Lecture 3

Recursion

- **The factorial function:**

```
fun fact(n:int) = if n=0 then 1 else n*fact(n-1);
> val fact = fn : int -> int

fact 5;
> val it = 120 : int
```

- **Compiler automatically detects recursive calls**

```
fun f n : int = n+1;
> val f = fn : int -> int

fun f(n:int) = if n=0 then 1 else n*f(n-1);
> val f = fn : int -> int

f 3;
> val it = 6 : int
```

- `f 3` **results in the evaluation of** `3*f(2)`

- **In old MLs, first** `f` **would have been used**

  - `f(2)` **would evaluate to** 2+1=3

  - **hence the expression** `f 3` **would evaluate to** 3*3=9

Alternative specification of recursion

- **Can use `val` and `fn` to define functions**

```
fun f n : int = n+1;
> val f = fn : int -> int

val f = fn (n:int) => if n=0 then 1 else n*f(n-1);
> val f = fn : int -> int

f 3;
> val it = 9 : int
```

- **`f` in `n*f(n-1)` is the previous version of `f`**

- **`rec` after `val` forces recursion**

```
fun f n : int = n + 1;
> val f = fn : int -> int

val rec f = fn (n:int) => if n=0 then 1 else n*f(n-1);
> val f = fn : int -> int

f 3;
> val it = 6 : int
```

Testing equality

- **Concrete values like integers, booleans and strings can be tested for equality**

- **Values of simple datatypes, like pairs and records, whose components have concrete types can also be tested for equality**

  - `(`v_1`,`v_2`) = (`v_1'`,`v_2'`)` **iff** $v_1 = v_1'$ **and** $v_2 = v_2'$

- **Many values can be tested for equality**

  - **there are infinitely many of them**

Equality types

- **In general, cannot test equality of functions**

- **Thus not possible to overload = on all types**

- **In old ML = was interpreted on all types**

  - **functions were equal if their addresses were equal**

  - **if test yielded `true` then functions equal**

  - **but many mathematically (i.e. extensionally) equal functions come out different**

- **Types whose values can be tested for equality are** *equality types*

  - **Equality type variables range over equality types**

  - **equality type variables have the form** `''`α

  - **ordinary type variables have the form** `'`α

- **= has type** `''a * ''a -> bool`

Examples of equality types

- **The ML typechecker infers types containing equality type variables**

```
fun Eq x y = (x = y);
> val Eq = fn : ''a -> ''a -> bool

fun EqualHd l1 l2 = (hd l1 = hd l2);
> val EqualHd = fn : ''a list -> ''a list -> bool
```

- **Trying to instantiate an equality type variable to a functional type results in an error**

```
hd = hd;
> Error: operator and operand don't agree
>   operator domain: ''Z * ''Z
>   operand:         ('Y list -> 'Y) * ('X list -> 'X)

EqualHd [hd] [hd];
> Error: operator and operand don't agree
>   operator domain: 'Z * 'Z
>   operand:         ''Y list -> ''Y list -> bool
```

- **Equality types are controversial:**
  - benefits not worth the messiness
  - future versions of ML may omit them

Pattern matching

- **Functions can be defined by pattern matching**

```
fun fact 0 = 1
 |  fact n = n * (fact(n-1));
> val fact = fn : int -> int

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

- **Suppose function f is defined by**

$$
\begin{array}{llll}
\texttt{fun} & f\ p_1 & = & e_1 \\
\texttt{|} & f\ p_2 & = & e_2 \\
 & \vdots & & \\
\texttt{|} & f\ p_n & = & e_n
\end{array}
$$

- **$f\ e$ is evaluated by:**
  - **matching e's value with p_1, p_2, $\ldots$, p_n (that order)**
  - **until a match is found, say with p_i**
  - **value of $f\ e$ is then value of e_i**

- **variables in patterns are locally bound to bits of e they match**

Patterns

- **Patterns need not be exhaustive**

