Pattern matching with lists

In a pattern, :: can be used to describe a value.   Example:

```
fun len ([ ]) = 0
 |    len (x::xs) = 1 + len(xs)
```

The first pattern is the basis case and matches an empty list.

The second pattern requires a list with at least one element. The head is bound to x and the tail is bound to xs.

Problem: Noting that x is never used, improve the above implementation.

Problem: Write a function sum_evens(L) that returns the sum of the even values in L, an int list.

Problem: Write a function drop2(L) that returns a copy of L with the first two values removed. If the length of L is less than 2, return L.

Pattern matching, continued

What's an advantage of using a pattern to work with a list rather than the hd and tl functions?

Hint: Consider the following two implementations of sum:

```
fun sum(L) = hd(L) + sum(tl(L));

fun sum(x::xs) = x + sum(xs);
```

Practice

Problem: Write a function member(v, L) that produces true iff v is contained in the list L.

```
- member(7, [3, 7, 15]);
val it = true : bool
```

Problem: Write a function contains(s, c) that produces true iff the char c appears in the string s.

Problem: Write a function maxint(L) that produces the largest integer in the list L. Raise the exception Empty if the list has no elements.

Pattern construction

A pattern can be:

- A literal value such as 1, "x", true (but not a real)
- An identifier
- An underscore
- A tuple composed of patterns
- A list of patterns in [ ] form
- A list of patterns constructed with :: operators

Note the recursion.

Pattern construction, continued

Unfortunately, a pattern <u>cannot</u> contain an arbitrary expression:

```
- fun f(n > 0) = n     (* not valid! *)
   |  f(n) = n;
stdIn:1.5-2.13 Error: non-constructor applied to argument in pattern: >
```

Note "non-constructor" in the message. In a pattern, operators like :: are known as *constructors*.

An identifier cannot appear more than once in a pattern:

```
- fun equals(x, x) = true        (* not valid! *)
    | equals(_) = false;
stdIn:1.5-3.24 Error: duplicate variable in pattern(s): x
```

Practice

What bindings result from the following val declarations?

```
val [ [ (x, y) ] ] = [ [ ( 1, 2) ] ];

val [ [ x, y ] ] = [ [ 1, 2, 3 ] ];

val [(x,y)::z] = [ [ ( 1, (2 ,3) ) ] ];

val (x, ( y::ys, x ) ) = (1, ([2,3,4], (1, 2) ) );
```

A batch of odds and ends

let expressions

Producing output

Common problems

let expressions

A let expression can be used to create name/value bindings for use in a following expression to improve clarity and/or efficiency.

One way to write a function:

```
fun calc(x, y, z) = f1(g(x + y) - h(z)) + f2(g(x + y) - h(z))
```

An alternative with let:

```
fun calc(x,y,z) =
    let
        val diff = g(x+y) - h(z)
    in
        f1(diff) + f2(diff)
    end
```

Would it be practical for a compiler to make the above transformation automatically, using CSE (common subexpression elimination)?

W. H. Mitchell (whm@msweng.com)

let expressions, continued

General form of a let expression:

```
let
    declaration1
    declaration2
    ...
    declarationN
in
    expression
end
```

The value of *expression* is the value produced by the overall let expression. The name/value binding(s) established in the declaration(s) are only accessible in *expression*.

```
- val result = let val x = 1 val y = 2 in x + y end;
val result = 3 : int

- x;
stdIn:2.1 Error: unbound variable or constructor: x
```

let expressions, continued

A cute example of let from Ullman, p.78:

```
fun hundredthPower(x:real) =
  let
    val four = x*x*x*x
    val twenty = four*four*four*four*four
  in
    twenty*twenty*twenty*twenty*twenty
  end
```

Usage:

```
- hundredthPower(10.0);
val it = 1.0E100 : real
```

let expressions, continued

A function to count the number of even and odd values in a list of integers and return the result as int * int:

```
fun count_eo([ ]) = (0,0)
 |  count_eo(x::xs) =
    let
        val (even,odd) = count_eo(xs)
    in
        if x mod 2 = 0 then (even+1,odd)
                                    else (even,odd+1)
    end
```

Usage:

```
- count_eo([7,3,5,2]);
val it = (1,3) : int * int

- count_eo([2,4,6,8]);
val it = (4,0) : int * int
```

Would it be as easy to write without the let?

let expressions, continued

Imagine a function remove_min(L) that produces a tuple consisting of the smallest integer in L and a copy of L with that integer removed:

```
- remove_min([3,1,4,2]);
val it = (1,[3,2,4]) : int * int list

- remove_min([3,2,4]);
val it = (2,[3,4]) : int * int list

- remove_min([3,4]);
val it = (3,[4]) : int * int list

- remove_min([4]);
val it = (4,[]) : int * int list
```

let expressions, continued

remove_min can be used to write a function that sorts a list:

```
fun remsort([ ]) = []
 |  remsort(L) =
        let
            val (min, remain) = remove_min(L)
        in
            min::remsort(remain)
        end
```

Usage:

```
- remsort([3,1,4,2]);
val it = [1,2,3,4] : int list
```

let expressions, continued

A common technique is to define “helper” functions inside a function using a let expression.

