Pattern Matching

- We informally use pattern matching all the time in real life.

- Informally, a **pattern** is an expression containing **variables**, for which other expressions may be substituted. The problem of matching a pattern against a given expression consists of finding a suitable substitution that makes the pattern identical to the desired expression, if one exists at all.

- For example, we may apply the commutativity of +.

\[
x + y = y + x
\]
to the formula

\[
F = 1 + 2 + 3
\]
to obtain an equivalent formula

\[
3 + 2 + 1
\]
Here the “meta-variables” \(x\) and \(y\) were replaced by numbers. How?

Higher-Order Functions

- In functional programming languages, parameters may denote **functions** and be used in definitions of other, so-called **higher-order**, functions.

- One example of a higher-order function is the function apply defined below, which applies its first argument (a function) to all elements in its second argument (a list of suitable type).

```sml
- fun apply(f,L) =
  = if (L=[]) then []
  = else f(hd(L))::apply(f,tl(L));
val apply = fn : ("a -> 'b) * 'a list -> 'b list
```

We may apply apply with any function as argument.

```sml
- fun square(x) = (x:int)*x;
val square = fn : int -> int
- apply(square,[2,3,4]);
val it = [4,9,16] : int list
```

- The function `doubleall we defined may be considered a special case of supplying apply with first argument `double` (a function we defined in a previous lecture).

```sml
- apply(double,[1,3,5,7]);
val it = [2,6,10,14] : int list
```

- Look at the predefined SML map function.

Function Definition by Patterns

- In SML there is an alternative form of defining functions via **patterns**.

- The general form of such definitions is:

```sml
fun <identifier>(<pattern1>) = <expression1>
| <identifier>(<pattern2>) = <expression2>
| ...[
| <identifier>(<patternN>) = <expressionN>;
```

where the identifiers, which name the function, are all the same, all patterns are of the same type, and all expressions are of the same type.

- For example, an alternative definition of the reverse function is:

```sml
- fun reverse(nil) = nil
  = | reverse(x::xs) = reverse(xs) @ [x];
val reverse = fn : 'a list -> 'a list
```

- In applying such a function to specific arguments, the patterns are inspected in order and the **first match** determines the value of the function.
Mutual Recursion

- Sometimes the most convenient way of defining (two or more different) functions is in **mutual dependence of each other**.

- Consider the functions, `even` and `odd` that test if a number is even and odd. We can define them in the following way.

```sml
- fun even(0) = true
  = | even(m) = odd(n-1)
  = and
  = odd(0) = false
  = | odd(n) = even(n-1);
val even = fn : int -> bool
val odd = fn : int -> bool
```

SML uses the keyword and (not to be confused with the logical operator `and` also) for such mutually recursive definitions.

Neither of the two definition is acceptable by itself.

```sml
- even(2);
  val it = true : bool
- odd(3);
  val it = true : bool
```

- Consider two functions, `take` and `skip`, both of which extract alternate elements from a given list, with the difference that `take` starts with the first element (and hence extracts all elements at odd-numbered positions), whereas `skip` skips the first element (and hence extracts all elements at even-numbered positions, if any).

```sml
- fun take(L) =
  = if L = nil then nil
  = else hd(L)::skip(tl(L))
  = and
  = skip(L) =
  = if L=nil then nil
  = else take(tl(L));
val take = fn : 'a list -> 'a list
val skip = fn : 'a list -> 'a list

- take[1,2,3];
val it = [1,3] : int list
- skip[1,2,3];
val it = [2] : int list
```

Sorting

- We next design a function for **sorting a list of integers**.

- More precisely, we want to define an SML function, `sort : int list -> int list` such that `sort(L)` is a sorted version (in non-descending order) of `L`.

- Sorting is an important problem for which a large variety of different algorithms have been proposed.

- The method we will explore is based on the following idea. To sort a list `L`,
  - `first split L` into two disjoint sublists (of about equal size),
  - then (recursively) `sort` the sublists, and
  - finally `merge` the (now sorted) sublists.