```
- fun foo 0 = 0;
***Warning:  Patterns in Match not exhaustive:  0=>0
> val foo = Fn : int -> int
```

- **What if a function is applied to an argument whose value doesn't match any pattern?**
  - **a run-time error called an exception results**
  - **exception are covered later**

```
- foo 1;
Exception raised at top level
Exception: Match  raised
```

- **The wildcard "_" matches anything**

```
fun null [] = true
 |  null _  = false;
> val null = fn : 'a list -> bool
```

Examples

- **functions `hd` and `tl` can be defined by:**

```
fun hd(x::l) = x;
> Warning: match nonexhaustive
> val hd = fn : 'a list -> 'a

fun tl(x::l) = l;
> Warning: match nonexhaustive
> val tl = fn : 'a list -> 'a list
```

- **Almost the same results as the built-in functions**
  - **on `[]` they give different exceptions**

```
hd [];                                  (* built-in "hd" *)
> uncaught exception Hd

fun hd(x::l) = x;
> Warning: match nonexhaustive
> val hd = fn : 'a list -> 'a

hd[];                                   (* redefined "hd" *)
> uncaught Match exception std_in:0.0-0.0
```

The as construct in patterns

- x `as` p **is a pattern that**
  - **matches the same things as** p
  - **binds value matched to** x

```
fun RemoveDuplicates[]          = []
 |  RemoveDuplicates[x]         = [x]
 |  RemoveDuplicates(x1::x2::l) =
     if x1=x2 then RemoveDuplicates(x2::l)
              else x1::RemoveDuplicates(x2::l);

>val RemoveDuplicates = fn : ''a list -> ''a list

RemoveDuplicates[1,1,1,2,3,4,5,5,5,5,5,6,7,8,8,8];
> val it = [1,2,3,4,5,6,7,8] : int list
```

- **Using** `as`**:**

```
fun RemoveDuplicates[]                 = []
 |  RemoveDuplicates(l as [x])         = l
 |  RemoveDuplicates(x1::(l as x2::_)) =
     if x1=x2 then RemoveDuplicates l
              else x1::RemoveDuplicates l;
```

Repeated variables not allowed

- **Alas:**

```
fun RemoveDuplicates[] = []
 |  RemoveDuplicates(l as [x]) = l
 |  RemoveDuplicates(x::(l as x::_)) =
     RemoveDuplicates l
 |  RemoveDuplicates(x::l) =
     x::RemoveDuplicates l;
> Error: duplicate variable in pattern(s): x
```

Anonymous functions can use patterns

- `fn`-expressions can use patterns

  - `fn` p_1 `=>` e_1 `|` $\ldots$ `|` p_n `=>` e_n

```
fn []     => "none"
|  [_]    => "one"
|  [_,_] => "two"
|  _      => "many";
> val it = fn : 'a list -> string

(it [], it[true], it[1,2], it[1,2,3]);
> val it = ("none","one","two","many")
```

Patterns and records

- **Patterns can be constructed out of records**
  - **"..." (three dots) acts as a wildcard**

```
fun IsMale({sex="male",...}:persondata) = true
 |  IsMale _ = false;
> val IsMale = fn : persondata -> bool

IsMale MikeData;
> val it = true : bool
```

- **An alternative definition**

```
fun IsMale({sex=x,...}:persondata) = (x = "male");
```

- **A more compact form of this is allowed**

```
fun IsMale({sex,...}:persondata) = (sex = "male");
```

- **The field name `sex` doubles as a variable**
  - `{···,v,···}` **abbreviates** `{···,v=v,···}`

The `case` construct

- **The following are equivalent:**
  - `case` e `of` p_1 `=>` e_1 `|` $\ldots$ `|` p_n `=>` e_n
  - `(fn` p_1 `=>` e_1 `|` $\ldots$ `|` p_n `=>` e_n`)` e

