CIS 500

Software Foundations

Fall 2002

11 September

Administrivia

Two new recitations have been opened.

The registration limit on the course has been (or will soon be) removed.

New recitation signup sheets going around the room now.

Even if you gave us your preferences on Monday, you must fill in the new sheet when it comes around.

Basic Pattern Matching

Recursive functions on lists tend to have a standard shape: we test whether the list is empty, and if it is not we do something involving the head element and the tail.

```
# let rec listSum (l:int list) =
    if l = [] then 0
    else List.hd l + listSum (List.tl l);;
```

OCaml provides a convenient pattern-matching construct that bundles the emptiness test and the extraction of the head and tail into a single syntactic form:

```
# let rec listSum (l: int list) =
    match l with
        []    -> 0
    | x::y -> x + listSum y;;
```

Pattern matching can be used with types other than lists. For example, here it is used on integers:

```
# let rec fact (n:int) =
     match n with
        0 -> 1
      | _ -> n * fact(n-1);;
```

The _ pattern here is a wildcard.

Complex Patterns

The basic elements (constants, variable binders, wildcards, [], ::, etc.) may be combined in arbitrarily complex ways in match expressions:

```
# let silly l =
    match l with
      [_;_;_] -> "three elements long"
    | _::x::y::_::_::rest -> if x>y then "foo" else "bar"
    | _ -> "dunno";;
val silly : int list -> string = <fun>

# silly [1;2;3];;
- : string = "three elements long"
# silly [1;2;3;4];;
- : string = "dunno"
# silly [1;2;3;4;5];;
- : string = "bar"
```

Type Inference

One pleasant feature of OCaml is a powerful type inference mechanism that allows the compiler to calculate the types of variables from the way in which they are used.

```
# let rec fact n =
    match n with
      0 -> 1
    | _ -> n * fact(n-1);;
val fact : int -> int = <fun>
```

The compiler can tell that `fact` takes an integer argument because `n` is used as an argument to the integer `*` and `-` functions.

Similarly:

```
# let rec listSum l =
    match l with
       []   -> 0
     | x::y -> x + listSum y;;
val listSum : int list -> int = <fun>
```

Polymorphism

Suppose we ask the OCaml system to infer the type of the following very similar definition:

```
# let rec length l =
      match l with
        []    -> 0
      | _::y -> 1 + listSum y;;
val listSum : 'a list -> int = <fun>
```

The `'a` here, pronounced "alpha," is a **type variable**, standing for an arbitrary type.

The inferred type tells us that the function can take a list with elements of **any** type (i.e., a list of with elements of type alpha, for any choice of alpha).

We'll have more to say about polymorphism on Monday.

Tuples

items connected by commas are "tuples"

```
# "age", 44;;
- : string * int = "age", 44
# ("professor",("age", 33));;
- : string * (string * int) = "professor", ("age", 33)
# ("children", ["bob";"ted";"alice"]);;
- : string * string list = "children", ["bob"; "ted"; "alice"]


# let g (x,y) = x*y;;
val g : int * int -> int = <fun>
```

How many arguments does g take?

Tuples are not lists

Please do not confuse them!

```
# let tuple = "cow", "dog", "sheep";;
val tuple : string * string * string = "cow", "dog", "sheep"
# let list = ["cow"; "dog"; "sheep"];;
val list : string list = ["cow"; "dog"; "sheep"]
# List.hd tuple;;
This expression has type string * string * string but is here used
with type
   'a list
# List.hd list;;
- : string = "cow"
# let tup2 = 1, "cow";;
val tup2 : int * string = 1, "cow"
# let l2 = [1; "cow"];;
This expression has type string but is here used with type int
```

Tuples and Pattern Matching

Tuples can be "deconstructed" by pattern matching:

```
# let lastName name =
    match name with
      (n1,_,_) -> n1;;
# lastName  ("Pierce", "Benjamin", "Penn");;
- : string = "Pierce"
```

Example: Finding words

Suppose we want to take a list of characters and return a list of lists of characters, where each element of the final list is a "word" from the original list.

```
# split ['t';'h';'e';' ';'b';'r';'o';'w';'n';' ';'d';'o';'g'];;
- : char list list =
      [['t'; 'h'; 'e']; ['b'; 'r'; 'o'; 'w'; 'n']; ['d'; 'o'; 'g']]
```

(Note that character constants are written with single quotes.)

