Pattern Matching

- Haskell has a notation (called patterns) for defining functions that is more convenient than conditional (if-then-else) expressions.
- Patterns are particularly useful when the function has more than two cases.

Pattern Syntax:

function name pattern1 = expression1
function name pattern2 = expression2
... 
function name patternn = expressionn

Pattern Matching...

- Pattern matching allows us to have alternative definitions for a function, depending on the format of the actual parameter. Example:

```haskell
isNice "Jenny" = "Definitely"
isNice "Johanna" = "Maybe"
isNice "Chris" = "No Way"
```

Pattern Matching...

- Fact function in Haskell:

```haskell
fact n = if n == 0 then 1
        else n * fact (n-1)
```

Pattern Matching...

- Fact function revisited:

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```
We can use pattern matching as a design aid to help us make sure that we're considering all possible inputs.

Pattern matching simplifies taking structured function arguments apart. Example:

\[
\begin{align*}
\text{fun} \ (x:xs) &= x \oplus \text{fun} \ xs \\
\Leftrightarrow \\
\text{fun} \ xs &= \text{head} \ xs \oplus \text{fun} \ (\text{tail} \ xs)
\end{align*}
\]

When a function \( f \) is applied to an argument, Haskell looks at each definition of \( f \) until the argument matches one of the patterns.

\[
\begin{align*}
\text{not} \ True &= False \\
\text{not} \ False &= True
\end{align*}
\]

In most cases a function definition will consist of a number of mutually exclusive patterns, followed by a default (or catch-all) pattern:

\[
\begin{align*}
\text{diary} \ "Monday" &= "Woke \ up" \\
\text{diary} \ "Sunday" &= "Slept \ in" \\
\text{diary} \ \text{anyday} &= "Did \ something \ else"
\end{align*}
\]

\[
\begin{align*}
\text{diary} \ "Sunday" &\Rightarrow "Slept \ in" \\
\text{diary} \ "Tuesday" &\Rightarrow "Did \ something \ else"
\end{align*}
\]

There are several kinds of integer patterns that can be used in a function definition.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Syntax</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>var_name</td>
<td>fact ( n = \cdots )</td>
<td>( n ) matches any argument</td>
</tr>
<tr>
<td>constant</td>
<td>literal</td>
<td>fact ( 0 = \cdots )</td>
<td>( _ ) matches the value</td>
</tr>
<tr>
<td>wildcard</td>
<td>_</td>
<td>five ( _ = 5 )</td>
<td>( _ ) matches any argument</td>
</tr>
<tr>
<td>((n+k)) pat.</td>
<td>(n+k)</td>
<td>fact ( n+1 = \cdots )</td>
<td>((n+k)) matches any integer ( \geq k )</td>
</tr>
</tbody>
</table>
There are also special patterns for matching and (taking apart) lists.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Syntax</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>(x:xs)</td>
<td>(\text{len } (x:xs) = \cdots)</td>
<td>matches non-empty list</td>
</tr>
<tr>
<td>empty</td>
<td>[]</td>
<td>(\text{len } [] = 0)</td>
<td>matches the empty list</td>
</tr>
<tr>
<td>one-elem</td>
<td>[x]</td>
<td>(\text{len } [x] = 1)</td>
<td>matches a list with exactly 1 element.</td>
</tr>
<tr>
<td>two-elem</td>
<td>[x,y]</td>
<td>(\text{len } [x,y] = 2)</td>
<td>matches a list with exactly 2 elements.</td>
</tr>
</tbody>
</table>

The **sumlist** Function

Using conditional expr:

```haskell
sumlist :: [Int] -> Int
sumlist xs = if xs == [] then 0
             else head xs + sumlist(tail xs)
```

Using patterns:

```haskell
sumlist :: [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs
```

Note that patterns are checked top-down! The ordering of patterns is therefore important.

The **length** Function Revisited

Using conditional expr:

```haskell
len :: [Int] -> Int
len s = if s == [] then 0 else 1 + len (tail s)
```

Using patterns:

```haskell
len :: [Int] -> Int
len [] = 0
len (_:xs) = 1 + len xs
```

Note how similar **len** and **sumlist** are. Many recursive functions on lists will have this structure.

The **fact** Function Revisited

Using conditional expr:

```haskell
fact n = if n == 0 then 1 else n * fact (n-1)
```

Using patterns:

```haskell
fact' :: Int -> Int
fact' 0 = 1
fact' (n+1) = (n+1) * fact' n
```

Are **fact** and **fact'** identical?

- **fact** (-1) ⇒ Stack overflow
- **fact'** (-1) ⇒ Program Error

The second pattern in **fact'** only matches positive integers \(\geq 1\).
Functional languages use recursion rather than iteration to express repetition.

We have seen two ways of defining a recursive function: using conditional expressions (if-then-else) or pattern matching.

A pattern can be used to take lists apart without having to explicitly invoke head and tail.

Patterns are checked from top to bottom. They should therefore be ordered from specific (at the top) to general (at the bottom).

**Homework**

Define a recursive function `addints` that returns the sum of the integers from 1 up to a given upper limit.

```
Homework
```

Simulate the execution of `addints 4`.

```
addints :: Int -> Int
addints a = ···
```

```
? addints 5
15
? addints 2
3
```

**Homework**

Define a recursive function `member` that takes two arguments – an integer `x` and a list of integers `L` – and returns `True` if `x` is an element in `L`.

Simulate the execution of `member 3 [1,4,3,2]`.

```
member :: Int -> [Int] -> Bool
member x xs = ···
```

```
? member 1 [1,2,3]
True
? member 4 [1,2,3]
False
```

**Homework**

Write a recursive function `memberNum x xs` which returns the number of times `x` occurs in `xs`.

Use `memberNum` to write a function `unique xs` which returns a list of elements from `xs` that occurs exactly once.

```
memberNum :: Int -> [Int] -> Int
unique :: [Int] -> [Int]
```

```
? memberNum 5 [1,5,2,3,5,5]
3
? unique [2,4,2,1,4]
[1]
```
Ackerman’s function is defined for nonnegative integers:

\[
A(0, n) = n + 1 \\
A(m, 0) = A(m - 1, 1) \\
A(m, n) = A(m - 1, A(m, n - 1))
\]

Use pattern matching to implement Ackerman’s function.
Flag all illegal inputs using the built-in function \texttt{error S}
which terminates the program and prints the string \texttt{S}.

\[
\text{ackerman :: Int -> Int -> Int} \\
\text{ackerman 0 5 } \Rightarrow 6 \\
\text{ackerman (-1) 5 } \Rightarrow \text{ERROR}
\]