CSc 520

Principles of Programming Languages

13: Haskell — Patterns

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 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:**

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

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

\[\text{fact Revisited:}\]

\[
\text{fact} :: \text{Int} \rightarrow \text{Int} \\
fact 0 = 1 \\
fact n = n \times \text{fact } (n-1)
\]
Pattern matching allows us to have alternative definitions for a function, depending on the format of the actual parameter. Example:

\[
\begin{align*}
\text{isNice} & \ "Jenny" = \ "Definitely" \\
\text{isNice} & \ "Johanna" = \ "Maybe" \\
\text{isNice} & \ "Chris" = \ "No Way"
\end{align*}
\]
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:

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

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

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

```
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) = ⋯</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 \texttt{length} Function Revisited

Using conditional expr:

\begin{verbatim}
len :: [Int] -> Int
len s = if s == [] then 0 else 1 + len (tail s)
\end{verbatim}

Using patterns:

\begin{verbatim}
len :: [Int] -> Int
len [] = 0
len (_:xs) = 1 + len xs
\end{verbatim}

Note how similar \texttt{len} and \texttt{sumlist} are. Many recursive functions on lists will have this structure.
The \texttt{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}

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

\begin{align*}
\text{fact' (-1)} & \Rightarrow \text{Stack overflow} \\
\text{fact' (-1)} & \Rightarrow \text{Program Error}
\end{align*}

\textbf{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).
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 `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]`.

```haskell
member :: Int -> [Int] -> Bool
member x L = ...

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

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

2. Use `memberNum` to write a function `unique L` which returns a list of elements from `L` 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 `error S` which terminates the program and prints the string `S`.

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