An implementation of `split`

```
# let rec loop w l =
    match w,l with
      _, [] -> [w]
    | _, (' '::ls) -> w :: (loop [] ls)
    | w, (c::ls) -> loop (w @ [c]) ls;;
val loop : char list -> char list -> char list list = <fun>

# let split l = loop [] l;;
val split : char list -> char list list = <fun>
```

Note the use of both tuple patterns and nested patterns (as well as wildcards).

Aside: Local function definitions

The `loop` function is completely local to `split`: there is *no reason for* anybody else to use it — or even, for anybody else to be able to see it! It is good style in OCaml to write such definitions as local bindings:

```
# let split l =
    let rec loop w l =
      match w,l with
        _, [] -> [w]
      | _, (' '::ls) -> w :: (loop [] ls)
      | w, (c::ls) -> loop (w @ [c]) ls
    in
    loop [] l;;
```

In general, any let definition that can appear at the top level

```
# let ...;;
# e;;;
```

can also appear in a `let...in...` form.

```
# let ... in e;;;
```

A Better Split

Our `split` function worked fine for the example we tried it on. But here are some other tests:

```
# split ['a';' ';' ';'b'];;
- : char list list = [['a']; []; ['b']]

# split ['a';' '];;
- : char list list = [['a']; []]
```

Could we refine `split` so that it would leave out these spurious empty lists in the result?

Sure. Just add a couple of cases to the pattern match:

```
# let better_split l =
    let rec loop w l =
      match w,l with
        [],[] -> []
      | _,[] -> [w]
      | [], (' '::ls) -> loop [] ls
      | _, (' '::ls) -> w :: (loop [] ls)
      | w, (c::ls) -> loop (w @ [c]) ls in
    loop [] l;;

# better_split ['a';'b';' ';' ';'c';' ';'d';' '];;
- : char list list = [['a'; 'b']; ['c']; ['d']]
# better_split ['a';' '];;
- : char list list = [['a']]
# better_split [' ';' '];;
- : char list list = []
```

Basic Exceptions

OCaml's exception mechanism is roughly similar to that found in, for example, Java.

We begin by defining an exception:

```
# exception Bad;;
```

Now, encountering `raise Bad` will immediately terminate evaluation and return control to the top level:

```
# let rec fact n =
    if n<0 then raise Bad
    else if n=0 then 1
    else n * fact(n-1);;
# fact (-3);;
Exception: Bad.
```

Naturally, exceptions can also be caught within a program (using the `try...with...` form), but let's leave that for another day.

Data Types

We have seen a number of data types:

int
bool
string
char
lists
tuples

Ocaml has other data types we haven't seen (e.g., float) One can also create completely new data types.

The need for new types

The ability to construct new types is an essential part of most programming languages.

Suppose we are building a (very simple) graphics program that displays circles and squares. We can represent each of these with three real numbers.

A circle is represented by the co-ordinates of its center and its radius. A square is represented by the co-ordinates of its bottom left corner and its width. So we can represent both shapes as elements of the type:

```
float * float * float
```

However, there are two problems with using this type to represent circles and squares. First, it is a bit long and unwieldy, both to write and to read. Second, because their types are identical, there is nothing to prevent us from mixing circles and squares. For example, if we write

```
# let areaOfSquare (_,_,d) = d *. d;;
```

we might accidentally apply the `areaOfSquare` function to a circle and get a nonsensical result.

(Nb. Numerical operations on the `float` type are written differently from the corresponding operations on `int` — e.g., `+.` instead of `+`. See the OCaml manual for more information.)