This recursive method is known as **Merge-Sort**.

- It evidently requires us to define suitable functions for
  - splitting a list into two sublists and
  - merging two sorted lists into one sorted list.
• First we consider the problem of merging two sorted lists.  
• A corresponding recursive definition can be easily defined by distinguishing between the different cases, as to whether one of the argument lists is empty or not.  
• The following SML definition is formulated in terms of patterns (against which specific arguments in applications of the function will be matched during evaluation).

```sml
- fun merge([],M) = M
  | merge(L,[]) = L
  | merge(x::xl,y::yl) = if (x:int)<y then x::merge(xl,y::yl)
                           else y::merge(x::xl,y::yl);
val merge = fn : int list * int list -> int list
- merge([1,5,7,9],[2,3,5,5,10]);
val it = [1,2,3,5,5,7,9,10] : int list
- merge([],[1,2]);
val it = [1,2] : int list
- merge([1,2],[1]);
val it = [1,2] : int list
• How do we split a list? Recursion seems to be of little help for this task, but fortunately we have already defined suitable functions that solve the problem.
```

Finally, some examples:

```sml
- sort[];
val it = [] : int list
- sort[1];
val it = [1] : int list
- sort[1,2];
val it = [1,2] : int list
- sort[2,1];
val it = [1,2] : int list
- sort[1,2,3,4,5,6,7,8,9];
val it = [1,2,3,4,5,6,7,8,9] : int list
- sort[9,8,7,6,5,4,3,2,1];
val it = [1,2,3,4,5,6,7,8,9] : int list
- sort[1,2,1,2,2,2,2,2,2] : int list
```

### Merging

### Merge Sort

• Using take and skip to split a list, we obtain the following function for sorting.

```sml
- fun sort(L) =  
  | if L=[] then []
  | else merge(sort(take(L)),sort(skip(L)));
val sort = fn : int list -> int list
```

Don't run this function, though, as it doesn't quite work. Why?

• To see where the problem is, observe what the result is of applying take to a one-element list.

```sml
- take[1];
val it = [1] : int list
```

Thus in this case, the first recursive call to sort will be applied to the same argument!

• Here is a modified version in which one-element lists are dealt with correctly.

```sml
- fun sort(L) =  
  | if L=[] then []
  | else if tl(L)=[] then L
  | else merge(sort(take(L)),sort(skip(L)));
val sort = fn : int list -> int list
```

### Tracing Mergesort

• It is important to be able to trace the execution of the mergesort program to convince yourself that it works correctly.

![Mergesort Diagram](image_url)

• In the course of executing the recursive algorithm, the computer has to keep track of what work still needs to be done as it is interrupted with additional recursive calls.
**Tracing Mergesort**

How to split?

![Diagram of Mergesort](image)

**Tower Moves**

- First consider how many moves are needed, at the least, to transfer a tower of \( k \) disks.
- Observe that we need to get to the following intermediate configuration, so as to be able to move the largest disk.

![Diagram of Tower Moves](image)

That is, we have to transfer the \( k - 1 \) smaller disks to the middle pole, we can then move the largest disks from the first to the third pole, and finally the \( k - 1 \) smaller disks from the second pole to the third pole.

- Let \( M(k) \) be the minimum number of moves required to transfer \( k \) disks from one pole to another pole. This function \( M \) satisfies the recursive identity:
  \[
  M(k) = M(k-1) + 1 + M(k-1) = 2M(k-1) + 1, 
  \]
  for all \( k > 0 \).

In addition, we set \( M(0) = 0 \), so that by the above identity \( M(1) = 1 \), which is correct as one move suffices to transfer a tower containing only a single disk.

**The Tower of Hanoi**

- The tower of Hanoi consists of a fixed number of disks stacked on a pole in decreasing size, that is, with the smallest disk at the top.

![Diagram of Tower of Hanoi](image)

- There are two other poles and the task is to transfer all disks from the first to the third pole, one at a time without ever placing a larger disk on top of a smaller one.

