Lecture 2
Product types

- add could take a single argument of the product type `int * int`

```sml
fun add(x,y):int = x+y;
> val add = fn : int * int -> int

add(3,4);
> val it = 7 : int

let val z = (3,4) in add z end;
> val it = 7 : int

add 3;
> Error: operator and operand don’t agree
> operator domain: int * int
> operand: int
```

- Error message based on SML/NJ

- N.B. the type of the `result` specified
  - sufficient to explicitly type any subexpression
  - must disambiguates all overloaded operators
Type abbreviations

• Types can be given names:

```ml
type intpair = int * int;
> type intpair defined

fun addpair ((x,y):intpair) = x+y;
> val addpair = fn : intpair -> int

(3,5);
> val it = (3,5) : int * int

(3,5):intpair;
> val it = (3,5) : intpair

addpair(3,5);
> val it = 8 : int
```

• The new name is simply an abbreviation
  
  • intpair and int*int are completely equivalent
Multiple results

- Functions may also return structured results

```
fun sumdiff(x:int,y:int) = (x+y,x-y);
> val sumdiff = fn : int * int -> int * int

sumdiff(3,4);
> val it = (7,-1) : int * int
```

- In ML all functions technically have one argument and one result
  - but arguments and results can be structures
Operators

- + (addition) and * are built-in infix operators

- Users can define their own infixes
  - using infix for left associative operators
  - and infixr for right associative ones

```plaintext
infix op1;
infixr op2;
```

- This tells the parser to parse $e_1 \text{ op }_1 e_2$ as $\text{op}_1(e_1,e_2)$ and $e_1 \text{ op }_2 e_2$ as $\text{op}_2(e_1,e_2)$

```plaintext
fun (x:int) op1 (y:int) = x + y;
> val op1 = fn : int * int -> int

1 op1 2;
> val it = 3 : int

fun (x:int) op2 (y:int) = x * y;
> val op2 = fn : int * int -> int

2 op2 3;
> val it = 6 : int
```
Precedence

- **infix** $n$ creates:
  - a left-associative infix
  - of precedence $n$

- **infixr** $n$ creates:
  - a right-associative infix
  - of precedence $n$

- If the $n$ is omitted a default precedence is 0
Suppressing infix status

- The ML parser can be told to ignore the infix status of an occurrence of an identifier by preceding the occurrence with `op`

```ml
op1;
> Error: nonfix identifier required

op op1;
> val it = fn : int * int \rightarrow int
```

- Infix status of an operator can be permanently removed using `nonfix`

```ml
1 + 2;
> val it = 3 : int

nonfix +;
> nonfix +

1 + 2;
> Error: operator is not a function
> operator: int
> in expression:
> 1 + : overloaded
```
Restoring infix status of +

- Removing the infix status of built-in operators is not recommended
- Let’s restore it
  - + is left-associative with precedence 6

```plaintext
infix 6 +;
> infix 6 +
```
Function composition

- A useful built-in operator is function composition o

```ml
op o;
> val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

fun add1 n = n+1
and add2 n = n+2;
> val add1 = fn : int -> int
> val add2 = fn : int -> int

(add1 o add2) 5;
> val it = 8 : int
```
Lists

- If $e_1, \ldots, e_n$ all have type $ty$

- then $[e_1, \ldots, e_n]$ has type $(ty \text{ list})$

- Standard functions on lists are:
  - hd (head)
  - tl (tail)
  - null (tests for empty list—i.e. equal to [])
  - :: (infixed ‘cons’)
  - @ (infixed ‘append’, i.e. concatenation)
Examples of list-processing operators

val m = [1,2,(2+1),4];
> val m = [1,2,3,4] : int list

(hd m , tl m);
> val it = (1,[2,3,4]) : int * int list

(null m , null []);
> val it = (false,true) : bool * bool

0::m;
> val it = [0,1,2,3,4] : int list

[1, 2] @ [3, 4, 5, 6];
> val it = [1,2,3,4,5,6] : int list

[1,true,2];
> Error: operator and operand don’t agree
>    operator domain: bool * bool list
>    operand: bool * int list

- Members of a list must have the same type
Strings

- Sequence of characters enclosed between quotes ("\") is a string

```
"this is a string";
> val it = "this is a string" : string
```

- The empty string is ""

- `explode` converts a string to a list of single-character strings

- `implode` is the inverse of `explode`

```
explode;
> val it = fn : string -> string list

explode "this is a string";
> val it =
>   ["t","h","i","s"," ","i","s"," ","a"," ","s","t"," ",
>    "r","i","n","g"]
> : string list

implode it;
> val it = "this is a string" : string
```
Records

- Records are data-structures with named components

- Contrasted with tuples whose components are determined by position

- \( \{x_1=v_1, \ldots, x_n=v_n\} \) creates a record with:
  - fields named \( x_1, \ldots, x_n \)
  - corresponding values \( v_1, \ldots, v_n \)

```haskell
val MikeData = {userid = "mjcg", sex = "male",
               married = true, children = 2};
> val MikeData =
>   {children=2,married=true,sex="male",userid="mjcg"}
> : {children:int, married:bool,
>   sex:string, userid:string}
```