Data Types

We can improve matters by defining `square` as a new type:

```
# type square = Square of float * float * float;;
```

This does two things:

♦ It creates a **new** type called `square` that is different from any other type in the system.

♦ It creates a **constructor** called `Square` (with a capital `S`) that can be used to create a `square` from three floats. For example:

```
# Square(1.1,2.2,3.3);;
- : square = Square (1.1, 2.2, 3.3)
```

Taking data types apart

We take types apart with (surprise, surprise...) pattern matching.

```
# let areaOfSquare s =
    match s with
      Square(_, _, d) -> d *. d;;
val areaOfSquare : square -> float = <fun>


# let bottomLeftCoords s =
    match s with
      Square(x, y, _) -> (x,y);;
val bottomLeftCoords : square -> float * float = <fun>
```

So we can use constructors like Square both as functions and as patterns.

Constructors are recognized by being capitalized (the first letter is upper case).

These functions can be written a little more concisely by combining the pattern matching with the function header:

```
# let areaOfSquare (Square(_, _, d)) = d *. d;;
# let bottomLeftCoords (Square(x, y, _)) = (x,y);;
```

Continuing, we can define a data type for circles in the same way.

```
# type circle = Circle of float * float * float;;

# let c = Circle (1.0, 2.0, 2.0);;

# let areaOfCircle (Circle(_, _, r)) = 3.14159 *. r *. r;;

# let centerCoords (Circle(x, y, _)) = (x,y);;

# areaOfCircle c;;
- : float = 12.56636
```

We cannot now apply a function intended for type square to a value of type circle:

```
# areaOfSquare(c);;
This expression has type circle but is here used with type square.
```

Variant types

Going back to the idea of a graphics program, we obviously want to have several shapes on the screen at once. For this we'd probably want to keep a list of circles and squares, but such a list would be heterogenous. How do we make such a list?

The answer is that we build a type that can be either a circle or a square.

```
# type shape = Circle of float * float * float
             | Square of float * float * float;;
```

Now both constructors Circle and Square create values of type shape. For example:

```
# Square (1.0, 2.0, 3.0);;
- : shape = Square (1.000000, 2.000000, 3.000000)
```

A type that can have more than one form is often called a variant type.

We can also write functions that do the right thing on all forms of a variant type. Again we use pattern matching:

```
# let area s =
    match s with
      Circle (_, _, r) -> 3.14159 *. r *. r
    | Square (_, _, d) -> d *. d;;

# area (Circle (0.0, 0.0, 1.5));;
- : float = 7.0685775
```

Further examples

A "heterogeneous" list:

```
# let l = [Circle (0.0, 0.0, 1.5); Square (1.0, 2.0, 1.0);
           Circle (2.0, 0.0, 1.5); Circle (5.0, 0.0, 2.5)];;

# List.map area l;;
- : float list = [7.0685775; 1; 7.0685775; 19.6349375]
```

Mixed-mode Arithmetic

Many programming languages (Lisp, Basic, Perl, database query languages) use variant types internally to represent numbers that can be either integers or floats. This amounts to "tagging" each numeric value with an indicator that says what kind of number it is.

```
# type num = Int of int | Float of float;;

# let add r1 r2 =
    match (r1,r2) with
      (Int  i1,   Int  i2)   -> Int (i1 + i2)
    | (Float r1, Int  i2)   -> Float (r1 +. float i2)
    | (Int  i1,   Float r2) -> Float (float i1 +. r2)
    | (Float r1, Float r2) -> Float (r1 +. r2);;

# add (Int 3) (Float 4.5);;
- : num = Float 7.5
```

