 2.0;
val it = 2.0 : real
- 1.0 * 2;
Error: operator and operand don't agree [literal]
  operator domain: real * real
  operand:         real * int
  in expression:
    1.0 * 2
```

The * operator, and others like + and <, are overloaded to have one meaning on pairs of integers, and another on pairs of reals

ML does not perform implicit type conversion

Operators (8)

```
- real(123);
val it = 123.0 : real
- floor(3.6);
val it = 3 : int
- floor 3.6;
val it = 3 : int
- str #"a";
val it = "a" : string
```

Builtin conversion functions: **real** (**int** to **real**), **floor**, **ceil**, **round**, **trunc** (**real** to **int**) , **ord** (**char** to **int**), **chr** (**int** to **char**), **str** (**char** to **string**).

You apply a function to an argument in ML just by putting the function next to the argument. Parentheses around the argument are rarely necessary, and the usual ML style is to omit them

Function Associativity

- Function application is left-associative
- So **f a b** means **(f a) b**, which means:
  - first apply **f** to the single argument **a**;
  - then take the value **f** returns, which should be another function;
  - then apply that function to **b**
- More on how this can be useful later
- For now, just watch out for it

Function Associativity (cont.)

```
- square 2+1;
val it = 5 : int
- square (2+1);
val it = 9 : int
```

Function application has higher precedence than any operator

Be careful!

Let's Practice

- What if anything is wrong with each of these expressions?

```
trunc 5
ord "a"
if 0 then 1 else 2
if true then 1 else 2.0
chr(trunc(97.0))
chr(trunc 97.0)
chr trunc 97.0
```

Defining Variables (1)

```
- val x = 1+2*3;
val x = 7 : int
- x;
val it = 7 : int
- x = 7;
val it = true : bool
- val y = if x = 7 then 1.0 else 2.0;
val y = 1.0 : real
```

Define a new variable and bind it to a value using **val**.

Variable names should consist of a letter, followed by zero or more letters, digits, and/or underscores.

Defining Variables (2)

```
- val $$$ = 1+2*3;
val $$$ = 7 : int
- $$$;
val it = 7 : int
- $$$ + 3;
val it = 10 : int
```

Ten characters can never be part of any identifier:
( ) [ ] { } " . , ;

Symbolic identifiers are allowed but can not be combined with alphanumeric symbols: **!@#%** is ok but not **!@a**.

Defining Variables (3)

```
- val fred = 23;
val fred = 23 : int
- fred;
val it = 23 : int
- val fred = true;
val fred = true : bool
- fred;
val it = true : bool
```

You can define a new variable with the same name as an old one, even using a different type. (This is not particularly useful.)

This is not the same as assignment. It defines a new variable but does not change the old one. Any part of the program that was using the first definition of **fred**, still is after the second definition is made (3.2.1).

Let's Practice

- Suppose we make these ML declarations:

```
val a = "123";
val b = "456";
val c = a ^ b ^ "789";
val a = 3 + 4;
```

- Then what is the value and type of each of these expressions?

a
b
c

Garbage Collection

- Sometimes the ML interpreter will print a line like this, for no apparent reason:

```
GC #0.0.0.0.1.3:    (0 ms)
```

- This is what ML says when it is performing a "garbage collection": reclaiming pieces of memory that are no longer being used
- We'll see much more about this when we look at Java. For now, ignore it

Tuples and Lists(1)

```
- val barney = (1+2, 3.0*4.0, "brown");
val barney = (3,12.0,"brown") : int * real * string
- val point1 = ("red", (300,200));
val point1 = ("red",(300,200)) : string * (int *
int)
- #2 barney;
val it = 12.0 : real
- #1 (#2 point1);
val it = 300 : int
```

Use parentheses to form tuples

Tuples can contain other tuples

A tuple is like a record with no field names

To get i'th element of a tuple x, use **#i x**

Tuples and Lists(2)

```
- (1, 2);
val it = (1,2) : int * int
- (1);
val it = 1 : int
- #1 (1, 2);
val it = 1 : int
- #1 (1);
Error: operator and operand don't agree [literal]
  operator domain: {1:'Y; 'Z}
  operand:         int
  in expression:
    (fn {1=1,...} => 1) 1
```

There is no such thing as a tuple of one

Tuple Type Constructor

- ML gives the type of a tuple using * as a type constructor
- For example, **int * bool** is the type of pairs (x,y) where x is an **int** and y is a **bool**
- Note that parentheses have structural significance here: **int * (int * bool)** is not the same as **(int * int) * bool**, and neither is the same as **int * int * bool**

Tuples and Lists(3)

```
- [1,2,3];
val it = [1,2,3] : int list
- [1.0,2.0];
val it = [1.0,2.0] : real list
- [true];
val it = [true] : bool list
- [(1,2),(1,3)];
val it = [(1,2),(1,3)] : (int * int) list
- [[1,2,3],[1,2]];
val it = [[1,2,3],[1,2]] : int list list
```

Use square brackets to make lists

Unlike tuples, all elements of a list must be the same type

Tuples and Lists(4)

```
- [];
val it = [] : 'a list
- nil;
val it = [] : 'a list
```

Empty list is **[]** or **nil**

Note the odd type of the empty list: **'a list**

Any variable name beginning with an apostrophe is a type variable; it stands for a type that is unknown

'a list means a list of elements, type unknown

The **null** Test

```
- null [];
val it = true : bool
- null [1,2,3];
val it = false : bool
```

- **null** tests whether a given list is empty
- You could also use an equality test, as in **x = []**
- However, **null x** is preferred; we will see why in a moment

List Type Constructor

- ML gives the type of lists using **list** as a type constructor
- For example, **int list** is the type of lists of things, each of which is of type **int**
- **'a list** means a list of elements, type unknown
- A list is not a tuple

Tuples and Lists(5)