- \( \{x_1:\sigma_1, \ldots, x_n:\sigma_n\} \) is type of \( \{x_1=v_1, \ldots, x_n=v_n\} \)
  - where \( \sigma_i \) is the type of \( v_i \)
Order of fields not significant

- Order of record components does not matter

```haskell
val MikeData' =
  {sex = "male", userid = "mjcg",
   children = 2, married = true};
> val MikeData' =
>  {children=2,married=true,sex="male",userid="mjcg"}
>  : {children:int, married:bool,
>    sex:string, userid:string}

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

- Component named $x$ extracted using $\#x$

```haskell
#children MikeData;
> val it = 2 : int
```
Record arguments need explicit types

- Record types need explicit disambiguation
  - different records can share field names

```haskell
fun Sex p = #sex p;
> Error: unresolved flex record in let pattern

type persondata = {userid:string, children:int,
                   married:bool, sex:string};
> type persondata =
> {children:int, married:bool,
>  sex:string, userid:string}

fun Sex(p:persondata) = #sex p;
> val Sex = fn : persondata -> string

type animal = {kind:string, sex:string};
> type animal = {kind:string, sex:string}

fun IsStallion a =
    #kind a = "horse" andalso #sex a = "male";
> Error: unresolved flex record in let pattern

fun IsStallion(a:animal) =
    #kind a = "horse" andalso #sex a = "male";
> val IsStallion = fn : animal -> bool
```
Tuples are records

- tuples in ML are a special case of records

- \((v_1, \ldots, v_n)\) is equivalent to \(\{1=v_1, \ldots, n=v_n\}\)

```ocaml
{1 = "Hello", 2 = true, 3 = 0};
> val it = ("Hello",true,0) : string * bool * int

#2 it;
> val it = true : bool
```

- \((\sigma_1 \,*\, \sigma_2 \,*\, \cdots \,*\, \sigma_n)\) = \(\{1:\sigma_1, 2:\sigma_2 \cdots, n:\sigma_n\}\)
Polymorphism

- hd, tl etc. can be used on all types of lists

```ocaml
hd [1,2,3];
> val it = 1 : int

hd [true,false,true];
> val it = true : bool

hd [(1,2),(3,4)];
> val it = (1,2) : int * int
```

- hd is used above with types
  - (int list) -> int
  - (bool list) -> bool
  - (int * int) list -> (int * int)

- hd has the type (ty list) -> ty
  - where ty is any type
Type variables

- Functions, like \texttt{hd}, with many types are called \textit{polymorphic}

- ML uses type variables \texttt{'a}, \texttt{'b}, \texttt{'c} etc. to represent their types

\begin{lstlisting}[language=Haskell]
hd;
> val it = fn : 'a list -> 'a
\end{lstlisting}
The ML function `map` takes

- a function with argument type `'a` and result type `'b`
- a list of elements of type `'a`

`map` returns the list obtained by applying the function to each element of the list

- the result has type `'b` list

```ml
map;
> val map = fn : ('a -> 'b) -> 'a list -> 'b list

fun add1 (x:int) = x+1;
> val add1 = fn : int -> int

map add1 [1,2,3,4,5];
> val it = [2,3,4,5,6] : int list
```

`map` can be used at any instance of its type

```ml
map null [[1,2], [], [3], []];
> val it = [false,true,false,true] : bool list
```
fn-expressions

- fn \( x \Rightarrow e \) evaluates to a function
  - with formal parameter \( x \)
  - and with body \( e \)

- fun \( f \ x = e \) is equivalent to val \( f = \text{fn} \ x \Rightarrow e \)

- Similarly
  fun \( f(x,y)z = e \)
  is equivalent to
  val \( f = \text{fn} \ (x,y) \Rightarrow \text{fn} \ z \Rightarrow e \)

- In the \( \lambda \)-calculus \( \lambda \) is used instead of fn
  - fn \( x \Rightarrow e \) in ML is \( \lambda x. e \) in the \( \lambda \)-calculus

```ml
fn x => x+1;
> val it = fn : int -> int

it 3;
> val it = 4 : int
```
Some examples using map

map (fn x => x*x) [1,2,3,4];
> val it = [1,4,9,16] : int list

val doubleup = map (fn x => x*x);
> val doubleup = fn : 'a list list -> 'a list list

doubleup [[1,2], [3,4,5]];
> val it = [[1,2,1,2],[3,4,5,3,4,5]] : int list list

doubleup [];
> val it = [] : 'a list list
Conditionals

- **Syntax** if \( e \) then \( e_1 \) else \( e_2 \)
  - expected meaning
  - true values are true and false
    - both of type bool

```plaintext
if true then 1 else 2;
> val it = 1 : int

if 2<1 then 1 else 2;
> val it = 2 : int
```

- \( e_1 \) orelse \( e_2 \)
  - abbreviates
    - if \( e_1 \) then true else \( e_2 \)

- \( e_1 \) andalso \( e_2 \)
  - abbreviates
    - if \( e_1 \) then \( e_2 \) else false