CSE 120/130

Introduction to

Programming Languages and Techniques

Fall 2000

Handout 4

Ordering, New types, Searching

Case study: Sorting other data structures

Suppose we have a *database* of information about people:

Last Name	First Name	Age	Phone
Smith	Carol	43	555-1234
Upper	Beth	31	567-8900
Turner	Drew	22	552-4321
Smith	Adam	42	555-1234

We can encode this information as a list of 4-tuples:

```
# let db = [
    ("Smith", "Carol", 43, 5551234); ("Upper", "Beth", 31, 5678900);
    ("Turner", "Drew", 22, 5524321); ("Smith", "Adam", 42, 5551234) ];;
```

The type of `db` is `(string * string * int * int) list`.

More pattern matching syntax

If you look at some of the previous examples you will see function definitions of the form

```
let f p = match p with (x,y) -> ...
```

OCaml allows you to abbreviate this to

```
let f (x,y) = ...
```

Similarly you can write

```
let (x,y) =  ... ;;
```

or

```
let (x,y) =  ...  in ...
```

However one cannot use `let` with multiple matching clauses; and if you say

```
let f (x::y) = ...
```

you will get a “not exhaustive” error.

Comparing people

So, for the time being, the only patterns we should use in `let ...` expressions are tuple patterns.

```
# let lastname  ( x , _ , _ , _ )    = x;;
# let firstname ( _ , x , _ , _ )    = x;;
# let age       ( _ , _ , x , _ )    = x;;
# let phone     ( _ , _ , _ , x )    = x;;

# let complast  p1 p2  = (lastname p1)  <= (lastname p2);;
# let compfirst p1 p2  = (firstname p1) <= (firstname p2);;
# let compage   p1 p2  = (age p1)       <= (age p2);;
```

What are the types of these functions?

Sorting lists of people

Recall our packaged version of top-down `mergesort`:

```
let rec mergesort comp l =
  match l with
    [] -> []
  | [x] -> [x]
  | l ->
    let rec merge l =  (* code for merge... *) in
    let rec split l =
      match l with
        [] -> [], []
      | [x] -> [x], []
      | h1 :: h2 :: t ->
          let l1,l2 = split t in
            h1::l1, h2 ::l2  in
    let l1,l2 = split l in
       merge(mergesort comp l1, mergesort comp l2);;
```



```
# mergesort complast db;;
- : (string * string * int * int) list =
["Smith", "Carol", 43, 5551234; "Smith", "Adam", 42, 5551234;
 "Turner", "Drew", 22, 5524321; "Upper", "Beth", 31, 5678900]

# mergesort compfirst db;;
- : (string * string * int * int) list =
["Smith", "Adam", 42, 5551234; "Upper", "Beth", 31, 5678900;
 "Smith", "Carol", 43, 5551234; "Turner", "Drew", 22, 5524321]

#  mergesort compage db;;
- : (string * string * int * int) list =
["Turner", "Drew", 22, 5524321; "Upper", "Beth", 31, 5678900;
 "Smith", "Adam", 42, 5551234; "Smith", "Carol", 43, 5551234]
```

Sorting on Multiple Keys

Suppose we want to put the database in alphabetical order by *both* first and last names.

We should be able to accomplish this by...

1. first sorting according to first names, and
2. then sorting according to last names (keeping the ordering of first names).

Smith	Carol	⇒	Smith	Adam	⇒	Smith	Adam
Upper	Beth		Upper	Beth		Smith	Carol
Turner	Drew		Smith	Carol		Turner	Drew
Smith	Adam		Turner	Drew		Upper	Beth

In our output we want the list to be sorted primarily on last name, with the first name only being used to order records with the same last name. The “prime” field is sometimes called the *major key* and the secondary field is called the *minor* key (nothing musical).

To achieve the required order we sort *first* on the minor key and then on the major key.

