Outline of the Course
(Version of 14 November 2005)

• Algorithm Analysis
• Sorting
• Stacks and Queues
• Trees
• Heaps
• Hashing
• Greedy Algorithms
• Graphs
• Constraint Processing
Revision

Things that should be known from the predecessor course:

• Specifications: types, pre-conditions, post-conditions
• Justifications: variants
• Recursion and tail-recursion
• Polymorphism
• Currying
• Higher-order functions
• Datatypes
• Exceptions
Polymorphism

Question: What is the type of the following function?

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

Answer:

`'a list -> int`

where `'a list` means that the function can take a list of anything:

to count the elements of a list, we do not have to know their type.

Polymorphism is a useful and important concept in SML:
it allows us to write functions only once,
but they can apply in a wide variety of situations.
A Non-Polymorphic Function & Currying

Question: What is the type of the following curried function?

```haskell
fun removeSmaller e [] = []
| removeSmaller e (x::xs) =
  if x < e then (removeSmaller e xs)
  else x::(removeSmaller e xs)
```

Answer:

```haskell
int -> int list -> int list
```

A declaration of a named function just declares a value identifier for an anonymous function:
functions are objects, just like numbers, strings, etc.
Polymorphism

SML always *infers* the most general type of an expression.

In the `removeSmaller` function, the fact that `<` is (by default) a function on integers forces the function to be on integer lists.

But the function would be the same if we used strings and compared them in alphabetical order!
The idea is to define a function that also takes a comparison function:

```haskell
fun removeSmallerGen compare e [] = []
| removeSmallerGen compare e (x::xs) =
  if compare(x,e) then (removeSmallerGen compare e xs)
  else x::(removeSmallerGen compare e xs)
```

The type of this higher-order function is:

```haskell
('a * 'b -> bool) -> 'a -> 'b list -> 'b list
```
Using `removeSmallerGen`

To use this function, we call it with a specific comparison function:

```haskell
fun removeSmallerInt e L = removeSmallerGen (op <) e L
```

Another way of doing this is:

```haskell
val removeSmallerInt = removeSmallerGen (op <)
```

The type of `removeSmallerInt` is:

```haskell
int -> int list -> int list
```

Why is the name fragment ‘`removeSmaller`’ inadequate now?
Exceptions

Exceptions are an important and useful mechanism in ML. They provide a way of dealing with error conditions. They can also be used to escape from local conditions: see the 8-Queens example page 100 in the Hansen & Rischel book.

```ml
exception NegativeInt

fun fact n =
  if n < 0 then raise NegativeInt
  else if n = 0 then 1
  else n * fact (n - 1)

where NegativeInt is an exception constructor.

What is a much better way of writing this function?
```
Catching Exceptions

To *catch* an exception, we need to use the `handle` construct:

```plaintext
fun factString n = Int.toString (fact n)
    handle NegativeInt => "Error: non-neg int expected!"
```

Usage:

- `factString 3 ;`
  > `val it = "6" : string`
- `factString ~3 ;`
  > `val it = "Error: non-neg int expected!" : string`

Most modern programming languages, such as C++, Java, Erlang, Scheme, ..., have some sort of exception mechanism.
Datatypes and Tagged Values

```ml
datatype answer = Yes | No
fun opposite Yes = No
  | opposite No = Yes
datatype shape = Circle of real | Square of real
fun area (Circle r) = Math.pi * r * r
  | area (Square a) = a * a

where Yes, No, Circle, and Square are value constructors,
just like the predefined :: (read cons) and nil (or []).

- area (Circle 1.0);
  > val it = 3.14159265359 : real
- area (Square 3.0);
  > val it = 9.0 : real
```
Recursive Datatypes

We will be using a lot of recursive datatypes in this course.

Variations on trees will come up a lot:

    datatype bTree = Void
                  | Node of int * bTree * bTree

Recursive datatypes require recursive functions:

    fun sum Void   = 0
    | sum (Node(x,t1,t2)) = x + sum t1 + sum t2

Is this function tail-recursive?
What is its variant and why does it terminate?
Parameterised/Polymorphic Recursive Datatypes

Example datatype:

```plaintext
datatype 'a myList = Empty
  | Cons of 'a * 'a myList
```

where `myList` is a type constructor, just like the predefined `list`.

Example function:

```plaintext
fun count Empty = 0
  | count (Cons(x,L)) = 1 + count L
```

What is the variant of this function and why does it terminate?
Is this function tail-recursive?
If not, then how to make it tail-recursive?