Consider a function that returns every Nth element in a list:

```
- every_nth([10,20,30,40,50,60,70], 3);
val it = [30,60] : int list
```

Implementation:

```
fun every_nth(L, n) =
  let
    fun select_nth([ ],_,_) = [ ]
      |  select_nth(x::xs, elem_num, n) =
          if elem_num mod n = 0 then
             x::select_nth(xs, elem_num+1, n)
          else
             select_nth(xs, elem_num+1, n)
  in
    select_nth(L, 1, n)
  end;
```

Simple output

The print function writes its argument, a string, to standard output.

```
- print("abc");
abcval it = () : unit

- print("i = " ^ Int.toString(i) ^ "\n");   (* assume i = 7 *)
i = 7
val it = () : unit
```

A function to print the integers from 1 through N:

```
fun printN(n) =
    let
        fun printN'(0) = ""
          |  printN'(n) = printN'(n - 1) ^ Int.toString(n) ^ "\n"
  in
      print(printN'(n))
  end
```

Note the similarity between this function and countTo, on slide 37 (1...2...3). Could a generalization provide both behaviors?

Simple output, continued

Imagine a function to print name/value pairs:

```
- print_pairs([("x",1), ("y",10), ("z",20)]);
x 1
y 10
z 20
val it = () : unit
```

Problem: Write it!

Common problems

When loading source code sml typically cites the line and position in the line of any errors that are encountered:

```
% cat -n errors.sml     ( -n produces numbered output )
     1  fun count_eo([ ]) = (0,0)
     2   |  count_eo(x::xs) =
     3    let
     4      (even,odd) = count_eo(xs)
     5    in
     6      if x mud 2 = 0 then (even+1,odd)
     7                     else (even,Odd+1)
     8    end
```

Loading:

```
- use "errors.sml";
[opening errors.sml]
errors.sml:4.5 Error: syntax error: inserting  VAL
errors.sml:6.10-6.13 Error: unbound variable or constructor: mud
errors.sml:7.31-7.34 Error: unbound variable or constructor: Odd
```

Common problems, continued

Infinite recursion:

```
fun sum(0) = 0
  |   sum(n) = n + sum(n);
```

Usage:

```
- sum(5);
...no response...
^C
Interrupt
```

Common problems, continued

Type mismatch when calling a function:

```
- fun double(n) = n*2;
val double = fn : int -> int

- fun f(x) = double(3.0 * x);
stdIn:3.27 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand:         real
  in expression:
    double (3.0 * x)
```

Type mismatch when recursively calling a function:

```
- fun f(x,y) = f(x);
Error: operator and operand don't agree [circularity]
  operator domain: 'Z * 'Y
  operand:         'Z
  in expression:
    f x
```

Common problems, continued

A non-exhaustive match warning can indicate incomplete reasoning, typically a missing basis case to terminate recursion:

```
- fun len(x::xs) = 1 + len(xs);
Warning: match nonexhaustive
          x :: xs => ...

- len([1,2,3]);
uncaught exception nonexhaustive match failure
  raised at: stdIn:368.3
```

Use of fun instead of | (or-bar) for a function case:

```
- fun f(1) = "one"
  fun f(n) = "other";
Warning: match nonexhaustive
          1 => ...

val f = <hidden-value> : int -> string
val f = fn : 'a -> string
```

Larger Examples

expand

travel

tally

expand

Consider a function that expands a string in a trivial packed representation:

```
- expand("x3y4z");
val it = "xyyyzzzz" : string

- expand("123456");
val it = "244466666" : string
```

Fact: The digits 0 through 9 have the ASCII codes 48 through 57. A character can be converted to an integer by subtracting from it the ASCII code for 0. Therefore,

```
fun ctoi(c) = ord(c) - ord(#"0")

fun is_digit(c) = #"0" <= c andalso c <= #"9"

- ctoi(#"5");
val it = 5 : int

- is_digit(#"x");
val it = false : bool
```

expand, continued

One more function:

```
fun repl(x, 0) = []
  | repl(x, n) = x::repl(x, n-1)
```

What does it do?

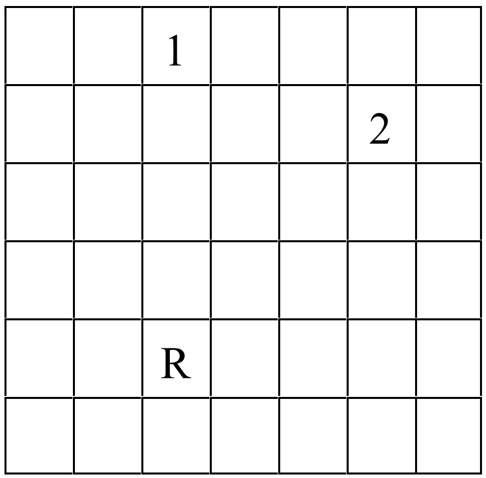
Finally, expand:

```
fun expand(s) =
    let
        fun expand'([ ]) = [ ]
         |  expand'([c]) = [c]
         |  expand'(c1::c2::cs) =
               if is_digit(c1) then
                   repl(c2, ctoi(c1)) @ expand'(cs)
               else
                   c1 :: expand'(c2::cs)
    in
        implode(expand'(explode(s)))
    end;
```

travel

Imagine a robot that travels on an infinite grid of cells. The robot's movement is directed by a series of one character commands: n, e, s, and w.