Will this work?

```
# let alphasort l =
    mergesort complast (mergesort compfirst l);;

# alphasort db;;
- : (string * string * int * int) list =
["Smith", "Adam", 42, 5551234; "Smith", "Carol", 43, 5551234;
 "Turner", "Drew", 22, 5524321; "Upper", "Beth", 31, 5678900]
```

This looks good so far, but...

```
# let db1 = [
    ("X", "A", 0, 0);
    ("X", "B", 0, 0);
    ("X", "C", 0, 0);
    ("X", "D", 0, 0);
    ("Y", "A", 0, 0);
    ("Y", "B", 0, 0);
    ("Y", "C", 0, 0);
    ("Y", "D", 0, 0) ];;

# alphasort db1;;
- : (string * string * int * int) list =
["X", "A", 0, 0; "X", "C", 0, 0; "X", "B", 0, 0; "X", "D", 0, 0;
 "Y", "A", 0, 0; "Y", "C", 0, 0; "Y", "B", 0, 0; "Y", "D", 0, 0]
```

What happened??

Stability of Sorting Algorithms

Our implementation of the `mergesort` algorithm is *unstable*: records with equal values of the field we are comparing are not kept in their original order.

The culprit is the `split` function:

```
# let rec split l =
    match l with
      [] -> [], []
    | [x] -> [x], []
    | h1 :: h2 :: t ->
        let l1,l2 = split t  in  h1::l1, h2 :: l2 ;;

# split [1; 2; 3; 4; 5; 6];;
- : int list * int list = [1; 3; 5], [2; 4; 6]

# split db1;;
["X", "A", 0, 0; "X", "C", 0, 0; "Y", "A", 0, 0; "Y", "C", 0, 0],
["X", "B", 0, 0; "X", "D", 0, 0; "Y", "B", 0, 0; "Y", "D", 0, 0]
```

A better `split`

```
# let rec take n l =
    if n = 0 then []
    else List.hd l :: take (n-1) (List.tl l);;

# let rec drop n l =
    if n = 0 then l
    else drop (n-1) (List.tl l);;

# let rec split l =
    let n = (List.length l) / 2 in
    (take n l, drop n l);;

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


A stable `mergesort`

Substituting our new `split` into the definition of `mergesort` yields a stable algorithm that can be used to sort on multiple fields:

```
# alphasort db1;;
- : (string * string * int * int) list =
["X", "A", 0, 0; "X", "B", 0, 0; "X", "C", 0, 0; "X", "D", 0, 0;
 "Y", "A", 0, 0; "Y", "B", 0, 0; "Y", "C", 0, 0; "Y", "D", 0, 0]
```

Lexicographic Ordering

[*lexicographic*: from *lexicography*, the art or practice of writing dictionaries—from Greek *lexicon*, dictionary—from *lexis*, word or phrase]

We are all quite familiar with the ordering we use to put words in a dictionary or names in a telephone directory, but computing that ordering is not trivial.

Consider the words

```
c  a  p
c  a  t
c  a  t  c  h
```

We place `cap` before `cat` because, reading from left to right, the letters on which they first disagree are ordered alphabetically.

We place `cat` before `catch` because `cat` is an initial segment (sometimes called a *prefix*) of `catch`.

Lexicographic ordering is rather subtle. How many character strings are there – in lexicographic order – between `cap` and `cat`?

An implementation

```
# let rec lexord l1 l2 =
    match (l1,l2) with
      [], _ -> true     (* true when both lists are empty *)
    | _, [] -> false
    | h1::t1, h2::t2 ->
        if h1<= h2 then
          if h2 <= h1 then lexord t1 t2 (* in this case h1=h2 *)
          else true
        else false;;

val lexord : 'a list -> 'a list -> bool = <fun>
```

Just as `<=` is a *total* ordering on any type `'a`, `lexord` is a total ordering on the type `'a list`.

A better implementation

There are some problems with our implementation. In particular, when used on lists of characters, the ordering is “case sensitive” – it will place `Cato` before `bat`.

OCaml has a built-in function `Char.uppercase`, but it is worth noting that we can write our own function which is independent of the character coding system used – provided the uppercase and lowercase characters are represented as consecutive numbers.