```
fun RemoveDuplicates[]                  = []
 |  RemoveDuplicates(l as [x])          = l
 |  RemoveDuplicates(x1::(l as x2::_)) =
      case x1=x2
      of  true => RemoveDuplicates l
      |    _    => x1::RemoveDuplicates l;
> val RemoveDuplicates = fn : ''a list -> ''a list
```

Exceptions

- **Exceptions are a kind of event that occur during evaluation**

- **Can result from run-time errors**
  - **e.g.** `1/0`

- **Can be generated explicitly**
  - **e.g.** `raise` Ex

Exception values

- **Exception values are ML values of type `exn`**
- **`exn` is a datatype**
  - **datatypes are explained later**
- **Exception value constructors:**
  - **delared using keyword `exception`**
  - **can be used in patterns**

Exception packets

- **An exception event raises an** *exception packet*

- **An exception packet is a raised exception value**

  - **Exception packets break normal flow-of-control**

  - **they can be trapped using a handler**

    e `handle` p_1 `=>` e_1 `|` $\ldots$ `|` p_n `=>` e_n

- **Expression evaluation either:**

  - **succeeds with a value**

  - **raises an exception**
    i.e. fails with an exception packet

  - **doesn't terminate**

Raised exceptions

- **Functions can *raise exceptions* at run-time**
- **A special kind of value is propagated**
  - **called an exception packet**
  - **usually identifies the cause of the exception**
- **Exception packets have names**
  - **usually reflect the function that raised the exception**
- **Exception packets may also contain values**

```
hd(tl[2]);
> uncaught exception Hd

1 div 0;
> uncaught exception Div

(1 div 0)+1000;
> uncaught exception Div
```

Declaring exceptions

- **Exceptions are declared using the keyword** `exception`
    - **they have type** `exn`
- **Exceptions are raised by evaluating** `raise` e
    - **where** e **evaluates to an exception value**

```
exception Ex1;exception Ex2;
> type exn
  con Ex1 = - : exn
> type exn
  con Ex2 = - : exn

- [Ex1,Ex2];
> [-,-] : exn list

- raise hd it;
Exception raised at top level
Warning: optimisations enabled -
         some functions may be missing from the trace
Exception: Ex1  raised
```

Exception packets

- "`exception` *name* `of` *ty*" **declares**
  - **an exception packet constructor called** *name*
  - **that constructs packets containing values of type** *ty*

```
exception Ex3 of string;
> exception Ex3

Ex3;
> val it = fn : string -> exn

raise Ex3 "foo";
> uncaught exception Ex3
```

- **The type `exn` is a datatype**
  - **see later**
- **Exceptions can be used in patterns**
  - **useful for handling exceptions**

Handling exceptions

- **Exceptions are trapped using exception handlers**

- **Example: trapping all exceptions:**
  - **Value of "e_1 `handle _ =>` e_2" is**
  - **value of e_1, unless e_1 raises an exception**
  - **in which case it is the value of e_2**

```
hd[1,2,3] handle _ => 0;
> val it = 1 : int

hd[] handle _ => 0;
> val it = 0 : int

hd(tl[2]) handle _ => 0;
> val it = 0 : int

1 div 0 handle _ => 1000;
> val it = 1000 : int
```

Example: `half`

- **The function `half` only succeeds on non-zero even numbers**
  - **on `0` it raises `Zero`**
  - **on odd numbers it raises `Odd`**

```
exception Zero; exception Odd;
> exception Zero
> exception Odd

fun half n =
 if n=0 then raise Zero
        else let
              val m = n div 2
             in
              if n=2*m then m else raise Odd
             end;
> val half = fn : int -> int
```

Some examples of using `half`

```
half 4;
> val it = 2 : int

half 0;
> uncaught exception Zero

half 3;
> uncaught exception Odd

half 3 handle _ => 1000;
> val it = 1000 : int
```

- **Exceptions can be trapped selectively**
  - **by matching the exception packet**

- **If e raises Ex**
  - **value of "e `handle` Ex_1 `=>` e_1 `|` ... `|` Ex_n `=>` e_n" is**
    - **the value of e_i if Ex equals Ex_i**
    - **otherwise the `handle`-expression raises Ex**