```
- [1,2,3]@[4,5,6];
val it = [1,2,3,4,5,6] : int list
- val x = #"c"::[];
val x = [#"c"] : char list
- val y = #"b"::x;
val y = [#"b",#"c"] : char list
```

The **@** operator concatenates lists
Operands are two lists of the same type
Note: **1@[2,3,4]** is wrong: either use **[1]@[2,3,4]** or **1::[2,3,4]**

List-builder (*cons*) operator is **::**
It takes an element of any type, and a list of elements of that same type, and produces a new list by putting the new element on the front of the old list

Tuples and Lists(6)

```
- val z = 1::2::3::[];
val z = [1,2,3] : int list
- hd z;
val it = 1 : int
- tl z;
val it = [2,3] : int list
- tl(tl z);
val it = [3] : int list
- tl(tl(tl z));
val it = [] : int list
```

The **::** operator is right-associative

The **hd** function gets the head of a list: the first element

The **tl** function gets the tail of a list: the whole list after the first element

Tuples and Lists(7)

```
- explode "hello";
val it = [#"h",#"e",#"l",#"l",#"o"] : char list
- implode [#"h",#"i"];
val it = "hi" : string
- concat ["ab", "cd"];
val it = “abcd" : string
```

The **explode** function converts a string to a list of characters, and the **implode** function does the reverse

A third operator **cancat** produces the string that is the concatenation of all the string on a list.

Let’s Practice

- What are the values of these expressions?
  #2(3,4,5)
  hd(1::2::nil)
  hd(tl(#2([1,2],[3,4])))
  implode(explode(“abcd”))
- What is wrong with the following expressions?
  1@2
  [#”a”,#”b”,”c”]
  hd(tl(tl [1,2]))
  [1]::[2,3]

Defining Functions (1)

```
- fun firstChar s = hd (explode s);
val firstChar = fn : string -> char
- firstChar "abc";
val it = #"a" : char
```

Define a new function and bind it to a variable using **fun**

Here **fn** means a function, the thing itself, considered separately from any name we've given it. The value of **firstChar** is a function whose type is **string -> char**

It is rarely necessary to declare any types, since ML infers them. ML can tell that **s** must be a **string**, since we used **explode** on it, and it can tell that the function result must be a **char**, since it is the **hd** of a **char list**

Function Definition Syntax

<*fun-def*> **::=**
fun <*function-name*> <*parameter*> **=** <*expression*> **;**

- **<function-name>** can be any legal ML name
- The simplest **<parameter>** is just a single variable name: the formal parameter of the function
- The **<expression>** is any ML expression; its value is the value the function returns
- This is a subset of ML function definition syntax

Function Type Constructor

- ML gives the type of functions using `->` as a type constructor
- For example, **`int -> real`** is the type of a function that takes an **`int`** parameter (the domain type) and produces a **`real`** result (the range type)
- Polymorphic function
  - **`- fun foo (x) = x;`**
  - **`- val foo = fn : 'a -> 'a`**

Defining Functions (2)

```
- fun quot(a,b) = a div b;
val quot = fn : int * int -> int
- quot (6,2);
val it = 3 : int
- val pair = (6,2);
val pair = (6,2) : int * int
- quot pair;
val it = 3 : int
```

All ML functions take exactly one parameter

To pass more than one thing, you can pass a tuple

ML Types So Far

- So far we have the primitive ML types **`int`**, **`real`**, **`bool`**, **`char`**, and **`string`**
- Also we have three type constructors:
  - Tuple types using **`*`**
  - List types using **`list`**
  - Function types using **`->`**

Combining Constructors

- When combining constructors, **`list`** has higher precedence than **`*`**, and **`->`** has lower precedence
  - **`int * bool list`** same as **`int * (bool list)`**
  - **`int * bool list -> real`** same as **`(int * (bool list)) -> real`**
- `->` is right-assciative
  - **`T1->T2->T3`** is **`T1->(T2->T3)`**
- Use parentheses as necessary for clarity

ML Types and Type Annotations

```
- fun prod(a,b) = a * b;
val prod = fn : int * int -> int
```

Why **`int`**, rather than **`real`**?

ML's default type for **`*`** (and **`+`**, and **`-`**) is **`int * int -> int`**

You can give an explicit type annotation to get **`real`** instead…

ML Types and Type Annotations

```
- fun prod(a:real,b:real):real = a*b;
val prod = fn : real * real -> real
```

Type annotation is a colon followed by a type

Can appear after any variable or expression

These are all equivalent:

```
fun prod(a,b):real = a * b;
fun prod(a:real,b) = a * b;
fun prod(a,b:real) = a * b;
fun prod(a,b) = (a:real) * b;
fun prod(a,b) = a * b:real;
fun prod(a,b) = (a*b):real;
fun prod((a,b):real * real) = a*b;
```

Defining Functions (3)

```
- fun fact n =
=   if n = 0 then 1
=   else n * fact(n-1);
val fact = fn : int -> int
- fact 5;
val it = 120 : int
```

Recursive factorial function

Defining Functions (4)

```
- fun listsum x =
=   if null x then 0
=   else hd x + listsum(tl x);
val listsum = fn : int list -> int
- listsum [1,2,3,4,5];
val it = 15 : int
```

Recursive function to add up the elements of an **`int list`**

A common pattern: base case for **`null x`**, recursive call on **`tl x`**

Defining Functions (5)

```
- fun length x =
=   if null x then 0
=   else 1 + length (tl x);
val length = fn : 'a list -> int
- length [true,false,true];
val it = 3 : int
- length [4.0,3.0,2.0,1.0];
val it = 4 : int
```

Recursive function to compute the length of a list

(This is predefined in ML, so you don't need this definition.)

Note type: this works on any type of list. It is polymorphic.

Defining Functions (6)

