Announcements

- Advanced recitation starts this week. (Wednesday, 3:30-5:00 PM)
- Homework 2 is on the website. It is due in one week.
- Homework 1 was due at noon.

The material in this course is mostly conceptual and mathematical. However, experimenting with small implementations is an excellent way to deepen intuitions about many of the concepts we will encounter. For this purpose, we will use the OCaml language. OCaml is a large and powerful language. For our present purposes, though, we can concentrate just on the "core" of the language, ignoring most of its features.
Functional Programming

OCaml is a functional programming language i.e., a language in which the functional programming style is the dominant idiom. Other well-known functional languages include Lisp, Scheme, Haskell, and Standard ML.

The functional style can be described as a combination of...

- persistent data structures (which, once built, are never changed)
- recursion as a primary control structure
- heavy use of higher-order functions (functions that take functions as arguments and/or return functions as results)
- emphasis on control, expression, and enhancement
- more emphasis on higher-order functions (functions that take functions as arguments and/or return functions as results)
- less use of first-order functions (functions that take and return values, rather than functions)
- less emphasis on control, expression, and enhancement

The functional style can be described as a combination of...

OCaml is a functional programming language — i.e., a language in which the

The top level

Giving things names

The let construct gives a name to the result of an expression so that it can be

Computing with Expressions

OCaml is an expression language. A program is an expression. The "meaning" of the program is the value of the expression. The "meaning" of an expression is an expression. The "meaning" of the expression is the value of the expression.

The top level

The let construct gives a name to the result of an expression so that it can be

CIS500, Programming with OCaml

5

The top level

Giving things names

The let construct gives a name to the result of an expression so that it can be

CIS500, Programming with OCaml

6

The top level

Computing with Expressions

CIS500, Programming with OCaml

7

Computing with Expressions
### Functions

```ocaml
# let cube (x:int) = x * x * x;;
val cube : int -> int = <fun>
```

We call `x` the parameter of the function `cube`. The expression `cube 9` is its body.

The type printed by OCaml, `int -> int` (pronounced `int arrow int`), indicates that `cube` is a function that should be applied to an integer argument and yields another integer as its result. The type annotation on the parameter `(x:int)` is optional. OCaml can figure it out. However, your life will be much simpler if you put it on.

Noting that OCaml responds by printing just `<fun>` as the function's value.

Here is a function with two parameters:

```ocaml
# let sum sq (x:int) (y:int) = x * x + y * y;;
val sum sq : int -> int -> int = <fun>
```

The type printed for `sum sq` is `int -> int -> int`, indicating that it should be applied to two integer arguments and yields an integer as its result.

To invoke a function declaration in OCaml, we write `cube 3` and `sum sq 3 4` rather than `cube(3)` and `sum sq(3,4)`. The function declaration in OCaml is slightly different from languages like C, C++, and Java families. We write `cube x` and `sum_sq x y` rather than `cube(x)` and `sum_sq(x, y)`. The type annotation on the parameter `x:(int)` is optional. OCaml can figure it out. However, your life will be much simpler if you put it on.

#### The boolean type

The boolean type has only two values: `true` and `false`. Comparison operations return boolean values.

```ocaml
# 1 = 2;;
val it : bool = false
```

```ocaml
# 4 >= 3;;
val it : bool = true
```

The `not` is a unary operation on booleans.

```ocaml
# not 5 <= 10;;
val it : bool = false
```

```ocaml
# not 2 = 2;;
val it : bool = false
```

#### Conditional expressions

The result of the conditional expression `if B then E1 else E2` is either the result of `E1` or `E2`, depending on whether the result of `B` is `true` or `false`.

```ocaml
# if 3 < 4 then 7 else 100;;
val it : int = 7
```

```ocaml
# if 3 < 4 then (3 + 3) else (10 * 10);;
val it : int = 6
```

```ocaml
# if false then (3 + 3) else (10 * 10);;
val it : int = 100
```

```ocaml
# if false then false else true;;
val it : bool = true
```

## Conditionals

There are only two values of type boolean: `true` and `false`.

### The function "valume"

Note that OCaml responds to a function declaration by printing just `<fun>` as the function's value. However, your life will be much simpler if you put it on.

