Chapter 5
Type Declarations

(Version of 27 September 2004)

1. *Renaming existing types* 5.2

2. *Enumeration types* 5.3

3. *Constructed types* 5.5

4. *Parameterised/polymorphic types* 5.10

5. *Exceptions, revisited* 5.12

6. *Application: expression evaluation* 5.13

5.1. Renaming existing types

Example: polynomials, revisited

Representation of a polynomial by a list of integers:

```
type poly = int list
```

- Introduction of an *abbreviation* for the type **int list**
- The two names denote the *same* type
- The object [4,2,8] is of type poly *and* of type **int list**

```
- type poly = int list ;
  type poly = int list
- type poly2 = int list ;
  type poly2 = int list
- val p:poly = [1,2] ;
  val p = [1,2] : poly
- val p2:poly2 = [1,2] ;
  val p2 = [1,2] : poly2
- p = p2 ;
  val it = true : bool
```

5.2. Enumeration types

Declaration of a *new* type having a *finite* number of values

Example (`weekend.sml`)

```
datatype months  =  Jan | Feb | Mar | Apr | May | Jun
                 |  Jul | Aug | Sep | Oct | Nov | Dec

datatype days = Mon | Tue | Wed | Thu | Fri | Sat | Sun

fun weekend Sat = true
  | weekend Sun = true
  | weekend d = false
```

```
- datatype months = Jan | ... | Dec ;
  datatype months = Jan | ... | Dec
- datatype days = Mon | ... | Sun ;
  datatype days = Mon | ... | Sun
- fun weekend ... ;
  val weekend = fn : days -> bool
```

- Convention (of this course):
  constant identifiers (tags) start with an uppercase letter,
  to avoid confusion with the function identifiers
- Possibility of using pattern matching
- Two datatype declarations cannot share the same constant,
  as otherwise there would be typing problems
- Impossibility of defining sub-types in ML,
  such as the integer interval 1..12 or the months {Jul, Aug}
- Equality is *automatically* defined on enumeration types

The bool and unit types, revisited

The types **bool** and **unit** are *not* primitive in ML:
they can be declared as enumeration types as follows:

```
datatype bool = true | false
datatype unit = ()
```

The order type

```
datatype order = LESS | EQUAL | GREATER
```

It is the result type of the comparison functions Int.compare, Real.compare, Char.compare, String.compare, etc.

5.3. Constructed types

Example (person.sml)

```
datatype name = Name of string
datatype sex = Male | Female
datatype age = Age of int        (* expressed in years *)
datatype weight = WeightInKg of int
datatype person = Person of name * sex * age * weight
val ayşe = Person (Name "Ayşe", Female, Age 30, WeightInKg 58)
```

```
- datatype name = Name of string ;
  datatype name = Name of string
- Name ;
  val it = fn : string -> name
- val friend = Name "Ali" ;
  val friend = Name "Ali" : name
- friend = "Ali" ;
  ! friend = "Ali"
  !          ^^^^^
  ! Error: operator and operand don't agree
```

The identifiers Name, Age, ..., Person are *value constructors*:

- Their declarations introduce collections of *tagged values*
- The type of a value constructor is a functional type; for instance, Age is of type **int** → age
- The constructor Age may only be applied to an expression of type **int**, giving as result an object of type age
- If expression e reduces to the normal form n of type **int**, then Age e reduces to Age n, which is of type age
- The usage of the constructor Age is the *only* means of constructing an object of type age
- The tags of enumeration types are nothing else but value constructors without arguments!
- Value constructors can be used in patterns

Example

```
fun youngLady (Person ( _ , Female, Age a, _ )) = a <= 18
  | youngLady p = false

-  youngLady ayşe ;
   val it = false
```

Types with several constructors

Example 1: coordinates of a point in the plane

Handling Cartesian *and* polar coordinates (`coord.sml`)

```
datatype coord = Cart of real * real
               | Polar of real * real
```

Transformation of coordinates

```
fun toPolar (Polar (r,phi)) = ...
  | toPolar (Cart (x,y)) = ...

fun toCart (Cart (x,y)) = ...
  | toCart (Polar (r,phi)) = ...
```

Addition and multiplication of coordinates

```
fun addCoord c1 c2 =
    let val (Cart (x1,y1)) = toCart c1
        val (Cart (x2,y2)) = toCart c2
    in Cart (x1+x2, y1+y2) end

fun multCoord c1 c2 =
    let val (Polar (r1,phi1)) = toPolar c1
        val (Polar (r2,phi2)) = toPolar c2
    in Polar (r1*r2, phi1+phi2) end
```