```
- fun badlength x =
=   if x=[] then 0
=   else 1 + badlength (tl x);
val badlength = fn : ''a list -> int
- badlength [true,false,true];
val it = 3 : int
- badlength [4.0,3.0,2.0,1.0];
Error: operator and operand don't agree
  [equality type required]
```

Same as previous example, but with **`x=[]`** instead of **`null x`**

Type variables that begin with two apostrophes, like **`''a`**, are restricted to equality types. ML insists on that restriction because we compared **`x`** for equality with the empty list.

That's why you should use **`null x`** instead of **`x=[]`**. It avoids unnecessary type restrictions.

Defining Functions (7)

```
- fun reverse L =
=   if null L then nil
=   else reverse(tl L) @ [hd L];
val reverse = fn : 'a list -> 'a list
- reverse [1,2,3];
val it = [3,2,1] : int list
```

Recursive function to reverse a list

That pattern again

Recursive Functions

- A recursive function consists of
  - A basis
  - An inductive step
- How about

```
(* Fact *)
fun fact n =
if n=1 then 1 else fact(n+1) div (n+1);
```

- Let's trace **`reverse`**

Essence of Recursions

1. Make a basis
2. Basis computes the result without making any recursive call
3. Write an inductive step, which are moving towards basis
4. Equalize inductive step

Inverse Mathematical Induction

1. Input and output of base case
2. Input and output of N case
3. Input and output of N-1 case
4. Equalize output of N and N-1 case

- Recursion code can be generated along the way
- No more trace – Don't even think about it
- No more "guessing"

Recursive Function Practice (1)

- Count number of true values in a list
- Duplicate each element of a list

Recursive Function Practice (2)

- Membership

Recursive Function Practice (3)

- Sum of a list of pairs
- Sum of the elements of a list of lists

infix

```
- infix 6 ++;
infix 6 ++
- fun (a,b) ++ (c,d) = (a+c, b+d);
val ++ = fn : (int * int) * (int * int) -> int * int
- (1,2) ++ (4,5);
val it = (5, 7) : int * int
```

- Predefined precedence
  - 7 – **/, div, mod, ***
  - 6 – **+, -, ^**
  - 5 – **::, @**
  - 4 – **=, <>, <, <=, >, >=**
  - 3 – **=, o**

*Tail Recursion

- A recursion can be rewritten as an iteration when all the recursive calls are in tail position, i.e., the call must be the last thing

```
(* tail recursive *)
fun foo (L) = if null L then nil
  else foo(tl L);

(* not tail recursive *)
fun length x =
     if null x then 0
     else 1 + length (tl x);
```

Accumulating Parameter

```
(* not tail recursive *)
fun length x =
     if null x then 0
     else 1 + length (tl x);

(* tail recursive *)
fun lengthHelper (x, len) =
     if null x then len
     else lengthHelper (tl x, len+1);

fun length(x) = lengthHelper(x, 0);
```

Two Patterns You Already Know

- We have seen that ML functions take a single parameter:
  `fun f n = n*n;`
- We have also seen how to specify functions with more than one input by using tuples:
  `fun f (a, b) = a*b;`
- Both **`n`** and **`(a, b)`** are patterns. The **`n`** matches and binds to any argument, while **`(a,b)`** matches any 2-tuple and binds **`a`** and **`b`** to its components

Underscore As A Pattern

```
- fun f _ = "yes";
val f = fn : 'a -> string
- f 34.5;
val it = "yes" : string
- f [];
val it = "yes" : string
```

- The underscore can be used as a pattern
- It matches anything, but does not bind it to a variable
- Preferred to:
  `fun f x = "yes";`

Constants As Patterns

```
- fun f 0 = "yes";
Warning: match nonexhaustive
          0 => ...
val f = fn : int -> string
- f 0;
val it = "yes" : string
```

- Any constant of an equality type can be used as a pattern
- But not:
  `fun f 0.0 = "yes";`

Non-Exhaustive Match

- In that last example, the type of **`f`** was **`int -> string`**, but with a "match non-exhaustive" warning
- Meaning: **`f`** was defined using a pattern that didn't cover all the domain type (**`int`**)
- So you may get runtime errors like this:

```
- f 0;
val it = "yes" : string
- f 1;
uncaught exception nonexhaustive match failure
```

Lists of Patterns As Patterns

```
- fun f [a,_] = a;
Warning: match nonexhaustive
          a :: _ :: nil => ...
val f = fn : 'a list -> 'a
- f [#"f",#"g"];
val it = #"f" : char
```

- You can use a list of patterns as a pattern
- This example matches any list of length 2
- It treats **a** and _ as sub-patterns, binding **a** to the first list element

Cons of Patterns As A Pattern

```
- fun f (x::xs) = x;
Warning: match nonexhaustive
          x :: xs => ...
val f = fn : 'a list -> 'a
- f [1,2,3];
val it = 1 : int
```

- You can use a cons of patterns as a pattern
- **x::xs** matches any non-empty list, and binds **x** to the head and **xs** to the tail
- Parens around **x::xs** are for precedence

ML Patterns So Far

- A variable is a pattern that matches anything, and binds to it
- A _ is a pattern that matches anything
- A constant (of an equality type) is a pattern that matches only that constant
- A tuple of patterns is a pattern that matches any tuple of the right size, whose contents match the sub-patterns
- A list of patterns is a pattern that matches any list of the right size, whose contents match the sub-patterns
- A cons (**::**) of patterns is a pattern that matches any non-empty list whose head and tail match the sub-patterns

Multiple Patterns for Functions

```
- fun f 0 = "zero"
= |   f 1 = "one";
Warning: match nonexhaustive
          0 => ...
          1 => ...
val f = fn : int -> string;
- f 1;
val it = "one" : string
```

- You can define a function by listing alternate patterns

Syntax

<*fun-def*> ::= **fun** <*fun-bodies*> **;**
<*fun-bodies*> ::= <*fun-body*>
| <*fun-body*> **'|'** <*fun-bodies*>
<*fun-body*> ::= <*fun-name*> <*pattern*> **=** <*expression*>

- To list alternate patterns for a function
- You must repeat the function name in each alternative

Overlapping Patterns