The type annotation on the parameter `x:(int)` is optional. OCaml can figure it out. However, your life will be much simpler if you put it on.

We call `x` the parameter of the function `cube`. The expression `cube x` is its body.

```ocaml
# let volume (x:int) = x * x * x;;
val volume : int -> int = <fun>
```

```ocaml
# if cube (x:int) = 729;#

val it : bool = true
```

```ocaml
# let cube (x:int) = x * x * x;;
val cube : int -> int = <fun>
```

## Comparison operations return boolean values.

There are only two values of type boolean: `true` and `false`.

### The boolean type

The boolean type has only two values: `true` and `false`. Comparison operations return boolean values.

```ocaml
# 1 = 2;;
val it : bool = false
```

```ocaml
# 4 >= 3;;
val it : bool = true
```

The `not` is a unary operation on booleans.

```ocaml
# not 5 <= 10;;
val it : bool = false
```

```ocaml
# not 2 = 2;;
val it : bool = false
```

#### Conditional expressions

The result of the conditional expression `if B then E1 else E2` is either the result of `E1` or `E2`, depending on whether the result of `B` is `true` or `false`.

```ocaml
# if 3 < 4 then 7 else 100;;
val it : int = 7
```

```ocaml
# if 3 < 4 then (3 + 3) else (10 * 10);;
val it : int = 6
```

```ocaml
# if false then (3 + 3) else (10 * 10);;
val it : int = 100
```

```ocaml
# if false then false else true;;
val it : bool = true
```

## Conditionals

There are only two values of type boolean: `true` and `false`.

### The function "valume"

Note that OCaml responds to a function declaration by printing just `<fun>` as the function's value. However, your life will be much simpler if you put it on.

The type annotation on the parameter `x:(int)` is optional. OCaml can figure it out. However, your life will be much simpler if you put it on.

We call `x` the parameter of the function `cube`. The expression `cube x` is its body.

```ocaml
# let volume (x:int) = x * x * x;;
val volume : int -> int = <fun>
```

```ocaml
# if cube (x:int) = 729;#

val it : bool = true
```

```ocaml
# let cube (x:int) = x * x * x;;
val cube : int -> int = <fun>
```
Lists

One handy structure for storing a collection of data values is a list. Lists are provided as a built-in type in OCaml and a number of other popular languages. We can build a list in OCaml by writing out its elements, enclosed in square brackets and separated by semicolons.

```ocaml
# [1; 2; 3; 5];;
- : int list = [1; 2; 3; 5]
```

The type that OCaml prints for this list is "integer list", pronounced "list of integers." OCaml does not allow different types of elements to be mixed within the same list.

Lists are homogeneous

In fact, for every type `t`, we can build lists of type `t list`. We can also build lists of lists:

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

OCaml provides a number of built-in operations that return lists. The most basic one creates a new list by appending an element to the front of an existing list. It is written `::` and pronounced "cons" (because it constructs lists).

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

We can also build lists whose elements are drawn from any of the basic types (int, bool, etc.).

```ocaml
# let add123 (l: int list) = 1 :: 2 :: 3 :: l;;
val add123 : int list -> int list = <fun>
# add123 [5; 6; 7];;
- : int list = [1; 2; 3; 5; 6; 7]
# add123 [];;
- : int list = [1; 2; 3]
```

The empty list, written `[]`, is sometimes called "nil."
Some recursive functions that generate lists:

### `let rec repeat (k:int) (n:int) = (* A list of n copies of k *)`

- `if n = 0 then [] else k :: repeat k (n-1);;`

- `repeat 7 12 ;;` : `int list = [7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7]`

### `let rec fromTo (m:int) (n:int) = (* The numbers from m to n *)`

- `if n < m then [] else m :: fromTo (m+1) n;;`

- `fromTo 9 18 ;;` : `int list = [9; 10; 11; 12; 13; 14; 15; 16; 17; 18]`

**Constructing Lists**

Any list can be built by `consing` its elements together:

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

**Taking Lists Apart**

OCaml provides two basic operations for extracting the parts of a list:

- `List.hd ([]) ;;` : `int = 0`

- `List.tl ([]) ;;` : `int list = []`

- `List.hd (List.tl ([]) ;;)` : `int = 0`

- `List.tl (List.hd ([]) ;;)` : `int list = []`

- `List.hd (List.tl (List.tl ([]) ;;)) ;;` : `int = 0`

- `List.tl (List.hd (List.tl (List.tl ([]) ;;))) ;;` : `int list = []`

### `let rec fromTo (m:int) (n:int) = (* The numbers from m to n *)`

- `if n < m then [] else m :: fromTo (m+1) n;;`:

- `fromTo 9 18 ;;` : `int list = [9; 10; 11; 12; 13; 14; 15; 16; 17; 18]`

**Some recursive functions that generate lists**

### `List.tl` (pronounced "tail")

- `List.tl ([]) ;;` : `int list = []`

**List.hd** (pronounced "head") returns the first element.

- `List.hd ([]) ;;` : `int = 0`
Likemostprogramminglanguages,OCamlincludesamechanismforgrouping
collectionsofdefinitionsinmodules.

Forexample,thefunctionListhdandListtlprovidesListinterface.

```
let rec snoc l x =
  if l = [] then x :: []
  else List.hd l :: snoc (List.tl l) x
```

```
# snoc [5;4;3;2] 1
val it : int list = [5;4;3;2;1]
```

```
let listSum (l:int list) =
  match l with
  | [] -> 0
  | x :: y -> x + listSum y
# listSum [5;4;3;2;1]
val it : int = 15
```
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 that matches any value.

**Complex Patterns**

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

```
#let silly(l:int list)=
  match l with
  [_;_;_]->"threeelementslong"
|_::x::y::_::_::rest->if x>y then "foo" else "bar"
|_->"dunno";;
```

```
valsilly:int list->string=<fun>
```

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

**Polymorphism**

What types should we give to the parameter \( l \) below?

```
#let rec length(l:???)=
  match l with
  []->0
|_::y->1+length y;;
```

It doesn't matter what type of objects are stored in the list, we could make it `int list` or `bool list` and OCaml would not complain. However, if we choose one of these types, we would not be able to apply `length` to the other.

```
if list or bool list OCaml would not complain. However, if we choose
```

```
OCaml lets us use a `??` (pronounced "alpha") standing
```

```
```

OCaml lets us use a `??` (pronounced "alpha") standing
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Tuples are not lists

Tuples and Pattern Matching

An implementation of split

```ocaml
val split : char list -> char list list
let split (l:char list) = loop [] l

let loop : (char list -> char list) -> char list -> char list list
  = let rec loop (r:char list) (l:char list) =
    match l with
    | [] -> r
    | c::ls -> loop (r @ [c]) ls
    | c::ls -> loop (r @ [c]) ls

let split = loop []
```

Example: Finding words

Suppose we want to take a list of characters and return a list of lists of words from the original list.

```ocaml
# 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';'';

(Notethatcharacterconstantsarewrittenwithsinglequotes.)
```

An implementation of split

```ocaml
val split : char list -> char list list
let split (l:char list) = loop [] l

let loop : (char list -> char list) -> char list -> char list list
  = let rec loop (r:char list) (l:char list) =
    match l with
    | [] -> r
    | c::ls -> loop (r @ [c]) ls
    | c::ls -> loop (r @ [c]) ls
    | c::ls -> loop (r @ [c]) ls

let split = loop []
```

This expression has type string but is here used with type list

```ocaml
let last : string list = "Fastest"
let last = (let last = "Fastest"
  in last)
```

This expression has type string but is here used with type list

```ocaml
let last = (let last = ":Last"
  in last)
```

Please do not confuse these.
Aside: Local function definitions

The loop function is completely local to split: there is no reason for anybody else to see it. It is good style in OCaml to write such definitions as local bindings:

```ocaml
let rec loop (w:charlist)(l:charlist) =
  match l with
  | [] -> w
  | (''::ls) -> w::(loop [] ls)
  | (c::ls) -> loop (w@[c]) ls
in loop [] l;;
```

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

```ocaml
#let...in...;;
```

