Overview of chapters 7 and 8

Main topics:
- `datatype declarations` and their use
  - non recursive (chap. 7)
  - recursive (chap. 8)
- `abstype declarations` and their use (chap. 8)

Additional topics (chap. 7):
- `case expressions`
- `patterns`
- `exceptions`
- `partial functions`

This slide set covers topics from chapter 7.
Disjoint Union: An Example

A shape is either a circle, a square, or a triangle

- the union of three disjoint sets

A datatype declaration for shapes:
datatype shape = Circle of real |
| Square of real |
| Triangle of real*real*real;

Answer from the SML system:
> datatype shape
> con Circle = fn : real -> shape
> con Square = fn : real -> shape
> con Triangle = fn : real * real * real -> shape

Constructors of a datatype

The tags Circle, Square and Triangle are constructors of values of type shape

- Circle 2.0;
  > val it = Circle 2.0 : shape

- Triangle(1.0, 2.0, 3.0);
  > val it = Triangle(1.0, 2.0, 3.0) : shape

- Square 4.0;
  > val it = Square 4.0 : shape

Equality on shapes is defined provided ...

- Triangle(1.0, 2.0, 3.0) = Square 2.0;
  > val it = false : bool

Constructors in Patterns

fun area(Circle r) = Math.pi * r * r |
| area(Square a) = a * a |
| area(Triangle(a,b,c)) =
  let val d = (a + b + c)/2.0
  in Math.sqrt(d*(d-a)*(d-b)*(d-c))
  end;

> val area = fn : shape -> real

  a constructor only matches itself

  area (Circle 1.2)
  ~> (Math.pi * r * r, [r->1.2])
  ~> ...

The case-expression

Form:

case exp of
  pat1 => e1
  | pat2 => e2
  | ...
  | patk => ek

Example:

fun area s =
  case s of
    (Circle r) => Math.pi * r * r
    | (Square a) => a*a
    | (Triangle(a,b,c)) =>
      let val d = (a + b + c)/2.0
      in Math.sqrt(d*(d-a)*(d-b)*(d-c))
      end;
Enumeration types – the predefined `order` type

datatype order = LESS | EQUAL | GREATER;
Predefined ‘compare’ functions, e.g.

\[
\text{Int.compare}(x,y) = \begin{cases} 
\text{LESS} & \text{if } x < y \\
\text{EQUAL} & \text{if } x = y \\
\text{GREATER} & \text{if } x > y 
\end{cases}
\]

Example:
fun countLEG [ ] = (0,0,0)
| countLEG(x::rest) = let val (y1,y2,y3) = countLEG rest in
  case Int.compare(x,0) of
    LESS => (y1+1,y2 ,y3 )
  | EQUAL => (y1 ,y2+1,y3 )
  | GREATER => (y1 ,y2 ,y3+1)
end;

Polymorphic types, the predefined `option` type

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

The type `'a option` is parameterized with the element type. The value constructors are polymorphic:

- con `'a NONE = NONE : `'a option
- con `'a SOME = fn : `'a -> `'a option

SOME 3 has type `int option` which is an `instance` of `'a option`.

Polymorphic types contain type variables: `'a list, `'a option, ...

Monomorphic types do not contain type variables:
int, int list, bool list, int option, ...

Partial functions

A function \( f : \tau_1 \rightarrow \tau_2 \) is partial, if application \( f(v) \) is undefined for some value \( v : \tau_1 \).

Three possibilities for treating \( f(v) \) when declaring \( f \) in SML:

- let \( f(v) \) be undefined (e.g. no match, no termination, ...)
  
  \[
  \text{fun fact} \ 0 = 1 | \text{fact} \ n = n * \text{fact}(n-1);
  \]

- let \( f(v) \) raise an user-defined exception
  
  \[
  \text{fun fact1} \ n = \text{if } n < 0 \text{ then raise BadArgument n else fact} n;
  \]

- let \( f(v) \) return a special value (NONE) and
  
  let \( f(x) \) return SOME \( y \) when \( f(x) \) is defined to give \( y \)
  
  \[
  \text{fun fact2} \ n = \text{if } n < 0 \text{ then NONE else SOME (fact} n);
  \]

Partial functions, example

- fact1 3;
> val it = 6 : int

- fact2 3;
> val it = SOME 6 : int option

- fact1 ~1;
  ! Uncaught exception:
  ! BadArgument

- fact2 ~1;
> val it = NONE : int option
Exceptions

Exceptions are used to terminate the evaluation of an expression with an "error signal".

Issues:

- declaration of exception constructors $E_i$:
  
  exception $E_i$ or exception $E_i$ of $ty$

- exception raising:
  
  raise $E_i$ or raise $E_i$ exp

- exception handling (catching):
  
  $exp$ handle $pat_1 <= exp'_1 | ... | pat_n <= exp'_n$
  
  where $pat_i$ is an exception raising pattern of one of the forms: $E_i$ or $E_i$ exp, and $exp'_i$ has same type as $exp$

Exceptions: examples

Declaring an exception: exception BadArgument of int;

Specifying an exception to be raised:

fun fact1 n = if n < 0 then raise BadArgument n else fact n;

No handling of exception:

- fact1 ~2;
  > ! Uncaught exception:
  > ! BadArgument

Handling an exception:

- fun usefact1 n =
-   ... fact1 n ...
-   handle BadArgument n => "Bad argument ":Int.toString n;

- usefact1 ~2;
  > val it = "Bad argument ~2" : string