```
- fun f 0 = "zero"
= |   f _ = "non-zero";
val f = fn : int -> string;
- f 0;
val it = "zero" : string
- f 34;
val it = "non-zero" : string
```

- Patterns may overlap
- ML uses the first match for a given argument

Pattern-Matching Style

- These definitions are equivalent:

```
fun f 0 = "zero"
|   f _ = "non-zero";
fun f n =
  if n = 0 then "zero"
  else "non-zero";
```

- But the pattern-matching style usually preferred in ML
- It often gives shorter and more legible functions

Pattern-Matching Example (1)

- Original function:

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

- Rewritten using patterns:

```
fun fact 0 = 1
|   fact n = n * fact(n-1);
```

Pattern-Matching Example (2)

- Original function:
- Rewritten using patterns:

More Examples (1)

- Adding up all the elements of a list:
- Counting the true values in a list:

More Examples (2)

- Making a new list of integers in which each is one greater than in the original list:
- The length of a list:

More Examples (3)

- Tell whether a list is empty
- @

More Examples (4)

- Membership
- Duplicate each element of a list

More Examples (5)

- Sum of a list of pairs
- Sum of the elements of a list of lists

More Examples (6)

- Find the largest element of a list of reals
- Take a list of pairs of integers and orders the elements of each pair such that the small number is first

A Restriction

- You can't use the same variable more than once in the same pattern
- This is not legal:

```
fun f (a,a) = …  for pairs of equal elements
|   f (a,b) = …  for pairs of unequal elements
```

- You must use this instead:

```
fun f (a,b) =
  if (a=b) then …  for pairs of equal elements
  else …  for pairs of unequal elements
```

Patterns Everywhere

```
- val (a,b) = (1,2.3);
val a = 1 : int
val b = 2.3 : real
- val a::b = [1,2,3,4,5];
Warning: binding not exhaustive
          a :: b = ...
val a = 1 : int
val b = [2,3,4,5] : int list
```

- Patterns are not just for function definition
- Here we see that you can use them in a **val**
- More ways to use patterns, later

Local Variable Definitions

- When you use **val** at the top level to define a variable, it is visible from that point forward
- There is a way to restrict the scope of definitions: the **let** expression

<*let-exp*> ::= **let** <*definitions*> **in** <*expression*> **end**

Example with `let`

```
- let val x = 1 val y = 2 in x+y end;
val it = 3 : int;
- x;
Error: unbound variable or constructor: x
```

- The value of a **`let`** expression is the value of the expression in the **`in`** part
- Variables defined with **`val`** between the **`let`** and the **`in`** are visible only from the point of declaration up to the **`end`**

Proper Indentation for `let`

```
let
  val x = 1
  val y = 2
in
  x+y
end
```

- For readability, use multiple lines and indent **`let`** expressions like this
- Some ML programmers put a semicolon after each **`val`** declaration in a **`let`**

Long Expressions with `let`

```
fun days2ms days =
  let
    val hours = days * 24.0
    val minutes = hours * 60.0
    val seconds = minutes * 60.0
  in
    seconds * 1000.0
  end;
```

- The **`let`** expression allows you to break up long expressions and name the pieces
- This can make code more readable

Patterns with `let`

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```

- By using patterns in the declarations of a **`let`**, you can get easy "deconstruction"
- This example takes a list argument and returns a pair of lists, with half in each

Again, Without Good Patterns

```
let
  val halved = halve cs
  val x = #1 halved
  val y = #2 halved
in
  (a::x, b::y)
end;
```

- In general, if you find yourself using # to extract an element from a tuple, think twice
- Pattern matching usually gives a better solution

`halve` At Work

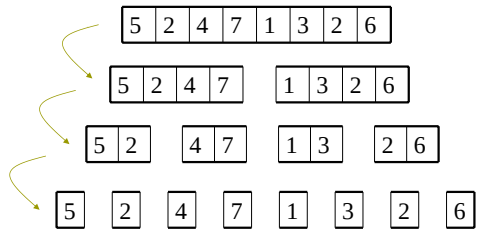
```
- fun halve nil = (nil, nil)
= |   halve [a] = ([a], nil)
= |   halve (a::b::cs) =
=       let
=         val (x, y) = halve cs
=       in
=         (a::x, b::y)
=       end;
val halve = fn : 'a list -> 'a list * 'a list
- halve [1];
val it = ([1],[]) : int list * int list
- halve [1,2];
val it = ([1],[2]) : int list * int list
- halve [1,2,3,4,5,6];
val it = ([1,3,5],[2,4,6]) : int list * int list
```

Merge Sort

- The **halve** function divides a list into two nearly-equal parts
- This is the first step in a merge sort
- For practice, we will look at the rest

Merge Sort: Divide Step

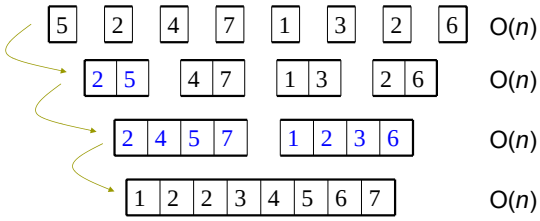
Step 1 – Divide



log(*n*) divisions to split an array of size *n* into single elements

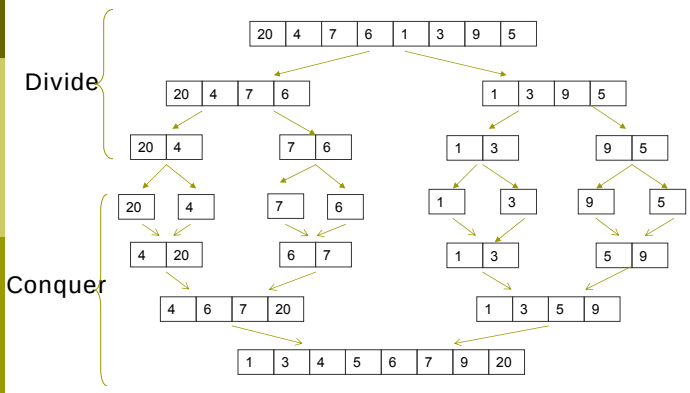
Mergesort: Conquer Step

Step 2 – Conquer



log*n* iterations, each iteration takes O*(n)* time. Total Time: O(*n* log*n*)

Mergesort: Another Example



Example: Merge

