CSc 372
Comparative Programming Languages

23: Haskell — Patterns

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright (©) 2004 Christian Collberg
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:**

```haskell
function_name pattern_1 = expression_1
function_name pattern_2 = expression_2
... 
function_name pattern_n = expression_n
```
Pattern Matching...

\[
\text{fact } n = \begin{cases} 
1 & \text{if } n == 0 \\
\text{else} & n \times \text{fact} \ (n-1) 
\end{cases}
\]

\text{fact Revisited:}

\[
\text{fact :: Int -> Int} \\
\text{fact 0 = 1} \\
\text{fact n = n \times \text{fact} \ (n-1)}
\]
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...

- 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:

\[
\text{fun } (x:xs) = x \uplus \text{fun } xs \\
\iff \\
\text{fun } xs = \text{head } xs \uplus \text{fun } (\text{tail } xs)
\]
Pattern Matching...

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

not True = False
not False = True
In most cases a function definition will consist of a number of mutually exclusive patterns, followed by a default (or catch-all) pattern:

```plaintext
diary "Monday" = "Woke up"
diary "Sunday" = "Slept in"
diary anyday = "Did something else"

```

```plaintext
diary "Sunday" ⇒ "Slept in"
diary "Tuesday" ⇒ "Did something else"
```

Pattern Matching – Integer Patterns

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 = · · ·</td>
<td>n matches any argument</td>
</tr>
<tr>
<td>constant</td>
<td>literal</td>
<td>fact 0 = · · ·</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) = · · ·</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>len (x:xs) = \cdots</td>
<td>matches non-empty list</td>
</tr>
<tr>
<td>empty</td>
<td>[ ]</td>
<td>len [ ] = 0</td>
<td>matches the empty list</td>
</tr>
<tr>
<td>one-elem</td>
<td>[x]</td>
<td>len [x] = 1</td>
<td>matches a list with exactly 1 element.</td>
</tr>
<tr>
<td>two-elem</td>
<td>[x,y]</td>
<td>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:**

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

**Using patterns:**

```
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 \textbf{fact} Function Revisited

\underline{Using conditional expr:}
\begin{verbatim}
fact n = if n == 0 then 1 else n * fact (n-1)
\end{verbatim}

\underline{Using patterns:}
\begin{verbatim}
fact' :: Int -> Int
fact' 0 = 1
fact' (n+1) = (n+1) * fact' n
\end{verbatim}

Are \texttt{fact} and \texttt{fact'} identical?

\begin{itemize}
  \item \texttt{fact} (-1) \Rightarrow Stack overflow
  \item \texttt{fact'} (-1) \Rightarrow Program Error
\end{itemize}

The second pattern in \texttt{fact'} only matches positive integers \((\geq 1)\).
Summary

- 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.
- Simulate the execution of `addints 4`.

```haskell
addints :: Int -> Int
addints a = ...

? addints 5
 15

? addints 2
 3
```
Define a recursive function \texttt{member} that takes two arguments – an integer \texttt{x} and a list of integers \texttt{L} – and returns \texttt{True} if \texttt{x} is an element in \texttt{L}.

Simulate the execution of \texttt{member 3 [1,4,3,2]}.

\begin{verbatim}
member :: Int -> [Int] -> Bool
member x L = ...

? member 1 [1,2,3]  
  True

? member 4 [1,2,3]  
  False
\end{verbatim}
Homework...

- Write a recursive function \( \text{memberNum} \ x \ L \) which returns the number of times \( x \) occurs in \( L \).

- Use \( \text{memberNum} \) to write a function \( \text{unique} \ L \) which returns a list of elements from \( L \) that occurs exactly once.

\[
\begin{align*}
\text{memberNum} & : \text{Int} \to \text{[Int]} \to \text{Int} \\
\text{unique} & : \text{[Int]} \to \text{Int}
\end{align*}
\]

\[
\begin{align*}
? \ \text{memberNum} \ 5 \ [1,5,2,3,5,5] \\
& 3 \\
? \ \text{unique} \ [2,4,2,1,4] \\
& 1
\end{align*}
\]
Homework...

- Ackerman’s function is defined for nonnegative integers:

\[
egin{align*}
A(0, n) & = n + 1 \\
A(m, 0) & = A(m - 1, 1) \\
A(m, n) & = A(m - 1, A(m, n - 1))
\end{align*}
\]

- 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}.

```haskell
ackerman :: Int -> Int -> Int

ackerman 0 5 \Rightarrow 6
ackerman (-1) 5 \Rightarrow ERROR
```