CSE 341
Lecture 3

let expressions; pattern matching
Ullman 3.3 - 3.4

slides created by Marty Stepp
http://www.cs.washington.edu/341/
String and char (2.2, 2.4.5)

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>explode(<code>string</code>)</td>
<td>breaks a string into an array of characters</td>
</tr>
<tr>
<td>implode(<code>char list</code>)</td>
<td>combines a list of chars into a string</td>
</tr>
<tr>
<td>concat(<code>string list</code>)</td>
<td>merges all strings from a list into one</td>
</tr>
<tr>
<td>ord(<code>char</code>)</td>
<td>converts a char into its int ASCII value</td>
</tr>
<tr>
<td>chr(<code>int</code>)</td>
<td>converts an int ASCII value into a char</td>
</tr>
</tbody>
</table>

- ML's String structure has additional functions:
  - `String.size(string)` (* length *)
  - `String.substring(string, start, length)`
  - `String.sub(string, index)` (* charAt *)
The keyword let (3.4)

```plaintext
let
  val name = expression
in
  expression
end;
```

- binds a symbol to a function's "local environment"  
  - like declaring a local variable in Java
- the variable will be used only by the function  
  - recall that its value cannot change
- let expressions can appear anywhere an expression can
let example

(* The distance between points \((x_1,y_1),(x_2,y_2)\). *)
fun dist(x1, y1, x2, y2) =
  let
    val dx = x2 - x1
    val dy = y2 - y1
  in
    Math.sqrt(dx * dx + dy * dy)
  end;

• useful when you will be computing a value that is:
  ▪ used multiple times, or
  ▪ used in a complex way by the overall function's expression.
Using let with functions

let
  fun name = expression
in
  expression
end;

• technically, any binding (function or variable) can be made in a let-expression

• useful for writing "helper" functions
  ▪ subtasks required by a larger function
  ▪ recursive helpers when a function needs more parameters
Function let example

(* Least common multiple (LCM) of a and b. *)
fun lcm(a, b) =
  let
    fun gcd(x, y) =
      if y = 0 then x
      else gcd(y, x mod y)
  in
    a * b div gcd(a, b)
  end;

• Exercise: Change the function convertNames from last lecture to use a let helper function.
More about functions and let

A function declared inside a `let` expression:

- is part of the environment of the enclosing function
  - can refer to any of the enclosing function's parameters/vars

- defines its own local sub-environment
  - can declare its own `let` sub-expressions
  - can use parameter names that collide with those of the enclosing function, without ambiguity
ML bindings can contain **patterns** to match name(s) on the left side of = with the value(s) on the right.

- **basic pattern**: one name on left (matches all of right)
  - `val point = (3, ~5);`

- **tuple pattern**: tuple of names on left match parts on right
  - `val (x, y) = (3, ~5);`
  - `val (p, (x2, y2)) = ((3, ~5), (4, 7));`

- **list pattern**: list of names on left; same-size list on right
  - `val [a, b, c] = [8, 2, 6];`
List patterns

- *list pattern with ::* matches a head element and tail list
  - `val first::rest = [10, 20, 30, 40];`
  - first stores 10; rest stores `[20,30,40]`

- You can break out as many elements as you like:
  - `val first::second::rest = [10, 20, 30, 40];`
  - first stores 10; second stores 20; rest stores `[30,40]`

- list patterns can contain :: but not @
Functions and patterns

fun name(pattern1) = expression1
  | name(pattern2) = expression2
  ...
  | name(patternN) = expressionN;

- describes the function's behavior as a series of cases, each corresponding to a pattern of parameter values
  - better than lots of if-then-else expressions
  - avoids a lot of calls on hd, tl, and length on lists
  - must be exhaustive (match all possible parameter values)
Function pattern example

fun factorial(0) = 1
| factorial(n) = n * factorial(n - 1);

- If a client calls `factorial` and passes 0, it matches the first pattern (base case)
- if a client calls `factorial` and passes some other value, it matches the second pattern (recursive case)
Exercises

• Write a function `fibonacci` that accepts an integer \( n \) and produces the \( n \)th Fibonacci number, where the first two are 1 and all others are the sum of the prior 2.
  - `fibonacci(6)` produces 13

• Write a function `evens` that accepts a list and produces the elements at even-numbered indexes (0, 2, 4, ...).
  - `evens([6, 19, 2, 7])` produces [6, 2]
  - `evens([3, 0, 1, ~5, 8])` produces [3, 1, 8]

(Use patterns in your solutions.)
Inexhaustive patterns

- fun evens([]) = []
  = | evens(first::second::rest) = first::evens(rest);
val evens = fn : 'a list -> 'a list

- evens([6, 19, 2, 8, 5]);
uncaught exception Match [nonexhaustive match failure]
  raised at: stdin:9.58

• ML raises an exception if a call doesn't match any pattern
  ▪ this happens when the recursion reaches evens([5])
  ▪ we must add a third pattern to match a one-element list
• anonymous pattern `_` matches any single parameter

```haskell
fun contains([], _) = false
| contains(first::rest, value) = first = value
  orelse contains(rest, value);
```

• What, if any, is the difference between these?
  - `fun f1() = 42;`
  - `fun f2(_) = 42;`
The as keyword (3.3.2)

\[\text{name as pattern}\]

(* Removes any 0s from front of a list. *)

\[
\text{fun noLeadZeros([], []) = []} \\
| \qquad \text{noLeadZeros(lst as first :: rest) =} \\
| \quad \text{if first = 0 then noLeadZeros(rest)} \\
| \quad \text{else lst;}
\]

• if you like, you can name the entire parameter and also break its contents apart using a pattern
  ▪ saves us from having to write, else first :: rest;