Example 2: computation with integers and reals

Handling numbers, whether integers or reals (`number.sml`)

```
datatype number = Int of int
                | Real of real

fun toReal (Int i) = Real ( real i )
  | toReal (Real r) = Real r

fun addNumbers (Int a) (Int b) = Int (a+b)
  | addNumbers nb1 nb2 =
      let val (Real r1) = toReal nb1
          val (Real r2) = toReal nb2
      in Real (r1+r2) end
```

- The union of types can now be simulated
- The value constructors reveal the types of the objects
- Equality is defined on constructed types
  if the types of the objects are not functional

Recursive types

Possibility of recursively using the declared type

Example 1: integer (linear) lists

```
datatype listInt = [ ]
                 | :: of int * listInt
```

Example 2: integer binary trees

```
datatype bTreeInt = Void
                  | Bt of int * bTreeInt * bTreeInt
```

Necessity of at least one non-recursive alternative!
The following objects are of type bTreeInt:

```
Void

Bt(12,Void,Void)

Bt(~5,Void,Void)

Bt(8+5, Bt(4,Void,Void), Void)

Bt(17, Bt(7,Void,Void), Bt(~1,Void,Void))

Bt(5, Bt(3, Bt(12, Bt(5,Void,Void), Bt(13,Void,Void)),
            Bt(1, Bt(10,Void,Void), Void)),
      Bt(8, Bt(5,Void,Void), Bt(4,Void,Void)))
```

Mutually recursive datatypes are declared together using **and** and **withtype**

5.4. Parameterised/polymorphic types

Example 1: (linear) lists

(Linear) lists are an example of polymorphic type:

```
datatype 'a list = [ ]
                 | :: of 'a * 'a list
```

where **list** is called a *type constructor*

Example 2: binary trees

The type bTreeInt defines binary trees with an *integer* as information associated to each node:
how to define binary trees of objects of type α?

```
datatype 'a bTree = Void
                  | Bt of 'a * 'a bTree * 'a bTree
```

where **bTree** is a *type constructor*

- Bt(2,Void,Void) is of type **int** bTree
- Bt("Ali",Void,Void) is of type **string** bTree
- Void is of type α bTree
- Bt(2, Bt("Ali",Void,Void), Void) is *not* a binary tree

Example 3: the predefined option type

The option type gives a new approach to partial functions

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

The predefined valOf function "removes the SOME" and "changes" the type of its argument:

function valOf expr
TYPE: 'a option → 'a
PRE: expr is of the form SOME e, otherwise exception Option is raised
POST: e

```
- valOf (SOME (19+3)) ;
  val it = 22 : int
- valOf NONE ;
  ! uncaught exception Option
```

Example: factorial (ch03/fact.sml)

```
fun factOpt n =
   if n < 0 then NONE
   else if n = 0 then SOME 1
   else SOME (n * valOf (factOpt (n-1)))
```

```
- factOpt 4 ;
  val it = SOME 24 : int option
- factOpt ~3 ;
  val it = NONE : int option
```

5.5. Exceptions, revisited

```
-   exception StopError ;
  exception StopError

-   exception NegArg of int ;
  exception NegArg of int
```

The identifiers StopError and NegArg
are declared as *exception constructors*

Patterns may involve value and exception constructors

An exception is *propagated* until being *caught*
by an *exception handler*, at the latest by the top-level one

Example: factorial (ch03/fact.sml)

```
local
   fun factAux 0 = 1
     | factAux n = n * factAux (n-1)
in fun factExc n =
     (if n < 0 then raise NegArg n else Int.toString (factAux n) )
     handle NegArg n => "fact " ^ Int.toString n ^ " is undefined"
end

-   factExc 4 ;
  val it = "24" : string

-   factExc ~3 ;
  val it = "fact ~3 is undefined" : string
```

5.6. Application: expression evaluation

Evaluation of an integer arithmetic expression containing occurrences of some variable