```
# let upperCase c =
    let v1 = Char.code c - Char.code 'a' in
      if v1 >=0 && v1 <= Char.code 'z'
      then Char.chr(Char.code 'A'+ v1 )
      else c;;

# let upperComp c1 c2 = upperCase c1 <= upperCase c2;;
```


The ASCII character set

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

Generalizing lexord

Our original `lexord` is too specific. It used the built-in OCaml comparison operation. We need to change it for our new comparison function, so we might as well generalize it:

```
# let rec genLexord comp l1 l2 =  (* comp must be a total order *)
    match (l1,l2) with
      [], _ -> true
    | _, [] -> false
    | h1::t1, h2::t2 ->
       if comp h1 h2 then
          if comp h2 h1 then genLexord comp t1 t2
          else true
      else false;;

val genLexord : ('a -> 'a -> bool) -> 'a list -> 'a list -> bool = <fun>
```

Can you rewrite the conditional using `&&` and `||`?

The completed function

```
# let lexord s1 s2 = genLexord upperComp (explode s1) (explode s2);;

# lexord "cat" "cap";;
- : bool = false

# lexord "Cato" "bat";;
- : bool = false
```

Writing a more general lexicographic ordering function has its uses. What does the following function do?

```
# let whatord l1 l2 = genLexord lexord l1 l2;;
```

Bignums

On most present-day computers the size of an integer is limited to what will fit in a 32-bit word, but it's possible to build arbitrarily large numbers by representing them exactly as we do when we write them—as a sequence of digits.

We'll represent a big number as a list of little numbers between 0 and 9. First recall how one does (or did before calculators) long addition:

$$\begin{array}{cccccccc} & 6^{1} & 7 & 3 & 4 & 5^{1} & 6^{1} & 7 \\ + & & 8 & 6 & 4 & 3 & 3 & 9 \\ \hline & 7 & 5 & 9 & 8 & 9 & 0 & 6 \end{array}$$

Note that the method starts at the *least significant* digits. Therefore we'll represent our big numbers as lists with the least significant digits first (i.e., leftmost).

To keep things simple, we'll work only with *nonnegative* numbers.

Conversion to bignums

```
(* Convert int to big *)
# let rec toBig n =
    if n = 0 then []
    else n mod 10 :: toBig (n / 10);;

#  toBig 35678;;
- : int list = [8; 7; 6; 5; 3]
```

Conversion from bignums

```
# let rec fromBig l = (* Convert big to int *)
    match l with
      [] -> 0
    | h::t -> (10 * fromBig t) + h;;

# fromBig([8; 7; 6; 5; 3]);;
- : int = 35678

# fromBig [8; 7; 6; 5; 3; 8; 7; 6; 5; 3; 8; 7; 6; 5; 3];;
- : int = 634556958
```

In the last case, the integer arithmetic operations have “overflowed.”

Formatting Bignums

```
# let rec big2string l =
    match l with
      [] -> big2string [0]
    | l -> let dig2char n = Char.chr(n + Char.code '0')
           in implode (List.rev (List.map dig2char l));;

# big2string [8; 7; 6; 5; 3; 8; 7; 6; 5; 3; 8; 7; 6; 5; 3];;
- : string = "356783567835678"
```

Note that we have to reverse the list of characters.

The functions `Char.chr: int -> char` and `Char.code: char ->int` map characters to integers in the range 0..255 and back. A commonly used encoding is ascii—the A(merican) S(tandard) C(ode) for I(nformation) I(nterchange). However `big2string` only assumes that the digits are mapped onto consecutive numbers, so it should work for other character representations as well.

The code for long addition