In this problem we will consider a function travel of type string -> string that moves the robot about the grid and determines if the robot ends up where it started (i.e., did it get home?) or elsewhere (did it get lost?).



If the robot starts in square R the command string nnnn leaves the robot in the square marked 1. The string nenene leaves the robot in the square marked 2. nnessw and news move the robot in a round-trip that returns it to square R.

travel, continued

Usage:

```
- travel("nnnn");
val it = "Got lost" : string

- travel("nnessw");
val it = "Got home" : string
```

How can we approach this problem?

travel, continued

One approach:

1. Map letters into integer 2-tuples representing X and Y displacements on a Cartesian plane.

2. Sum the X and Y displacements to yield a net displacement.

Example:

Argument value:	"nnee"
Mapped to tuples:	(0,1) (0,1) (1,0) (1,0)
Sum of tuples:	(2,2)

Another:

Argument value:	"nnessw"
Mapped to tuples:	(0,1) (0,1) (1,0) (0,-1) (0,-1) (-1,0)
Sum of tuples:	(0,0)

travel, continued

A couple of building blocks:

```
fun mapmove(#"n") = (0,1)
  |  mapmove(#"s") = (0,~1)
  |  mapmove(#"e") = (1,0)
  |  mapmove(#"w") = (~1,0)

fun sum_tuples([ ]) = (0,0)
  |  sum_tuples((x,y)::ts) =
    let
        val (sumx, sumy) = sum_tuples(ts)
    in
        (x+sumx, y+sumy)
    end
```

travel, continued

The grand finale:

```
fun travel(s) =
    let
        fun mk_tuples([ ]) = [ ]
          |   mk_tuples(c::cs) = mapmove(c)::mk_tuples(cs)

        val tuples = mk_tuples(explode(s))

        val disp = sum_tuples(tuples)

    in
    if disp = (0,0) then
        "Got home"
    else
        "Got lost"
    end
```

Note that mapmove and sum_tuples are defined at the outermost level. mk_tuples is defined inside a let. Why?

Larger example: tally

Consider a function tally that prints the number of occurrences of each character in a string:

```
- tally("a bean bag");
a 3
b 2
  2
g 1
n 1
e 1
val it = () : unit
```

Note that the characters are shown in order of decreasing frequency.

How can this problem be approached?

tally, continued

Implementation:

```
(*
 * inc_entry(c, L)
 *
 *    L is a list of (char * int) tuples that indicate how many times a
 *  character has been seen.
 *
 *    inc_entry() produces a copy of L with the count in the tuple
 *  containing the character c incremented by one.  If no tuple with
 *  c exists, one is created with a count of 1.
 *)
 fun inc_entry(c, [ ]) = [(c, 1)]
   |   inc_entry(c, (char, count)::entries) =
         if c = char then
             (char, count+1)::entries
         else
             (char, count)::inc_entry(c, entries)
```

tally, continued

```
(* mkentries(s) calls inc_entry() for each character in the string s *)

fun mkentries(s) =
    let
        fun mkentries'([ ], entries) = entries
          |  mkentries'(c::cs, entries) =
             mkentries'(cs, inc_entry(c, entries))
    in
        mkentries'(explode s, [ ])
    end

(* fmt_entries(L) prints, one per line, the (char * int) tuples in L *)

fun fmt_entries(nil) = ""
   |  fmt_entries((c, count)::es) =
      str(c) ^ " " ^ Int.toString(count) ^ "\n" ^ fmt_entries(es)
```

tally, continued

```
(*
 * sort, insert, and order_pair work together to provide an insertion sort
 *
 *    insert(v, L) produces a copy of the int list L with the int v in the
 * proper position.  Values in L are descending order.
 *
 *    sort(L) produces a sorted copy of L by using insert() to place
 *  values at the proper position.
 *
 *)
fun insert(v, [ ]) = [v]
  |  insert(v, x::xs) =
       if order_pair(v,x) then v::x::xs
                                  else x::insert(v, xs)

fun sort([ ]) = [ ]
  |  sort(x::xs) = insert(x, sort(xs))

fun order_pair((_, v1), (_, v2)) = v1 > v2
```

tally, continued

With all the pieces in hand, tally itself is a straightforward sequence of calls.

```
(*
 * tally: make entries, sort the entries, and print the entries
 *)
fun tally(s) = print(fmt_entries(sort(mkentries(s))))
```