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

```ocaml
let... in
```

Basic Exceptions

OCaml's exception mechanism is roughly similar to that found in, for example, Java. Naturally, exceptions can also be cancelled within a program (using the

```ocaml
# exception Bad
```

Now, encountering raise Bad will immediately terminate evaluation and

```ocaml
#let...in...
```

We begin by defining an exception:

```ocaml
#exception Bad
```

In general, anylet definition that can appear at the top level:

```ocaml
#let...;;
```

Naturally, exceptions can also be cancelled within a program (using the

```ocaml
# exception Bad
```

We begin by defining an exception:

```ocaml
#exception Bad
```

Now, encountering raise Bad will immediately terminate evaluation and

```ocaml
#let...in...
```

We begin by defining an exception:

```ocaml
#exception Bad
```

In general, anylet definition that can appear at the top level:

```ocaml
#let...;;
```

Naturally, exceptions can also be cancelled within a program (using the

```ocaml
# exception Bad
```

We begin by defining an exception:

```ocaml
#exception Bad
```

Now, encountering raise Bad will immediately terminate evaluation and

```ocaml
#let...in...
```

We begin by defining an exception:

```ocaml
#exception Bad
```
Data Types

We have seen a number of data types:

- `int`
- `bool`
- `string`
- `char`
- `list` of `st``uples

OCaml has a few other built-in data types — in particular, `float`, `int`, etc.

We will see new data types as elements of the type system.

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

We can improve matters by defining a new type:

```ocaml
# types 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.

Data Types

One can also create complex new data types:

```ocaml
# type complex = Complex of float * float ;;
```

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

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

Sowecanuseconstructorslike `Square` both as functions and as patterns.

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

Continuing, we can define a datatype 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):circle)=3.14159*.r*.r;;
# let centerCoords(Circle(x,y,_):circle)=(x,y);;
# areaOfCircle c ;;
- : float = 12.56636
```

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

```ocaml
# areaOfSquare(c);;
Error: 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 these shapes. Since we want to execute a function for each type of shape, we obviously want to have a function that can match against each type of shape.

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

To do this, we can create a type that can have *more than one form*.

```
# 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);;
val it : shape = Square (1.000000, 2.000000, 3.000000)
```

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

**Pattern matching**

When matching with the function header,

```
val square : int -> int
```

these functions can be written in a little more compactly by combining the pattern matching with the function header:

```
val square : int -> int
```

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

```
# let bottomLeftCoordinates (square x' y') = (x',y');
# let areaBottomLeftCoordinates (square x y r) = (x,y);;
```

These functions can be written a little more concisely by combining the pattern matching with the function header:
We can also write functions that do the right thing on all forms of a variant type. Again we use pattern matching:

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

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

A "heterogeneous" list:

```ocaml
# 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)];
```

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. Mixed-mode arithmetic follows exactly the same pattern.

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

```ocaml
# let add (r1:num) (r2:num) =
  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);;
```

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

```ocaml
# let mult (r1:num) (r2:num) =
  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);;
```

```ocaml
# let mult (Int 3) (Float 4.5);;
- : num = Float 13.5
```

A Data Type for Optional Values

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

However, this isn't quite enough. When implementing a given string phone in the directory, we should look up return.

where directory is a yet to be decided type that will be used to represent the phone.

We expect to have a function lookup whose type is

Suppose we are implementing a simple lookup function for a phone book. We want to give it a string and get back a number (say an integer).

Some Higher-Level Mixed-Mode Functions
Enumerations

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

```
#type color = Red | Yellow | Green;;
#let next (c : color) =
    match c with
    | Green -> Yellow
    | Yellow -> Red
    | Red -> Green;;
#type day = Sunday | Monday | Tuesday | Wednesday
          | Thursday | Friday | Saturday;;
#let weekend (d : day) =
    match d with
    | Saturday -> true
    | Sunday -> true
    | _ -> false;;
```

Recursive Types

Consider the tiny language of arithmetic expressions defined by the following grammar:

```
exp ::= number
     | exp + exp
     | exp - exp
     | exp * exp
```

We can translate this grammar directly into a data type definition:

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

Notes:

- This data type (like the original grammar) is recursive.
- The type `ast` represents abstract syntax trees, which capture the underlying tree structure of expressions, suppressing surface details such as parentheses.
- The type `ast` represents abstract syntax trees, which capture the underlying tree structure of expressions, suppressing surface details such as parentheses.

We can translate this grammar directly into a `let` expression definition:

```
let rec evaluate (e : ast) =
    match e with
    | Anum n -> n
    | APlus (e1, e2) -> evaluate e1 + evaluate e2
    | AMinus (e1, e2) -> evaluate e1 - evaluate e2
    | ATimes (e1, e2) -> evaluate e1 * evaluate e2
```

```
let rec evaluate (e : ast) =
    match e with
    | Anum n -> n
    | APlus (e1, e2) -> evaluate e1 + evaluate e2
    | AMinus (e1, e2) -> evaluate e1 - evaluate e2
    | ATimes (e1, e2) -> evaluate e1 * evaluate e2
```

A Boolean Data Type

A Boolean data type can be used to replace the built-in boolean values `true` and `false`. We use different names to avoid confusion between our booleans and the built-in ones:

```
#type myBool = False | True;;
#let myNot (b : myBool) =
    match b with
    | False -> True
    | True -> False;;
#let myAnd (b1 : myBool) (b2 : myBool) =
    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 the built-in `&&` and `!` operators:

```
match (b1, b2) with
    | (True, True) -> True
    | (True, False) -> False
    | (False, True) -> False
    | (False, False) -> False
```

Recursive Types

Consider the tiny language of arithmetic expressions defined by the following grammar:

```
exp ::= number
     | exp + exp
     | exp - exp
     | exp * exp
```

We can translate this grammar directly into a data type definition:

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

Notes:

- This data type (like the original grammar) is recursive.
- The type `ast` represents abstract syntax trees, which capture the underlying tree structure of expressions, suppressing surface details such as parentheses.

```
let rec evaluate (e : ast) =
    match e with
    | Anum n -> n
    | APlus (e1, e2) -> evaluate e1 + evaluate e2
    | AMinus (e1, e2) -> evaluate e1 - evaluate e2
    | ATimes (e1, e2) -> evaluate e1 * evaluate e2
```

```
let rec evaluate (e : ast) =
    match e with
    | Anum n -> n
    | APlus (e1, e2) -> evaluate e1 + evaluate e2
    | AMinus (e1, e2) -> evaluate e1 - evaluate e2
    | ATimes (e1, e2) -> evaluate e1 * evaluate e2
```

A Boolean Data Type

A Boolean data type can be used to replace the built-in boolean values `true` and `false`. We use different names to avoid confusion between our booleans and the built-in ones:

```
#type myBool = False | True;;
#let myNot (b : myBool) =
    match b with
    | False -> True
    | True -> False;;
#let myAnd (b1 : myBool) (b2 : myBool) =
    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 the built-in `&&` and `!` operators:

```
match (b1, b2) with
    | (True, True) -> True
    | (True, False) -> False
    | (False, True) -> False
    | (False, False) -> False
```

Recursive Types

Consider the tiny language of arithmetic expressions defined by the following grammar:

```
exp ::= number
     | exp + exp
     | exp - exp
     | exp * exp
```

We can translate this grammar directly into a data type definition:

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

Notes:

- This data type (like the original grammar) is recursive.
- The type `ast` represents abstract syntax trees, which capture the underlying tree structure of expressions, suppressing surface details such as parentheses.

```
let rec evaluate (e : ast) =
    match e with
    | Anum n -> n
    | APlus (e1, e2) -> evaluate e1 + evaluate e2
    | AMinus (e1, e2) -> evaluate e1 - evaluate e2
    | ATimes (e1, e2) -> evaluate e1 * evaluate e2
```

```
let rec evaluate (e : ast) =
    match e with
    | Anum n -> n
    | APlus (e1, e2) -> evaluate e1 + evaluate e2
    | AMinus (e1, e2) -> evaluate e1 - evaluate e2
    | ATimes (e1, e2) -> evaluate e1 * evaluate e2
```

A Boolean Data Type

A Boolean data type can be used to replace the built-in boolean values `true` and `false`. We use different names to avoid confusion between our booleans and the built-in ones:

```
#type myBool = False | True;;
#let myNot (b : myBool) =
    match b with
    | False -> True
    | True -> False;;
#let myAnd (b1 : myBool) (b2 : myBool) =
    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 the built-in `&&` and `!` operators:

```
match (b1, b2) with
    | (True, True) -> True
    | (True, False) -> False
    | (False, True) -> False
    | (False, False) -> False
```

Recursive Types

Consider the tiny language of arithmetic expressions defined by the following grammar:

```
exp ::= number
     | exp + exp
     | exp - exp
     | exp * exp
```

We can translate this grammar directly into a data type definition:

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

Notes:

- This data type (like the original grammar) is recursive.
- The type `ast` represents abstract syntax trees, which capture the underlying tree structure of expressions, suppressing surface details such as parentheses.

```
let rec evaluate (e : ast) =
    match e with
    | Anum n -> n
    | APlus (e1, e2) -> evaluate e1 + evaluate e2
    | AMinus (e1, e2) -> evaluate e1 - evaluate e2
    | ATimes (e1, e2) -> evaluate e1 * evaluate e2
```

```
let rec evaluate (e : ast) =
    match e with
    | Anum n -> n
    | APlus (e1, e2) -> evaluate e1 + evaluate e2
    | AMinus (e1, e2) -> evaluate e1 - evaluate e2
    | ATimes (e1, e2) -> evaluate e1 * evaluate e2
```

A Boolean Data Type

A Boolean data type can be used to replace the built-in boolean values `true` and `false`. We use different names to avoid confusion between our booleans and the built-in ones:

```
#type myBool = False | True;;
#let myNot (b : myBool) =
    match b with
    | False -> True
    | True -> False;;
#let myAnd (b1 : myBool) (b2 : myBool) =
    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 the built-in `&&` and `!` operators:

```
match (b1, b2) with
    | (True, True) -> True
    | (True, False) -> False
    | (False, True) -> False
    | (False, False) -> False
```
A evaluator for expressions

Goal: write an evaluator for these expressions.

val eval : ast -> int = <fun>

# eval (ATimes (APlus (ANum12, ANum340), ANum5));;
- : int = 1760

The solution uses a recursive function plus a pattern match.

let rec eval (e : ast) =
  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

CIS 500, Programming with OCaml

A final example

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

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

# letrec interleave (l1:'a list) (l2:'a list) =
  match l1, l2 with
  | [], [], -> []
  | x::xs, y::ys -> x::y::(interleave xs ys)
  | _ -> raise Bad

# interleave [1;2;3] [4;5;6];;
- : int list = [1;4;2;5;3;6]
Now suppose that we want to calculate all the possible interleavings of two lists.

In other words, the type of the function in a polymorphic version of last is:

\[ \text{last : } 'a \text{ list} \rightarrow 'a \text{ list} \]

while any type that we like for the type of the function last in a polymorphic version is:

\[ \text{last : } 'a \text{ list list} \rightarrow 'a \text{ list} \]

Note that the type of the function last in a polymorphic version is not just any different type of function, but a function of type 'a list 'a list -> 'a list. This is a many different types of functions. A polymorphic function is a function that can be applied to any type of input.
Polarimorph renv

#let retraux(l:'alist)(res:'alist)=
  if l=[] then res
  else retraux(List.tl l)(List.hd l::res);;
val retraux:'alist->'alist->'alist =<fun>

#let rev(l:'alist)=retraux l[];;
val rev:'alist->alist =<fun>
# rev["cat";"in";"the";"hat"];;
-: string list = ["hat";"the";"in";"cat"]
# rev[false;true];;
-: bool list = [true;false]

Polymorphic

What is the type of repeat?

# repeat 7 12;;
-: int list = [7;7;7;7;7;7;7;7;7;7;7;7]
# repeat true 3;;
-: bool list = [true;true;true]
# repeat [6;7] 4;;
-: int list list = [[6;7];[6;7];[6;7];[6;7]]

OCaml has a predefined function List.map that takes a function \( f \) and a list \( l \) and produces another list by applying \( f \) to each element of \( l \). We'll soon see how to define List.map, but first let's look at some examples and produce another list by applying \( f \) to each element of \( l \). We'll see how to define List.map, but first let's look at some examples.

CIS 500, Programming with OCaml

Palindrome

OCaml has a predefined function List.map that takes a function \( f \) and a list \( l \) and produces another list by applying \( f \) to each element of \( l \). We'll soon see how to define List.map, but first let's look at some examples and produce another list by applying \( f \) to each element of \( l \). We'll see how to define List.map, but first let's look at some examples.

List.map = "map": "apply-to-each"
An interesting feature of `List.map` is its first argument is itself a function. For this reason, we call `List.map` a higher-order function.

Natural uses for higher-order functions are frequent in programming. One of OCaml's strengths is that it makes higher-order functions very easy to work with. In other languages such as Java, higher-order functions can be (and often are) simulated using objects.

More on `map`
Similarly, we can define our own `filter` that behaves like the same as