Multiplication, `mult` follows exactly the same pattern:

```
# let mult r1 r2 =
    match (r1,r2) with
      (Int i1,   Int i2)   -> Int (i1 * i2)
    | (Float r1, Int i2)   -> Float (r1 *. float i2)
    | (Int i1,   Float r2) -> Float (float i1 *. r2)
    | (Float r1, Float r2) -> Float (r1 *. r2);;
```

```
# let unaryMinus n =
    match n with Int i -> Int (- i) | Float r -> Float (-. r);;

# let minus n1 n2 = add n1 (unaryMinus n2);;

# let rec fact n =
    if n = Int 0 then Int 1
    else mult n (fact (minus n (Int 1)));;

# fact (Int 7);;
- : num = Int 5040
```

A Data Type for Optional Values

Suppose we are implementing a simple lookup function for a telephone directory. We want to give it a string and get back a number (say an integer). We expect to have a function `lookup` whose type is

```
lookup: string  -> directory -> int
```

where `directory` is a (yet to be decided) type that we'll use to represent the directory.

However, this isn't quite enough. What happens if a given string isn't in the directory? What should `lookup` return?

There are several ways to deal with this issue. One is to raise an exception. Another is based on the following data type:

```
# type maybe = Absent | Present of int;;
```

To see how this type is used, let's represent our directory as a list of pairs:

```
# let directory  = [("Joe", 1234); ("Martha", 5672);
                     ("Jane", 3456); ("Ed", 7623)];;


# let rec lookup s l =
    match l with
      [] -> Absent
    | (k,i)::t -> if k = s then Present(i)
                           else lookup s t;;


# lookup "Jane" directory;;
- : maybe = Present 3456


# lookup "Karen" directory;;
- : maybe = Absent
```

Built-in options

Because options are often useful in functional programming, OCaml provides a built-in type `t option` for each type `t`. Its constructors are `None` (corresponding to `Absent`) and `Some` (`Present`).

```
# let rec lookup s l =
    match l with
      [] -> None
    | (k,i)::t -> if k = s then Some(i)
                               else lookup s t;;


# lookup "Jane" directory;;
- : maybe = Some 3456
```

Enumerations

Our `maybe` data type has one variant, `Absent`, that is a "constant" constructor carrying no data values with it. Data types in which all the variants are constants can actually be quite useful...

```
# type color = Red | Yellow | Green;;
# let next c =
    match c with Green -> Yellow | Yellow -> Red | Red -> Green;;
```

```
# type day =   Sunday | Monday | Tuesday | Wednesday
              | Thursday | Friday | Saturday;;
# let weekend d =
    match d with
      Saturday -> true
    | Sunday   -> true
    | _        -> false;;
```

A Boolean Data Type

A simple data type can be used to replace the built-in booleans.

We use the constant constructors True and False to represent true and false. We'll use different names as needed to avoid confusion between our booleans and the built-in ones:

```
# type myBool = False | True;;
# let myNot b = match b with False -> True | True -> False;;
# let myAnd b1 b2 =
    match (b1,b2) with
      (True,  True)  ->  True
    | (True,  False) ->  False
    | (False, True)  ->  False
    | (False, False) ->  False;;
```

Note that the behavior of myAnd is not quite the same as &&!

Recursive Types

Consider the tiny language of arithmetic expressions defined by the following (BNF-like) grammar:

```
exp  ::=  number
          ( exp + exp )
          ( exp - exp )
          ( exp * exp )
```

(We'll come back to these grammars in more detail next week...)

We can translate this grammar directly into a datatype definition:

```
type ast =
    ANum of int
  | APlus of ast * ast
  | AMinus of ast * ast
  | ATimes of ast * ast;;
```

Notes:

♦ This datatype (like the original grammar) is recursive.