```
fun merge (nil, ys) = ys
|   merge (xs, nil) = xs
|   merge (x::xs, y::ys) =
      if (x < y) then x :: merge(xs, y::ys)
      else y :: merge(x::xs, ys);
```

- Merges two sorted lists
- Note: default type for **<** is **int**

Merge At Work

```
- fun merge (nil, ys) = ys
= |   merge (xs, nil) = xs
= |   merge (x::xs, y::ys) =
=       if (x < y) then x :: merge(xs, y::ys)
=       else y :: merge(x::xs, ys);
val merge = fn : int list * int list -> int list
- merge ([2],[1,3]);
val it = [1,2,3] : int list
- merge ([1,3,4,7,8],[2,3,5,6,10]);
val it = [1,2,3,3,4,5,6,7,8,10] : int list
```

Merge Sort

```
fun mergeSort (nil) = nil
|   mergeSort ([a]) = [a]
|   mergeSort theList =
      let
        val (x, y) = halve (theList)
      in
        merge(mergeSort x, mergeSort y)
      end;
```

- Merge sort of a list
- Type is **`int list -> int list`**, because of type already found for **`merge`**

Merge Sort At Work

```
- fun mergeSort nil = nil
= |   mergeSort [a] = [a]
= |   mergeSort theList =
=       let
=         val (x, y) = halve theList
=       in
=         merge(mergeSort x, mergeSort y)
=       end;
val mergeSort = fn : int list -> int list
- mergeSort [4,3,2,1];
val it = [1,2,3,4] : int list
- mergeSort [4,2,3,1,5,3,6];
val it = [1,2,3,3,4,5,6] : int list
```

Nested Function Definitions

- You can define local functions, just like local variables, using a **`let`**
- You should do it for helper functions that you don't think will be useful by themselves
- We can hide **`halve`** and **`merge`** from the rest of the program this way
- Another potential advantage: inner function can refer to variables from outer one

*Statement Lists

```
( <first expression>; …; <last expression> )
```

```
- fun printList (nil) =()
=     | printList(x::xs) = (
=     print(Int.toString(x));
=     print("\n");
=     printList(xs) );
val printList = fn: int list -> ()
```

- **`print`** function returns type: unit
- **`Int.toString()`** is a ML standard library function.

*Pretty Print a List

- Print an integer list; put m numbers on one line.

```
fun printListHelper (nil, _, _) =
  print("\n\n")
| printListHelper(L, n, m) = (
  print(Int.toString(hd(L)));
  print("\t");
  if n=m then (
     print("\n");
     printListHelper(tl(L), 1) )
  else printListHelper(tl(L), n+1) );
fun printPList(L)=printListHelper(L, 1, 5);
```

*More Pattern-Matching

- Last time we saw pattern-matching in function definitions:

```
fun f 0 = "zero"
  |   f _ = "non-zero";
```

- Pattern-matching occurs in several other kinds of ML expressions:

```
case n of
    0 => "zero" |
    _ => "non-zero";
```

*Match Syntax

- A rule is a piece of ML syntax that looks like this:

  <*rule*> ::= <*pattern*> **=>** <*expression*>

- A match consists of one or more rules separated by a vertical bar, like this:

  <*match*> ::= <*rule*> | <*rule*> '**|**' <*match*>

- Each rule in a match must have the same type of expression on the right-hand side
- A match is not an expression by itself, but forms a part of several kinds of ML expressions

*Case Expressions

```
- case 1+1 of
=    3 => "three" |
=    2 => "two" |
=    _ => "hmm";
val it = "two" : string
```

- The syntax is

  <*case-expr*> ::= **case** <*expression*> **of** <*match*>

- This is a very powerful case construct—unlike many languages, it does more than just compare with constants

*Example

```
case x of
  _::_::c::_ => c |
  _::b::_ => b |
  a::_ => a |
  nil => 0
```

The value of this expression is the third element of the list **x**, if it has at least three, or the second element if **x** has only two, or the first element if **x** has only one, or 0 if **x** is empty.

Predefined Functions

- When an ML language system starts, there are many predefined variables
- Some are bound to functions:

```
- ord;
val it = fn : char -> int
- ~;
val it = fn : int -> int
```

Defining Functions

- We have seen the **fun** notation for defining new named functions
- You can also define new names for old functions, using **val** just as for other kinds of values:

```
- val x = ~;
val x = fn : int -> int
- x 3;
val it = ~3 : int
```

Function Values

- Functions in ML do not have names
- Just like other kinds of values, function values may be given one or more names by binding them to variables
- The **fun** syntax does two separate things:
  - Creates a new function value
  - Binds that function value to a name

Anonymous Functions

- Named function:

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

- Anonymous function:

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

The `fn` Syntax

- Another use of the match syntax

  <*fun-expr*> ::= **fn** <*match*>

- Using **fn**, we get an expression whose value is an (anonymous) function
- We can define what **fun** does in terms of **val** and **fn**
- These two definitions have the same effect:
  - **fun f x = x + 2;**
  - **val f = fn x => x + 2;**

Using Matches to Define Functions

$$\begin{aligned}&\texttt{fun } f(P_1) = E_1\\ &\mid f(P_2) = E_2\\ &\mid \ldots\\ &\mid f(P_n) = E_n;\end{aligned}$$

$$\begin{aligned}&\texttt{val rec } f = \texttt{fn } P_1 \Rightarrow E_1\\ &\mid P_2 \Rightarrow E_2\\ &\mid \ldots\\ &\mid P_n \Rightarrow E;\end{aligned}$$

Examples (1)

- Original function:

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

- Rewritten using patterns:

```
fun fact 0 = 1
|   fact n = n * fact(n-1);
```

- Rewritten using matches:

```
val rec fact = fn  0 => 1
|   n => n * fact(n-1);
```

Examples (2)

- Original function:

```
fun reverse nil = nil
|   reverse (first::rest) =
        reverse rest @ [first];
```

- Rewritten using matches:

```
val rec reverse = fn nil => nil
|   first::rest => reverse rest @ [first];
```

- Rewritten using matches and case:

```
val rec reverse =
(fn t=>
  case t of
  nil => nil
  | x::xs => reverse xs @[x]) ;
```

*Exceptions

```
- 5 div 0;
uncaught exception divide by zero
  raised at: <file stdIn>
-hd(nil);
uncaught exception Empty
  raised at: boot/list.sml:36.38-36.43
```

Many functions are partial, i.e. they do not produce a value for some of the possible arguments of the function' domain type. It is essential that we be able to catch such errors.

*User Defined Exceptions

```
- exception Infinity;
exception Infinity
- fun myDiv (a:int, b) =
      if b = 0 then raise Infinity
      else a div b;
- myDiv (2,0);
uncaught exception Infinity
  raised at: stdIn:16.22-16.30
```

We can define our own exceptions and raise them in code we write when an exception condition is discovered. We can even give our defined exception an associated type.
exception Infinity of string;

*Handling Exceptions

```
- exception Infinity;
- fun myDiv (a:int, b) =
      if b = 0 then raise Infinity
      else a div b;
- fun safeDiv (a:int, b) = myDiv (a,b) handle
      Infinity => ~10000;
- saveDiv(2,0);
val it = ~10000 : int
```

Raise an uncaught exception always stops computation. We may prefer that when an exception is raised, there is an attempt to produce an appropriate value and continue the computation. Use the form <expression> **handle** <match>.

Polymorphic Functions

- ML is strongly typed
  - It is possible to determine the type of any variable or the value returned by any function by examining the program but without running the program
- Polymorphic functions
  - However, in ML we can define functions whose type are partially or completely flexible
  - **'a list**
  - Polymorphism (poly=many; morph=form)
    - The ability of a function to allow arguments of different types

Polymorphism

- One extreme application: identity function

```
- fun identity (x) = x;
val identity = fn : 'a -> 'a
- identity (2) + floor(identity (3.5));
val it = 5 : int
- identity (safeDiv);
val it = fn : int * int -> int
```

- First class values:

```
-fun polyDiv(a,b) = if b=0 then safeDiv else op div;
val polyDiv = fn : 'a * int -> int * int -> int
- val ff = polyDiv(2,0);
val ff = fn : int * int -> int
- ff(2,0);
val it = ~10000 : int
```

Restrictions on Polymorphic Functions

- A type variable **'a** is generalizable: can represent any one type that we choose. However, once that type is selected, the type cannot change.
- In **val x = e;** we can give **x** a polymorphic type only if **e** is a syntactic value (also known as a nonexpansive expression)
  - literals and identifiers (e.g. **3**, **n**)
  - function expressions (e.g. **(fn n=>n)**)
  - constructors applied to syntactic values (e.g. **(12, x::nil)**)

Restrictions (cont.)

- Syntactic (nonexpansive expression) values do not include function calls

```
- val x = rev [];
stdIn:34.1-34.15 Warning: type vars not generalized
because of value restriction are instantiated to
dummy types (X1,X2,...)
val x = [] : ?.X1 list
```

- SML does not give **x** a polymorphic type, since **rev []** is not a syntactic value. But of course SML can't tell what the type of **x** should be--it could be a list of anything (**'Z** in the textbook).

Restrictions (cont.)

- Type determination is complex
- When does a type problem arise
  - The expression is at the top level (not subexpression)
  - The type of the expression involves at least one type variable
  - The form of the expression does not meet the conditions for it to be nonexpansive

```
- identity(identity);
val it = fn : ?.X1 -> ?.X1
- (identity, identity);
val it = <poly-record> : ('a -> 'a) * ('b -> 'b)
- let val x = rev[] in 3::x end;
val it = [3] : int list
```

Operators and Polymorphism

- Polymorphism-destroying operators
  - Arithmatic +, -, *, /, div, mod
  - Inequality <, <=, >=, >
  - Boolean andalso, orelse, not
  - String concatenation ^
  - Type conversion operators ord, chr, …
- Operators that allow polymorphism
  - Tuple (), #1, #2
  - List ::, @, hd, tl, nil, []
  - Equality =, <>

The Equality Operators

- Equality types
  - Allow equality to be tested among values of that type

```
- val x = (1, 2);
val x = (1,2) : int * int
- x = (1, 2);
val it = true : bool
- val L = [1, 2, 3];
val L = [1, 2, 3] : int list
- M = (2, 3);
val M = [2, 3] : int list
- L <> M
val it = true : bool
```

Type ''Z

- ''Z type
  - Type variables whose values are restricted to be an equality type
  - Operator = requires arguments of the same equality type
  - Functions can not be compared for equality

```
- identity = identity;
stdIn:44.1-44.20 Error: operator and operand don't
agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         ('Y -> 'Y) * ('X -> 'X)
  in expression:
    identity = identity
```

Type 'a

- 'a type
  - Type variables whose values can be any type provided it is an equality type

```
- fun rev1 (L) = if L = nil then nil
            else rev1(tl(L)) @ [hd(L)];
val rev1 = fn : ''a list -> ''a list
```

```
- fun rev2 (L) = if null L then nil
            else rev1(tl(L)) @ [hd(L)];
val rev2 = fn : 'a list -> 'a list
```