- There is an elegant solution to this problem by recursion.

**Minimum Number of Moves**

- \( M(0) = 0 \)
  \[
  M(k) = M(k-1) + 1 + M(k-1) = 2M(k-1) + 1 \]
  for all \( k > 0 \).

- Let us evaluate the function for some arguments:
  \[
  M(0) = 0 \\
  M(1) = 2M(0) + 1 = 1 \\
  M(2) = 2M(1) + 1 = 3 \\
  M(3) = 2M(2) + 1 = 7 \\
  M(4) = 2M(3) + 1 = 15 \\
  M(5) = 2M(4) + 1 = 31 \\
  M(6) = 2M(5) + 1 = 63 \\
  \]

- The values grow fairly fast. In fact one can show that the function \( M \) can be explicitly defined by
  \[
  M(k) = 2^k - 1, 
  \]
  for all \( k \geq 0 \). That is, function values grow exponentially with the argument.

- This tells us that a lot of moves are needed to transfer a tall tower, though we don’t know the actual sequence of moves yet. For that purpose we will write an SML function.
**Tower of Hanoi in SML**

- Poles are represented by the numbers 1, 2, and 3.
- We represent a move as a pair of integers \((x, y)\). That is, \((x, y)\) is interpreted as moving a disk from pole \(x\) to pole \(y\). The pair \((x, y)\) is an example of a tuple of length 2 and type \(\text{int} \times \text{int}\).
- The function `tower` takes three integer arguments \(k, x\), and \(y\) such that \(k \geq 0\), \(1 \leq x \leq 3\) and \(1 \leq y \leq 3\). It returns a list of moves that transfer a tower of \(k\) discs from pole \(x\) to pole \(y\).

The result returned by `tower` is of type \(\text{int} \times \text{int}\) list.

- Here are some simple sequences of moves.

\[
\text{tower}(1,1,3);
\text{val it = }[(1,3)] \text{ : (int * int) list}
\]

\[
\text{tower}(2,2,2);
\text{val it = }[] \text{ : (int * int) list}
\]

and a few longer ones,

\[
\text{tower}(2,1,3);
\text{val it = }[(1,2),(1,3),(2,3)] \text{ : (int * int) list}
\]

\[
\text{tower}(3,1,3);
\text{val it = }[(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3)]
\text{ : (int * int) list}
\]

\[
\text{tower}(4,1,3);
\text{val it = }[(1,2),(1,3),(2,3),(1,2),(3,1),
(3,2),(1,2),(1,3),(2,3),(2,1),
(3,1),(2,3),(1,2),(1,3),(2,3)]
\text{ : (int * int) list}
\]

- The function `comp`, if provided with two of the numbers 1, 2, or 3 as arguments (2 poles), returns the third (pole).

\[
\text{fun comp}(x,y) = 6-(x+y);
\text{val comp = fn : int * int -> int}
\]

\[
\text{comp}(3,1);
\text{val it = 2 : int}
\]

- The function `tower` is defined by:

\[
\text{fun tower}(k, x, y) =
\begin{align*}
& \text{if } (k=0 \text{ or else } x=y) \text{ then } [] \\
& \text{else if } k=1 \text{ then } [(x,y)] \\
& \text{else tower}(k-1, x, \text{comp}(x,y)) \\
& \text{else tower}(k-1, x, \text{comp}(x,y))
\end{align*}
\]

\[
\text{val tower = fn : int * int * int}
\text{ -> (int * int) list}
\]

- The second line indicates that no move is needed if \(k = 0\) or the tower is to remain at the same pole.
- The third line provides an explicit solution for moving a tower of one disk.
- The fourth and fifth line show that in the general case we can

(a) move \(k-1\) disks from \(x\) to the “auxiliary” pole \(z\),
(b) move the largest disk from \(x\) to \(y\), and
(c) move \(k-1\) disks from \(z\) to \(y\).