♦ The type `ast` represents abstract syntax trees, in which surface details such as parentheses have been dropped.
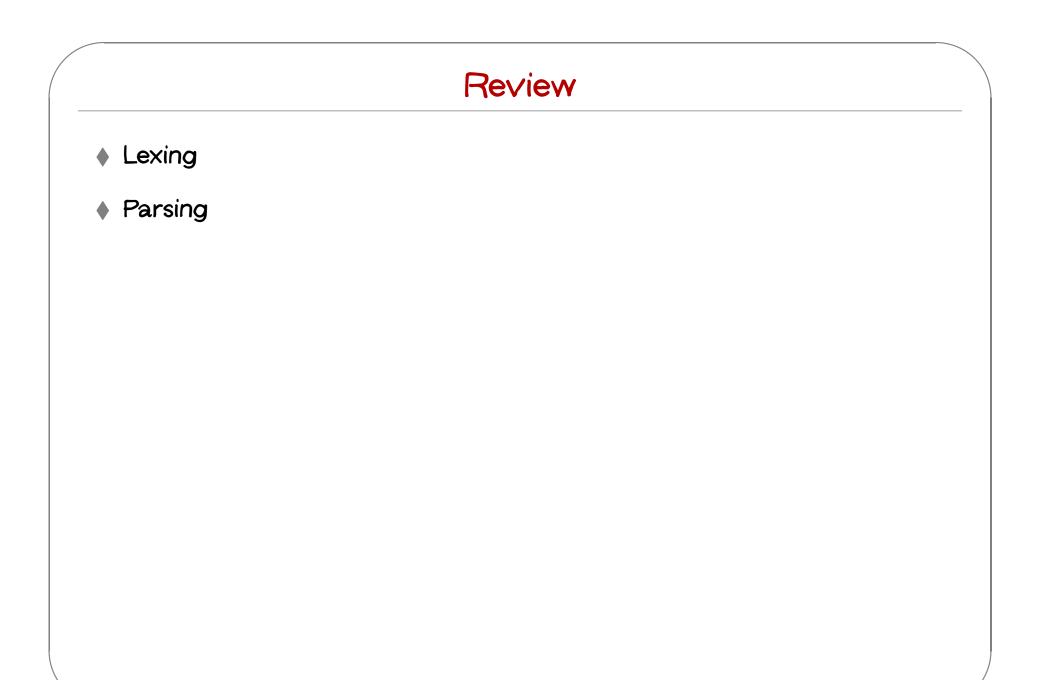
An evaluator for expressions

Goal: write an avaluator for these expressions.

```
val eval : ast -> int = <fun>

# eval (ATimes (APlus (ANum 12, ANum 340), ANum 5));;
- : int = 1760
```

The solution uses a recursive function plus a pattern match.

```
let rec eval e =
  match e with
    ANum i -> i
  | APlus (e1,e2) -> eval e1 + eval e2
  | AMinus (e1,e2) -> eval e1 - eval e2
  | ATimes (e1,e2) -> eval e1 * eval e2;;
```

Review

- ♦ Lexing

- ♦ Parsing

A final example

Goal: write a function that takes two lists of equal length and interleaves their elements:

```
# interleave [1;2;3] [4;5;6];;
- : int list = [1; 4; 2; 5; 3; 6]
```

Solution:

```
# let rec interleave l1 l2 =
    match l1,l2 with
      [],[] -> []
    | x::xs, y::ys -> x::y::(interleave xs ys)
    | _ -> raise Bad;;
```

Harder version

Now suppose that we want to calculate all the possible interleavings of two lists — i.e., all the lists that can be formed by interleaving elements of the input lists in an arbitrary fashion (but maintaining the ordering from the original lists).

For example:

```
# interleave [1;3] [2;4];;

- : int list list =
      [[1; 3; 2; 4]; [1; 2; 3; 4]; [1; 2; 4; 3]; [2; 1; 3; 4];
       [2; 1; 4; 3]; [2; 4; 1; 3]]
```

```
# let rec cons_all e l =
    match l with
      [] -> []
    | h::t -> (e::h) :: (cons_all e t);;
val cons_all : 'a -> 'a list list -> 'a list list = <fun>

# let rec interleave l1 l2 =
    match l1,l2 with
      [],_ -> [l2]
    | _,[] -> [l1]
    | x::xs, y::ys ->
        List.append
          (cons_all x (interleave xs l2))
          (cons_all y (interleave l1 ys));;
val interleave : 'a list -> 'a list -> 'a list list = <fun>
```