```
- rev2 ([floor, trunc, ceil]);
- rev1 ([floor, trunc, ceil]);
```

Quick Sort in C

```
void  qsort ( void * base, size_t num, size_t width,
   int (*fncompare)(const void *, const void *) );

#include <stdio.h>
#include <stdlib.h>

int values[] = { 40, 10, 100, 90, 20, 25 };
int compare (const void * a, const void * b){
  return ( *(int*)a - *(int*)b );
}

int main (){
  int * pItem;  int n;
  qsort (values, 6, sizeof(int), compare);
  for (n=0; n<6; n++)  {
    printf ("%d ",values[n]);
  }
  return 0;
}
```

Using Anonymous Functions

- One simple application: when you need a small function in just one place
- Without **fn**:

```
- fun intBefore (a,b) = a < b;
val intBefore = fn : int * int -> bool
- quicksort ([1,4,3,2,5], intBefore);
val it = [1,2,3,4,5] : int list
```

- With **fn**:

```
- quicksort ([1,4,3,2,5], fn (a,b) => a<b);
val it = [1,2,3,4,5] : int list
- quicksort ([1,4,3,2,5], fn (a,b) => a>b);
val it = [5,4,3,2,1] : int list
```

The **op** keyword

```
- op *;
Val fn: int -> int
- quicksort ([1,4,3,2,5], op <);
val it = [1,2,3,4,5] : int list
```

- Binary operators are special functions
- Sometimes you want to treat them like plain functions: to pass **<**, for example, as an argument of type **int * int -> bool**
- The keyword **op** before an operator gives you the underlying function

Higher-order Functions

- Every function has an *order*:
  - A function that does not take any functions as parameters, and does not return a function value, has order 1
  - A function that takes a function as a parameter or returns a function value has order n+1, where *n* is the order of its highest-order parameter or returned value
- The **quicksort** we just saw is a second-order function

Practice

What is the order of functions with each of the following ML types?

```
int * int -> bool
int list * (int * int -> bool) -> int list
int -> int -> int
(int -> int) * (int -> int) -> (int -> int)
int -> bool -> real -> string
```

What can you say about the order of a function with this type?

```
('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

The **map** Function

- Used to apply a function **F** to every element of a list $[a_1, a_2, \ldots, a_n]$, and collect a list of results $[F(a_1), F(a_2), \ldots F(a_n)]$.

```
- fun simpleMap (F, nil) = nil
| simpleMap(F,x::xs) = F(x)::simpleMap(F,xs);
val simpleMap = fn : ('a -> 'b) * 'a list -> 'b list
- simpleMap(square, [1,2,3,4]);
val it = [1,4,9,16] : int list
- simpleMap (~, [1,2,3,4]);
val it = [~1,~2,~3,~4] : int list
- simpleMap (fn x=>x*x, [1.0, 2.0, 3.0]);
val it = [1.0, 4.0, 9.0] : real list
```

More **map** Examples

- Change every lower-case letter in a list of characters to the corresponding upper-case letter

```
simpleMap(fn(c)=>if c>=#"a" andalso
  c<=#"z" then chr(ord(c)-32) else c, L);
```

- What is wrong with the following code?

```
simpleMap (op +) [(1,2),(3,4),(5,6)];
```

More **map** Examples (cont.)

```
(* Simple map *)
fun  simpleMap(F, nil) = nil
|    simpleMap(F, x::xs) = F(x)::simpleMap(F, xs);

fun square (x:real) = x * x;
fun cube (x:real) = x * x * x;
fun plus(x:real,y) = x+y;

simpleMap(square, [1.0, 2.0, 3.0]);
simpleMap(cube, [1.0, 2.0, 3.0]);
simpleMap(~, [1.0, 2.0, 3.0]);

(* Anonymous *)
simpleMap(fn x=> x+1, [1.0, 2.0, 3.0]);
```

The **reduce** Function

- Used to apply a function of two parameters **F** to an nonempty list **[a_1, a_2, …, a_n]**

```
- Exception EmptyList;
- fun reduce (F, nil) = raise EmptyList
| reduce (F, [a]) = a
| reduce (F, x::xs) = F(x, reduce(F, xs));
val reduce = fn : ('a * 'a -> 'a) * 'a list -> 'a

- reduce(op +, [1,2,3,4]);
val val it = 10 : int
- reduce(op +, simpleMap(square, [1,2,3,4]));
val it = 30 : int
```

More **reduce** Examples

- Find the maximum of a list of reals

```
fun maxl L = reduce(fn(x,y)=> if
  x<y then y else x, L);
```

- Variance

```
fun variance L =
  let val n = real(length(L)) in
        reduce(op +, simpleMap(square,
  L))/n – square(reduce(op +, L)/n)
  end;
```

The **filter** Function

- Used to apply a predicate **P** and a list **[a_1, a_2, …, a_n]**, and collect a list of elements that satisfy **P**.

```
-fun filter (P, nil) = nil
| filter(P,x::xs)=if P(x) then x::filter(P,xs)
          else filter(P, xs);
val filter = fn : ('a -> bool) * 'a list -> 'a list

- filter(fn x => x<10, [1,10,23,5,16]);
val it = [1,5] : int list
- filter(fn x=> x mod 2 = 0, [1,2,3,4]);
val it = [2,4] : int list
```

Currying

- We've seen how to get two parameters into a function by passing a 2-tuple:
  fun f (a,b) = a + b;
- Another way is to write a function that takes the first argument, and returns another function that takes the second argument:
  fun g a = fn b => a+b;
- The general name for this is currying

Curried Addition

```
- fun f (a,b) = a+b;
val f = fn : int * int -> int
- fun g a = fn b => a+b;
val g = fn : int -> int -> int
- f(2,3);
val it = 5 : int
- g 2 3;
val it = 5 : int
```

- Remember that function application is left-associative
- So **g 2 3** means **((g 2) 3)**

