Record Types

{- name="Jones", salary=20300, age=26};

val it =
{age = 26, name = "Jones", salary = 20300} : {age : int, name : string, salary : int}

- {1="Jones", 2=20300,3=26};

> val it = ("Jones", 20300, 26) : string * int * int
- `val emp1 =` 
  `{name="Jones", salary=20300, age=26};`

> `val emp1 =` 
  `{age = 26, name = "Jones", salary = 20300}` 
  `: {age : int, name : string, salary : int}`

- `val {name=n1,salary=s1,age=a1}= emp1;`

> `val n1 = "Jones" : string`
  `val s1 = 20300 : int`
  `val a1 = 26 : int`

- `val {name=n1,salary=s1,...} = emp1;`

> `val n1 = "Jones" : string`
  `val s1 = 20300 : int`

- `val {name,age,...} = emp1;`

> `val name = "Jones" : string`
  `val age = 26 : int`
type employee = {name: string, 
salary: int, 
age: int};

> type employee

fun tax (e: employee) =
real(#salary e)*0.22

Or,

fun tax ({salary,...}: employee) =
real(salary)*0.22;
Enumerated Types

Consider the King and his court:

datatype degree = Duke
| Marquis
| Earl
| Viscount
| Baron;

datatype person =
   King
| Peer of degree*string*int
| Knight of string
| Peasant of string;

All constructors are distinct.
Functions on Datatypes

[King,
  Peer(Duke, "Gloucester", 5),
  Knight "Gawain",
  Peasant "Jack Cade"];

val it = ... : person list

fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _,Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false;
Exceptions

Exceptions are raised when there is no matching pattern, when an overflow occurs, when a subscript is out of range, or some other run-time error occurs.

Exceptions can also be explicitly raised.

```
exception Failure;
exception BadVal of Int;

raise Failure
raise (BadVal 5)
```

```
E handle P_1 => E_1 | ... | P_n => E_n
```
Recursive Datatypes

The built-in type operator of lists might be defined as follows:

\[
\text{infix} :: ;
\]

\[
\text{datatype} \ 'a\ \text{list} = \text{nil} \\
| \quad :: \text{of} \ 'a * 'a\ \text{list};
\]

Binary Trees:

\[
\text{datatype} \ 'a\ \text{tree} = \\
\quad \text{Lf} \\
| \quad \text{Br of} \ 'a * 'a\ \text{tree} * 'a\ \text{tree};
\]

\[
\text{Br}(1, \text{Br}(2, \text{Br}(4, \text{Lf, Lf}), \text{Br}(5, \text{Lf, Lf})), \text{Br}(3, \text{Lf, Lf}))
\]
Functions on Trees

Counting the number of branch nodes

fun count Lf = 0
  | count (Br(v,t1,t2)) =
    1+count(t1)+count(t2);

val count = fn : 'a tree -> int

Depth of a tree

fun depth Lf = 0
  | depth (Br(v,t1,t2)) =
    1+Int.max(depth t1, depth t2);

val depth = fn : 'a tree -> int
Listing a Tree

Three different ways to list the data elements of a tree

Pre-Order

\[
\text{fun preorder } Lf = []
\]
\[
| \text{preorder } (\text{Br}(v,t1,t2)) =
\]
\[
[v] @ \text{preorder } t1 @ \text{preorder } t2;
\]

In-Order

\[
\text{fun inorder } Lf = []
\]
\[
| \text{inorder } (\text{Br}(v,t1,t2)) =
\]
\[
\text{inorder } t1 @ [v] @ \text{inorder } t2;
\]

Post-Order

\[
\text{fun postorder } Lf = []
\]
\[
| \text{postorder } (\text{Br}(v,t1,t2)) =
\]
\[
\text{postorder } t1 @ \text{postorder } t2 @ [v];
\]
Multi-Branching Trees

To define a datatype of a tree where each node can have any number of children

datatype 'a mtree =
    Branch of 'a * ('a mtree) list;

To recursively define functions, we can use map.

fun double (Branch(k,ts)) =
    Branch(2*k, map double ts);