Review

- Last class you saw lots of types:
  - **Base types:**
    int real bool
    char unit string
  - **Product types:**
    int*bool real*int*string
    int*int etc.
  - **Function types**
    int->int (real*int*int)->bool
    (int*bool)->(bool*int) etc.
  - **List types**
    int list (int*bool) list
    (int list) list etc.
Review

• You saw ways to bind variables to values:

```plaintext
val x    = [3+4, 5+6]
val succ = (fn x => x+1)
fun succ(x) = x+1
```

• You saw pattern-matching and clausal definitions

```plaintext
fun power(x,0) = 1.0
  | power(x,n) = x * power(x,n-1)
fun prod [] = 1
  | prod (n::ns) = n * (prod ns)
```

Length of a List

• You saw a function to compute list length:

```plaintext
fun length []      = 0
  | length ( :::xs) = 1 + length xs
```

• What is the type of `length`?
Types of the Empty List

• Note that
  
  [] : int list
  [] : bool list
  [] : ((string * string) -> string) list
  [] : (string * (string -> string)) list

• In fact, for any type t we have:

  [] : t list

Types of length

fun length [] = 0
  | length (_::xs) = 1 + length xs

• Similarly, for any type t, we have

  length : t list -> int
Polymorphic Types

• SML permits variables representing types, written with a leading prime
  - For example, 'a or 'b or 'c

• Then we can say
  
  ```
  []     : 'a list
  length : 'a list -> int
  ```

• Type variables in such types are implicitly universally-quantified

More Polymorphic Functions

```ml
fun identity x = x
fun diag x = (x,x)
fun swap(x,y) = (y,x)
fun append([],ys) = ys
    | append(x::xs, ys) = x :: append(xs,ys)
```
Datatypes

- The `datatype` mechanism generalizes:
  - Enumerated types
  - Tagged unions
  - Inductive types (lists, trees, etc.)

Enumerated Types

datatype day =
  Sunday | Monday | Tuesday | Wednesday |
  Thursday | Friday | Saturday

val weekdays : day list =
  [Monday, Tuesday, Wednesday, 
  Thursday, Friday]

fun isWeekend Saturday = true
  | isWeekend Sunday = true
  | isWeekend _ = false
Tagged Unions

• Suppose we want a list that can contain both integers and reals. First, define:

\[
\text{datatype} \ \text{num} = I \ \text{of} \ \text{int} \\
\ | \ R \ \text{of} \ \text{real}
\]

• Then

\[
\begin{align*}
I(5) & : \ \text{num} \\
R(5.0) & : \ \text{num} \\
R(3.1) & : \ \text{num}
\end{align*}
\]

\[
\text{val} \ \text{mymlist} : \ \text{num} \ \text{list} = \\
[I \ 3, \ R \ 4.0, \ R \ 1.1, \\
I(5+5), \ I \ ~17]
\]

Note: The addition operator + does not work on num's!
Tagged Unions

datatype num = INT of int
  | REAL of real

fun addnums (I n, I m) = I(n+m)
  | addnums (R r, R s) = R(r+s)
  | addnums (I n, R s) = R((Real.fromInt n) + s)
  | addnums (R r, I m) = R(r + (Real.fromInt m))

Tags and Enumerations

datatype color = Red | Orange | Yellow | Blue
  | Green | Indigo | Violet
  | RGB of int*int*int

  Red : color
  Indigo : color
  RGB(25,25,25) : color
Dataless Trees

datatype ttree = TLeaf
  | TNode of ttree*ttree

TLeaf : ttree
TNode(TLeaf,TLeaf) : ttree
TNode(TLeaf,TNode(TLeaf,TLeaf)) : ttree

fun nodes TLeaf = 0
  | nodes (TNode(left,right)) = 1 + (nodes left) + (nodes right)
Trees with Leaf Data

datatype itree = ILeaf of int
   | INode of itree*itree

ILeaf 3 : itree
INode(ILeaf 4, ILeaf 5) : itree

fun sumtree (ILeaf n) = n
   | sumtree (INode(left, right)) = (sumtree left)+(sumtree right)
Arithmetic Expressions

```
datatype exp = Num of real
             | Sum of exp*exp
             | Diff of exp*exp
```

Num 4.0 : exp
Sum(Num 3.0, Diff(Num 4.0, Num 1.0)) : exp

Exercise: define the function eval : exp -> real

Type Abbreviations

- The datatype construct always defines a new type
- We can also give shorter names to existing types

```
type ip = int * int

type bp = bool * bool
```

- Then ip is synonymous with int*int
- Similarly bp is interchangeable with bool*bool
Type Abbreviations with Parameters

• Definitions of types can be parameterized

\[
\text{type } \, 'a \text{ pair } = \, 'a \ast 'a
\]

• Then
  - string pair is synonymous with string*string
  - int pair = int*int = ip

Datatypes with Parameters

• Datatypes can also be parameterized

\[
\text{datatype } \, 'a \text{ tree } = \\
\text{ Leaf of } \, 'a \\
\text{ Node of } \, (\, 'a \, \text{ tree} \, ) \ast (\, 'a \, \text{ tree} \, )
\]

• Then

Leaf 5 : int tree
Node(Leaf true,Leaf false) : bool tree
Datatypes with Parameters

```haskell
datatype 'a tree =
    Leaf of 'a
    Node of ('a tree) * ('a tree)

fun collect (Leaf x) = [x]
| collect (Node(left,right)) =
    (collect left) @ (collect right)
```

Predefined Datatypes: option

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

NONE : int option
SOME(3) : int option

Int.fromString : string -> int option
```
Predefined Datatypes:  list

\[
\text{datatype} \quad 'a \text{ list} = \begin{align*}
\text{nil} \\
\mathbf{::} & \quad \text{of} \quad ('a \times 'a \text{ list})
\end{align*}
\]

\text{infix} \quad ::

nil  : int list
3::(4::nil) : int list

(The \([x, y, z]\) notation is built-in magic, however.)