Advantages

- No tuples: we get to write **`g 2 3`** instead of **`f(2,3)`**
- But the real advantage: we get to specialize functions for particular initial parameters

```
- val add2 = g 2;
val add2 = fn : int -> int
- add2 3;
val it = 5 : int
- add2 10;
val it = 12 : int
```

Advantages: Example

- Like the previous **`quicksort`**
- But now, the comparison function is a first, curried parameter

```
- quicksort (op <) [1,4,3,2,5];
val it = [1,2,3,4,5] : int list
- val sortBackward = quicksort (op >);
val sortBackward = fn : int list -> int list
- sortBackward [1,4,3,2,5];
val it = [5,4,3,2,1] : int list
```

Multiple Curried Parameters

- Currying generalizes to any number of parameters

```
- fun f (a,b,c) = a+b+c;
val f = fn : int * int * int -> int
- fun g a = fn b => fn c => a+b+c;
val g = fn : int -> int -> int -> int
- f (1,2,3);
val it = 6 : int
- g 1 2 3;
val it = 6 : int
```

Notation For Currying

- There is a much simpler notation for currying (on the next slide)
- The long notation we have used so far makes the little intermediate anonymous functions explicit

```
fun g a = fn b => fn c => a+b+c;
```

- But as long as you understand how it works, the simpler notation is much easier to read and write

Easier Notation for Currying

- Instead of writing:

```
fun f a = fn b => a+b;
```

- We can just write:

```
fun f a b = a+b;
```

- This generalizes for any number of curried arguments

```
- fun f a b c d = a+b+c+d;
val f = fn : int -> int -> int -> int -> int
```

More Curried Functions

```
fun sum1 (x:real, 0) = x
|  sum1 (x:real, y) = 1.0 + sum1 (x, y-1);

val sum1 = fn : real * int -> real

fun sum2 x 0  = x:real
|  sum2 x y  = 1.0 + sum2 x (y-1);

val sum2 = fn : real -> int -> real
```

Predefined Higher-Order Functions

- We will use three important predefined higher-order functions:
  - **map**
  - **foldr**
  - **foldl**
- Actually, **foldr** and **foldl** are very similar, as you might guess from the names

The **map** Function

- Used to apply a function to every element of a list, and collect a list of results

```
- map ~ [1,2,3,4];
val it = [~1,~2,~3,~4] : int list
- map (fn x => x+1) [1,2,3,4];
val it = [2,3,4,5] : int list
- map (fn x => x mod 2 = 0) [1,2,3,4];
val it = [false,true,false,true] : bool list
- map (op +) [(1,2),(3,4),(5,6)];
val it = [3,7,11] : int list
```

The **map** Function Is Curried

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

Composition of Functions (1)

- Function composition

http://en.wikipedia.org/wiki/Function_composition

  - Let F(x)=x+3, G(x)=y*y+2y, what is F **o** G, what is G **o** F?
  - Let the functions f: $X \to Y$ and g: $Y \to Z$ then g o f: $X \to Z$ defined by $(g \text{ o } f)(x) = g(f(x))$.
  - Example above

    (F **o** G) (x) = F(G(x)) = x*x+2x+3

    (G **o** F) (x) = G(F(X)) = x*x+8x+15

Composition of Functions (2)

- Let F(x)=x+3, G(x)=y*y+2y, what is F **o** G, what is G **o** F?

```
fun comp (F, G, x) = G(F(x));
val comp = fn : ('a -> 'b) * ('b -> 'c) *
'a -> 'c

fun comp2 (G, F, x) = F(G(x));
val comp = fn : ('a -> 'b) * ('b -> 'c) *
'a -> 'c
```

Composition of Functions (3)

```
fun comp(F, G, x) = G(F(x));
comp(fn x=>x+3, fn y=>y*y+2*y, 10);
val it = 195 :int

comp(fn y=>y*y+2*y, fn x=>x+3, 10);
val it = 123 :int

comp(fn x=>hd(x), fn y=>tl(y),
     [[1,2],[3,4],[5,6]]);
val it = [2] :int list
```

Composition of Functions (4)

```
fun compC F G x = G(F(x));
compC (fn x=>x+3, fn y=>y*y+2*y, 10);
Errors!

fun compC F G x = G(F(x));
compC (fn x=>x+3) (fn y=>y*y+2*y) 10;

compC (fn x=>hd(x)) (fn y=>tl(y))
    [[1,2],[3,4],[5,6]];
```

Composition of Functions (5)

```
fun F x = x+3;
fun G y = y*y+2*y;
val H = G o F;
val H = fn : int -> int
H 10;
val it = 195 : int

fun F x = hd x;
fun G x = tl x;
val H = G o F;
val H = fn : int -> int
H [[1,2],[3,4],[5,6]];
```

Composition of Functions (6)

```
fun comp F G =
  let fun C x = G(F(x))
  in
     C
  end;
fun F x = x+3;
fun G x = x*x+2*x;
val H = comp F G;
H 10;
```

Real Version of **map**

```
(* real version of MAP *)
fun map F =
  let
     fun M nil = nil
     |   M(x::xs) = F x :: M xs
  in
     M
  end;

val squareList = map square;
squareList [1.0, 2.0, 3.0];
```

That's All

- That's all the ML we will see
- There is, of course, a lot more
- A few words about the parts we skipped:
  - records (like tuples with named fields)
  - arrays, with elements that can be altered
  - references, for values that can be altered
  - exception handling

More Parts We Skipped

- support for encapsulation and data hiding:
  - structures: collections of datatypes, functions, etc.
  - signatures: interfaces for structures
  - functors: like functions that operate on structures, allowing type variables and other things to be instantiated across a whole structure
- API: the standard basis
  - predefined functions, types, etc.
  - Some at the top level but most in structures: **`Int.maxInt`**, **`Real.Math.sqrt`**, **`List.nth`**, etc.