```
# let bigAdd l1 l2 =
    let rec aux w =
      match w with
        (0, l1, []) -> l1
      | (c, l1, []) -> aux (0, l1, [c])
      | (c, [], l2) -> aux (0, [c], l2)
      | (c, h1::t1, h2::t2) ->
          let s = c + h1 + h2 in
          if s > 9 then (s-10) :: aux(1,t1,t2)
                   else s :: aux(0,t1,t2) in
    aux(0,l1,l2);;
```

The work is done by the function `aux`. The parameter `c` is the "carry" and is 0 or 1.

Compare

```
# bigAdd [7; 6; 5; 4; 3; 7; 6] [9; 3; 3; 4; 6; 8];;
- : int list = [6; 0; 9; 8; 9; 5; 7]
```

with

$$\begin{array}{cccccccc} & 6^1 & 7 & 3 & 4 & 5^1 & 6^1 & 7 \\ & & 8 & 6 & 4 & 3 & 3 & 9 \\ \hline 7 & 5 & 9 & 8 & 9 & 0 & 6 & \end{array}$$

Multiplication by a single digit

```
# let digMul d l =
    if d = 0 then []
    else
      let rec aux n ll =
        match ll with
          [] -> []
        | h::t -> let s = n + d * h in
                  (s mod 10) :: aux (s / 10) t  in
      aux 0 l;;
```

The parameter `n` of `aux` is the "carry" (in the range 0..9). Note that `d` is global to `aux`.

What simple function multiplies a big number by 10?

Multiplication

Again, this is how we learned to do it in school:

```
# let rec bigMul l1 l2 =
    match l1 with
      [] -> []
    | h::t -> bigAdd (digMul h l2) (0 :: bigMul t l2);;
```

A test:

```
# big2string  (bigMul(toBig 123456) (toBig 123456));;
- : string = "15241383936"
```

This could not have been computed with Ocaml (or any 32 bit) integer arithmetic.

Two more useful functions:

```
# let rec eq0 l =
    match l with
      [] -> true
    | 0::t -> eq0 t
    |  _ -> false;;
```

Note that initial (i.e. trailing) 0's are redundant.

Representing negative numbers adds some complexity. Here is a "subtract one" function that leaves 0 as 0.

```
# let rec sub1 l =
    match l with
      [] -> []
    | (0::t) -> 9::sub1 t
    | (h::t) -> (h-1)::t;;
```


An example

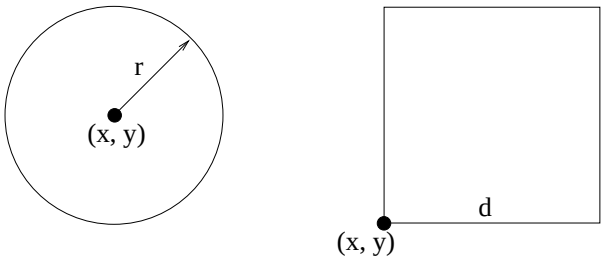
```
# let rec fact n = if eq0 n then [1] else bigMul n (fact (sub1 n));;

# big2string(fact(toBig 256));;
- : string =
"26260659216323446215164912407747281732306847420484525177895709509518219
86258504820796498034223101315840564396133568748565215901765160070464171 8
02707725338158905196380536841053564407787734960317185721259630435239358 0
76410912003387536187610888809553772446782799468865176647760650622462779 8
84726305831512826444459189611128692876042997642477873428504129027283353 6
00000000000000000000000000000000000000000000000000000000000000000000000"
```

Building New 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.

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 *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);;
```

Actually, we have seen several instances of this already. Three slides ago, we wrote

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

instead of:

```
# let areaOfSquare z =
    match z with
      (_,_,d) -> d *. d;;
```

Similarly,

```
# let (x,y) = (3,4) in x+y;;
```

is just shorthand for

```
# let z = (3,4) in
    match z with
      (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, 2, 3)
```

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]
```

A "bounding box" – the smallest enclosing rectangle – is useful in graphics. In our simplified case the bounding box is always a square:

```
# let boundingBox s =
    match s with
      Circle (x, y, r) -> Square(x -. r, y -. r, 2.0 *. r)
    | s -> s;;
```

What is the type of `boundingBox` ?

A challenge: Write a function to determine whether two shapes overlap. (This is *very* important in graphics.)

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.

We can represent such a type in OCaml as follows:

```
# 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 squareRoot n =
    match n with
       Int i -> Float (sqrt (float i))
     | Float r -> Float(sqrt  r);;
```

Challenge: write `squareRoot` so that it returns an integer when this is a sensible thing to do.


```
# 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
```

Does `fact` work for all inputs of type `num`?

An Option Data Type

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 general technique is based on the following data type:

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

(Can you think of any other approaches, using only what we already know about OCaml?)

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
```

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;;
```

In what way does the behavior of `myAnd` differ from `&&`?

Binary Search Trees

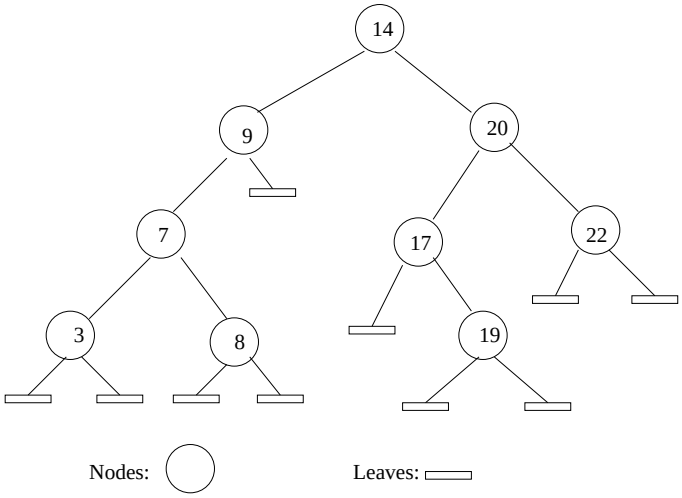
Search trees come in all shapes and flavors in computer science and are used in a wide variety of applications. For example, whenever you look something up in a database, you are almost certainly using a search tree. We will deal with a particularly simple form, called *binary search trees*.

Our first application will be representing sets of integers. We have already seen how to implement sets using lists, but we can build more efficient data structures. (Using a list representation, how long does it take to determine whether a given integer is in a list?)

Here is the data type for a binary tree:

```
# type tree = Leaf
            | Node of (tree * int * tree);;
```

Each `Node` has an associated value—in this case an integer—plus "left" and "right" subtrees. A `Leaf` has no associated value and no subtrees.



We shall maintain the *invariant* that the value at each node is greater than any of the values in its left subtree and less than any of the values in its right subtree. (Observe that this property holds for *every* node in the tree shown above.)

The following sequence of declarations will build this tree:

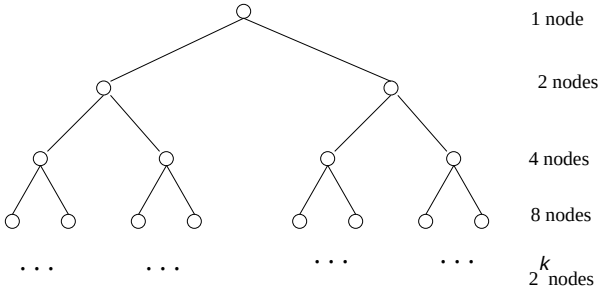
```
# let n3  = Node(Leaf, 3, Leaf);;
# let n7  = Node(Leaf, 7, Leaf);;
# let n8  = Node(n7, 8, Leaf);;
# let n6  = Node(n3, 6, n8);;
# let n12 = Node(Leaf, 12, Leaf);;
# let n9  = Node(n6, 9, n12);;
# let n19 = Node(Leaf, 19, Leaf);;
# let n17 = Node(Leaf, 17, n19);;
# let n22 = Node(Leaf, 22, Leaf);;
# let n20 = Node(n17, 20, n22);;
# let n14 = Node(n9, 14, n20);;
```


Functions on Binary Search Trees

The most useful function on trees is one that tests whether a given integer appears underneath some node:

```
# let rec lookup x n =
    match n with
      Leaf -> false
    | Node(l,v,r) ->
        if x = v then true
        else if x < v then lookup x l
        else lookup x  r;;

# lookup 3 n14;;
- : bool = true

# lookup 2 n14;;
- : bool = false
```

The ordering is used to cut down the number of nodes that must be inspected.



In a full binary search tree of depth k there are $2^{k+1} - 1$ nodes. This means that if we put n numbers in a tree, and the tree is (roughly) full, it will have depth $\log_2 n$. Therefore, `lookup` requires $\log_2 n$ work in this case. Compare this with the cost of the same function when we use a list representation.

"Roughly full" trees, where no leaf is more than 1 level deeper than any other, are commonly called *balanced*.

Inserting numbers

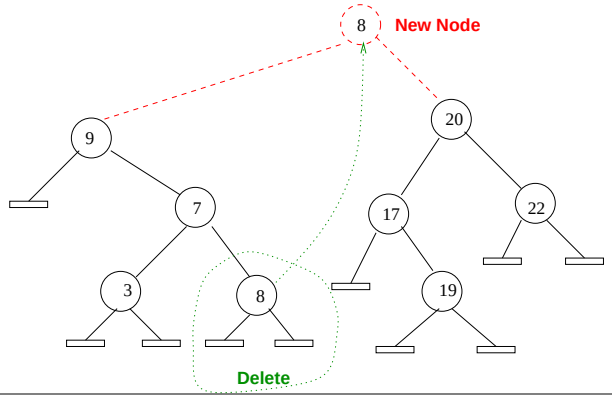
Insertion into a binary search tree is straightforward. We simply find the leaf where the number "belongs" and replace it by a new node:

```
# let rec insert x n =
    match n with
      Leaf -> Node(Leaf,x,Leaf)
    | Node(l,v,r) ->
        if x = v then
          Node(l,v,r)              (* already present *)
        else if x < v then
          Node(insert x l, v, r)   (* put in left subtree *)
        else
          Node(l, v, insert x r)   (* put in right subtree *);;

# lookup 5 (insert 88 (insert 5 n14));;
- : bool = true
```

Deleting from a binary search tree

The deletion operation is trickier. To delete from a tree we search the left or right subtree as appropriate until we find either a leaf or a node with the value to be deleted.

Now, to delete a node with a leaf subtree we simply return the other subtree. If both subtrees are not leaves, we need to "merge" the left and right subtrees of this node by finding some value to serve as the new root (replacing the node we are deleting). One possibility is the *maximum* value in the left sub-tree.



Our first function is therefore one to find the maximum value in a tree:

```
let rec maxVal n =
  match n with
    Node(_, v, Leaf) -> v
  | Node(_, _, r) -> maxVal(r)

# maxVal n14;;
- : int = 22
```

Next a function to delete the *maximum* value from a tree. This is simpler than the general deletion problem.

```
let rec  deleteMax n =
  match n with
    Node(l, v, Leaf) -> l
  | Node(l, v, r) -> Node(l, v, deleteMax r)
```

Why does OCaml "grumble" about these definitions?

Another simple function is a leaf test:

```
let isLeaf n =  match n with Leaf -> true | _ -> false
```

Now we can write the deletion function:

```
let rec delete x n =
  match n with
    Leaf -> Leaf (* it isn't there !*)
  | Node(l,v,r) ->
      if x < v then Node(delete x l, v, r)  (* first easy case *)
      else if x > v then Node(l, v, delete x r) (* second easy case *)
           else if isLeaf (l) then r  (* v = x -- delete v*)
                else Node(deleteMax l, maxVal l, r)
```

Of course, we might "clean up" our delete function by making `maxVal` and `deleteMax` local to `delete`

Maintaining balance

We observed that, for efficiency, we should try to keep our binary search trees balanced. However neither `insert` nor `delete` preserves this property. Consider the function

```
let rec listToTree l =
  match l with
    [] -> Leaf
  | x::y -> insert x (listToTree y)
```

What kind of tree will it build when applied to the list `[1; 2; 3; 4; 5; 6; 7; 8]`?

One can design more sophisticated algorithms for keeping a binary search tree balanced. You'll learn about these in later courses.

Polymorphic data types

We can parameterize data types by a type variable. One such type is already well-known to us:

```
type 'a list = Nil | Cons of 'a * 'a list
```

With this we can write all the basic functions on lists:

```
let hd (Cons(x,_)) = x

let tl (Cons(_,y)) = y
```

What will happen when we evaluate `tl(Nil)` ?

All that is missing from our lists is the syntactic convenience of having an infix `cons` such as `::` and a special notation for reading/displaying lists [e_1, ..., e_n].

The option data type

For convenience, OCaml provides a built-in polymorhic option type:

```
type 'a option = None | Some of 'a
```

This is a simple generalizaton of the `maybe` data type described earlier. Here is another application...

```
datatype 'a option = NONE | SOME of 'a

(* find the roots of a*x*x + b*x + c = 0 *)
let roots (a,b,c) =
  let  v = b *. b -. 4.0 *. a *. c in
    if v < 0.0 then None
    else let s = sqrt v in
           Some((-.b-.s)/.(2.0*.a), (-.b +.s)/.(2.0*.a))


val roots : float * float * float -> (float * float) option = <fun>
```

Using a binary search trees as a dictionary

A more general binary tree data type

```
type ('a,'b) tree =
    Leaf
  | Node of ('a,'b) tree * ('a * 'b) * ('a,'b) tree
```

can be used to build dictionaries. First, we define a generic lookup function (we are not only generalizing the types, but also the comparison function.)

```
let rec lookup comp x t =
  match t with
    Leaf -> None
  | Node(l,(v,w),r) ->
      if comp x v then lookup comp x l
      else if comp v x then lookup comp x r
           else Some w

val lookup: ('a -> 'a -> bool) -> 'a -> ('a, 'b) tree -> 'b option =<fun>
```


To specialize this function to, say, a `string` to `int` dictionary, one would write

```
let stringcomp (s1:string) (s2:string) = s1 <= s2

let find s d = lookup stringcomp s d
```

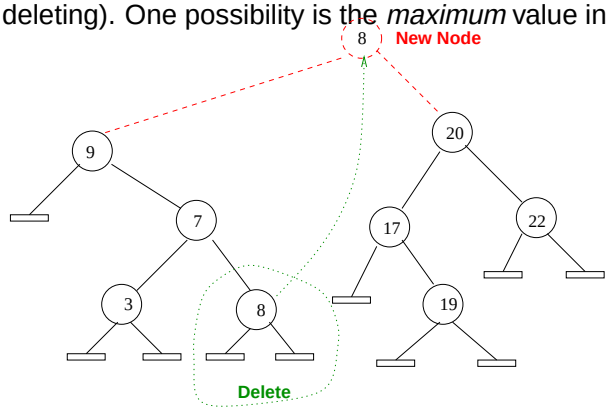
Search trees are extremely important in almost all branches of computer science.

An important generalization of the binary search tree is an n-ary search tree. For example, if n is 30-40 a balanced tree of depth 6 will accommodate 10^9 nodes. What this means in practice is that we can build enormous dictionaries—such as web indexes—that require only a few disk accesses to do a lookup.

Deleting from a binary search tree

The deletion operation is trickier. To delete from a tree we search the left or right subtree as appropriate until we find either a leaf or a node with the value to be deleted.

Now, to delete a node that has a leaf as an immediate subtree, we can simply return the other subtree. If both subtrees are *not* leaves, we need to "merge" the left and right subtrees of this node by finding some value to serve as the new root (replacing the node we are deleting). One possibility is the *maximum* value in the left sub-tree.



The first function we need is one that finds the maximum value in a tree:

```
# let rec maxVal n =
    match n with
      Node(_, v, Leaf) -> v
    | Node(_, _, r) -> maxVal(r);;

# maxVal n14;;
- : int = 22
```

Next, we need a function to delete the maximum value from a tree. (This is simpler than the general deletion problem.)

```
# let rec deleteMax n =
    match n with
      Node(l, v, Leaf) -> l
    | Node(l, v, r) -> Node(l, v, deleteMax r);;
```

(Why does OCaml "grumble" about these definitions?)

Another simple function is a leaf test:

```
# let isLeaf n =  match n with Leaf -> true | _ -> false;;
```

Now we can write the deletion function:

```
# let rec delete x n =
    match n with
      Leaf -> Leaf (* it isn't there !*)
    | Node(l,v,r) ->
        if x < v then
          Node(delete x l, v, r)    (* keep searching to the left *)
        else if x > v then
          Node(l, v, delete x r)    (* keep searching to the right *)
        else if isLeaf (l) then
          r                         (* left subtree is a leaf *)
        else if isLeaf (r) then
          l                         (* right subtree is a leaf *)
        else
          (* use max val in left subtree as new node val *)
          Node(deleteMax l, maxVal l, r);;
```

Of course, we might "clean up" our delete function by making `maxVal` and `deleteMax` local to `delete`.

Maintaining balance

We observed earlier that, for efficiency, we should try to keep our binary search trees balanced. However neither `insert` nor `delete` preserves this property. Consider the function

```
# let rec listToTree l =
    match l with
      [] -> Leaf
    | x::y -> insert x (listToTree y);;
```

What kind of tree will it build when applied to the list `[1; 2; 3; 4; 5; 6; 7; 8]`?

One can design more sophisticated algorithms for keeping a binary search tree balanced during insertions and deletions. You'll learn about these in later courses.

Search trees play a crucial role in many branches of computer science.

An important generalization of the binary search tree is an n-ary search tree. For example, if n is 32 a balanced tree of depth 6 will accommodate 10^9 nodes. What this means in practice is that we can build enormous dictionaries—such as web indexes—that require only a few disk accesses to do a lookup.

Parameterized data types

OCaml allows a data type definition to be "parameterized" by a type variable. One such type is already well-known to us:

```
# type 'a list = Nil | Cons of 'a * 'a list;;
```

With this type definition we can re-write all the basic functions on lists. For example:

```
# let hd (Cons(x,_)) = x;;
# let tl (Cons(_,y)) = y;;
```

All that is missing from these lists, compared to the built-in ones, is the syntactic convenience of having an infix `cons` such as `::` and a special notation for reading/displaying lists `[`e_1`; ...; `e_n`]`.

Recall that a function definition with a pattern "folded-into" the function header can be written equivalently as an ordinary function header followed by a `match`:

```
# let tl l =
    match l with (Cons(_,y)) -> y;;
```

What will happen when we evaluate `tl(Nil)`?

The `option` data type

For convenience, OCaml provides a built-in parametric `option` type:

```
# type 'a option = None | Some of 'a;;
```

This is a simple generalizaton of the `maybe` data type described earlier. Here is another application...

```
# (* find the roots of a*x*x + b*x + c = 0 *)
  let roots (a,b,c) =
    let  v = b *. b -. 4.0 *. a *. c in
      if v < 0.0 then None
      else let s = sqrt v in
           Some((-.b-.s)/.(2.0*.a), (-.b +.s)/.(2.0*.a));;

val roots : float * float * float -> (float * float) option = <fun>
```