Lecture 4
More examples with half

half(0) handle Zero => 1000;
> val it = 1000 : int

half(1) handle Zero => 1000;
> uncaught exception Odd

half(0) handle Zero => 1000 | Odd => 1001;
> val it = 1000 : int

half(3) handle Zero => 1000 | Odd => 1001;
> val it = 1001 : int

- Instead of Zero and Odd could have a single kind of exception containing a string

eception Half of string;
> exception Half

fun half n =
    if n=0 then raise Half "Zero"
    else let
        val m = n div 2
    in
        if n=2*m then m else raise Half "Odd"
    end;
> val half = fn : int -> int
Yet more examples with half

- Contents of exception packets not printed

```haskell
half 0;
> uncaught exception Half

half 3;
> uncaught exception Half

half(0)
    handle Half "Zero" => 1000 | Half "Odd" => 1001;
> val it = 1000 : int

half(3)
    handle Half "Zero" => 1000 | Half "Odd" => 1001;
> val it = 1001 : int
```

- Could match contents of exception packet to a variable, s say, and then branch on the value matched to s

```haskell
half(0)
    handle Half s => (if s="Zero" then 1000 else 1001);
> val it = 1000 : int

half(3)
    handle Half s => (if s="Zero" then 1000 else 1001);
> val it = 1001 : int
```
Datatype declarations

- New types can also be defined

- Datatypes are defined by a set of constructors
  - which can be used to create objects of that type
  - and also – via patterns – to decompose objects

```
datatype card = king | queen | jack | other of int;
datatype card
con jack : card
con king : card
con other : int -> card
con queen : card
```

- Declares constructors king, queen, jack, other

- Gives constructors values
  - value of a 0-ary constructors is constant value
  - value of 1-ary constructor other is a function
    - given an integer value n produces other(n)

```
kng;
val it = king : card

other(4+5);
val it = other 9 : card
```
Patterns and constructors

- Constructors can be used in pattern matching

fun value king = 500
| value queen = 200
| value jack = 100
| value (other n) = 5*n;
> val value = fn : card -> int

- Or:

val value = fn king => 500
| queen => 200
| jack => 100
| (other n) => 5*n;
> val value = fn : card -> int
Primitive datatypes

• The booleans could be defined by:

```haskell
datatype bool = true | false;
> datatype bool
> con false : bool
> con true : bool
```

• The positive integers

```haskell
datatype int = zero | suc of int;
> datatype int
> con suc : int -> int
> con zero : int
```
Lisp S-expressions

datatype sexp \triangledown litatom of string
  | numatom of int
  | cons of sexp * sexp;

> datatype sexp
> con cons : sexp * sexp -> sexp
> con litatom : string -> sexp
> con numatom : int -> sexp

fun car (cons(x,y)) = x and cdr (cons(x,y)) = y;
> Warning: match nonexhaustive
> val car = fn : sexp -> sexp
> Warning: match nonexhaustive
> val cdr = fn : sexp -> sexp

val a1 = litatom "Foo" and a2 = numatom 1;
> val a1 = litatom "Foo" : sexp
> val a2 = numatom 1 : sexp

car(cons(a1,a2));
> val it = litatom "Foo" : sexp

cdr(cons(a1,a2));
> val it = numatom 1 : sexp

• These funtions are only partially specified

car (litatom "foo");
> uncaught exception Match
Abstract types

- An abstract type declaration has the form
  \begin{align*}
  \text{abstype } d \text{ with } b \text{ end}
  \end{align*}

  - $d$ is a datatype specification
  - $b$ is a binding
    - i.e. the kind of phrase that can follow `val`

- Such a declaration introduces:
  - a new type, $ty$ say
  - specified by the datatype declaration $d$

- Constructors declared on $ty$ by $d$ only available within $b$

- Exported bindings are those specified in $b$

- Values of an abstract type are printed as “–”
exception BadTime;
> exception BadTime

abstype time = time of int * int
with
  fun maketime(hrs, mins) =
    if hrs<0 orelse 23<hrs orelse mins<0 orelse 59<mins
      then raise BadTime
      else time(hrs, mins)
    and hours(time(t1,t2)) = t1
    and minutes(time(t1,t2)) = t2
end;
> type time
> val maketime = fn : int * int -> time
> val hours = fn : time -> int
> val minutes = fn : time -> int

val t = maketime(8,30);
> val t = - : time

(hours t , minutes t);
> val it = (8,30) : int * int

• Defines an abstract type time
  • with three primitive functions:
    maketime, hours, minutes
• An abstract type declaration simultaneously declares
  • a new type
  • together with primitive functions for the type

• The representation datatype is not accessible outside the with-part of the declaration
Type constructors

• list and * are type constructors
  
  • list has one argument – hence ’a list
  
  • * has two – hence ’a * ’b

• Useful operations can be defined using patterns

```ml
fun fst(x,y) = x and snd(x,y) = y;
> val fst = fn : ’a * ’b -> ’a
> val snd = fn : ’a * ’b -> ’b

val p = (8,30);
> val p = (8,30) : int * int

fst p;
> val it = 8 : int

snd p;
> val it = 30 : int
```

• See also previous definitions of hd, tl, null
Example: sets

- set represents sets as lists without repetitions

```ocaml
abstype 'a set = set of 'a list
with
  val emptyset = set[]
  fun isempty(set s) = null s
  fun member(_:_, set[]) = false
     | member(x, set(y::z)) =
         (x=y) orelse member(x, set z)
  fun add(x, set[]) = set[x]
     | add(x, set(y::z)) =
         if x=y then set(y::z)
         else let val set l = add(x, set z)
              in set(y::l) end
end
> val emptyset = [] : 'a list
> val isempty = fn : 'a set -> bool
> val member = fn : ''a * ''a set -> bool
> val add = fn : ''a * ''a set -> ''a set

val s = add(1,(add(2,(add(3,emptyset)))))
> val s = - : int set

member(3,s);
> val it = true : bool

member(5,s);
> val it = false : bool
```
References and assignment

- References are ‘boxes’ that can contain values
- Contents can be changed using :=
- “ty ref” is type of references containing values of type ty
- References are created using the ref operator
  - takes a value of type ty to a value of type ty ref.
- $x := e$ changes
  - contents of reference $x$
  - to the value of $e$
- Value of assignment expression is ()
  - assignments are executed for a ‘side effect’, not for their value
- Contents of a reference can be extracted using the ! operator
Example showing references

```ml
val x = 0;

x:=1;
> Type clash in: (x := 1)
> Looking for a: 'a ref
> I have found a: int

val x = ref 1 and y = ref 2;
> val x = ref 1 : int ref
> val y = ref 2 : int ref

x;
> val it = ref 1 : int ref

x:=6;
> val it = () : unit

x;
> val it = ref 6 : int ref

!x;
> val it = 6 : int
```

- Only use references if you have to!
  - experience shows their use increases errors
Iteration

- Semicolon denotes sequencing
  - value of $e_1; \ldots; e_n$ is value of $e_n$

- Evaluating while $e$ do $c$ consists in
  - evaluating $e$
  - if the result is true
    - $c$ is evaluated for its side-effect
    - and then the whole process repeats
  - if $e$ evaluates to false
    - the evaluation of while $e$ do $c$ terminates with value ()
Example: iterative factorial

- An iterative definition of fact
  - uses two local references: count and result

```ml
fun fact n =  
  let val count = ref n and result = ref 1  
  in while !count > 0  
    do (result := !count * !result;  
         count := !count-1);  
    !result  
  end;  
> val fact = fn : int -> int

fact 6;  
> val it = 720 : int
```