```
let rec filter (p:'a->bool) (l:'alist)=
if l=[] then []
elseif p(List.hd l) then List.hd l::filter p (List.tl l)
else filter p (List.tl l)
```

(Assuming `List` is the data structure similar to `List` in OCaml.)

---

**Approaches to Typing**

A strongly typed language prevents programs from accessing private data, corrupting memory, crashing the machine, etc. A weakly typed language does not. A statically typed language performs type consistency checks at when programs are first entered. A dynamically typed language delays these checks until programs are executed.

<table>
<thead>
<tr>
<th>Strong</th>
<th>Weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>C++, ML, ADA, Java</td>
<td>Lisp, Scheme</td>
</tr>
</tbody>
</table>

Strictly speaking, Java should be called “mostly static.”

---

**Practice with Types**

What are the types of the following functions?

```
let f (x:int)= x+1
let f' (x:int)= x+1
let f (x:int)= [x]
let f' (x:int)= [x]
let f (x:int)= x
let f' (x:int)= x
let g (x:int)= x::[1.0]
let f' (x:int)= x::[]
let f (x:int)= 1::x
```

Also:

```
let rec f (x:int)=
if (List.tl x)=[] then x
else f (List.tl x)
```

---

Another:

```
let rec g t (l:int list) =
if []=l then ()
else match List.rev t with
  | [] -> g [] t
  | h::tl -> let rec x =
    if List.hd l then x
    else List.hd l::tl
  x
```

(which is a variant of the `filter` function)
Aside: Polymorphism

The polymorphism in ML that arises from type parameters is an example of parametric polymorphism. Different languages support generic programming in different ways...

OCaml supports parametric polymorphism in a very general way, and also supports subtyping (though we shall not get to see this aspect of OCaml, its support for subtyping is what distinguishes it from other dialects of ML). It does not allow overloading.

Java provides a subtyping, as well as moderately powerful overloading, but no parametric polymorphism. (Various Java extensions with parametric polymorphism are under discussion.)

Confusingly, the bare term "polymorphism" is used to refer to parametric polymorphism in the ML community and for subtype polymorphism in the Java community.

OCaml supports parametric polymorphism in a very general way, and also supports subtyping (though we shall not get to see this aspect of OCaml, its support for subtyping is what distinguishes it from other dialects of ML). It does not allow overloading.

Java provides a subtyping, as well as moderately powerful overloading, but no parametric polymorphism. (Various Java extensions with parametric polymorphism are under discussion.)

Confusingly, the bare term "polymorphism" is used to refer to parametric polymorphism in the ML community and for subtype polymorphism in the Java community.

OCaml supports parametric polymorphism in a very general way, and also supports subtyping (though we shall not get to see this aspect of OCaml, its support for subtyping is what distinguishes it from other dialects of ML). It does not allow overloading.

Java provides a subtyping, as well as moderately powerful overloading, but no parametric polymorphism. (Various Java extensions with parametric polymorphism are under discussion.)

Confusingly, the bare term "polymorphism" is used to refer to parametric polymorphism in the ML community and for subtype polymorphism in the Java community.