Representation of expressions as *syntax trees*

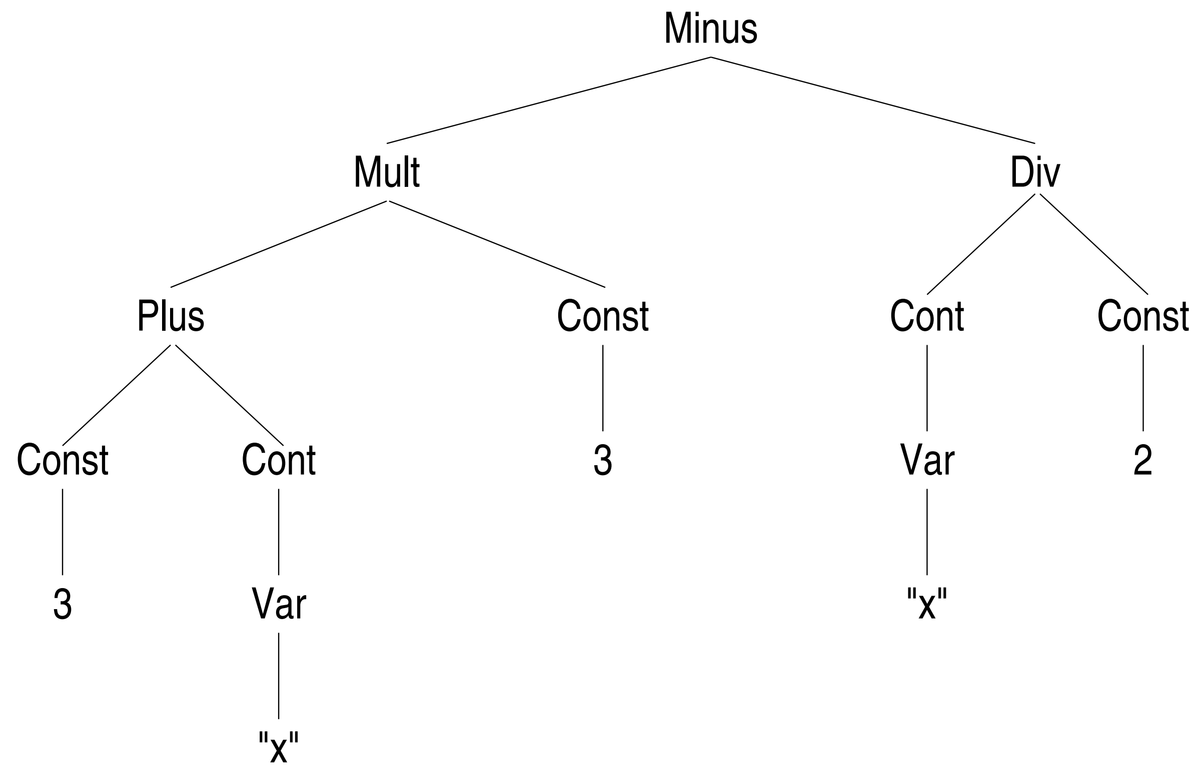
Datatype declarations

```
datatype variable = Var of string

datatype expression = Const of int
                    | Cont of variable
                    | Plus of expression * expression
                    | Minus of expression * expression
                    | Mult of expression * expression
                    | Div of expression * expression
```

Example

The expression $(3 + x) * 3 - x/2$
is represented by the following syntax tree:



and is written in ML with our new datatypes as follows:

```
Minus ( Mult ( Plus ( Const 3, Cont (Var "x") ),
               Const 3 ),
        Div ( Cont (Var "x"), Const 2 ) )
```

Evaluation (eval.sml)

function eval exp var val
TYPE: expression → variable → int → int
PRE: var is the only variable that may appear in exp
POST: the value of exp where var is replaced by val

```
fun eval (Const n) var v = n
  | eval (Cont a) var v =
      if a <> var then error "eval: pre-condition violated"
      else v
  | eval (Plus(e1,e2)) var v =
      let val n1 = eval e1 var v
          val n2 = eval e2 var v
      in n1 + n2 end
  | eval (Minus(e1,e2)) var v =
      let val n1 = eval e1 var v
          val n2 = eval e2 var v
      in n1 - n2 end
  | eval (Mult(e1,e2)) var v =
      let val n1 = eval e1 var v
          val n2 = eval e2 var v
      in n1 * n2 end
  | eval (Div(e1,e2)) var v =
      let val n1 = eval e1 var v
          val n2 = eval e2 var v
      in if n2 = 0 then error "eval: division by 0"
         else n1 div n2
      end
```

For the expression $(3 + x) * 3 - x/2$
the call `eval (Minus (...)) (Var "x") 5` returns the value `22`

Exercises

- How to integrate variables and constants of type **real**?
- Extend the `eval` function such that the expression may contain *several* variables: the second argument could then be a *list* of (variable, value) pairs