Haskell

COMP360
Haskell Reading

• Read the Haskell tutorial at learnyouahaskell.com
• Learn Haskell up to and including Higher order functions in chapter 6
Haskell Patterns

• You can write multiple definitions of a function
• Haskell will consider them in order and execute the first on that fits
• You can put constants as the parameters in the definition. If you call the function with that value, it will execute that function definition
Fibonacci Pattern Example

• You might have written the Fibonacci function as
  \[
  \text{fib } n = \begin{cases} 
  1 & \text{if } n \leq 2 \\
  \text{fib } (n-1) + \text{fib } (n-2) & \text{else}
  \end{cases}
  \]

• It can also be written as
  \[
  \begin{align*}
  \text{fib } 1 &= 1 \\
  \text{fib } 2 &= 1 \\
  \text{fib } n &= \text{fib } (n-1) + \text{fib } (n-2)
  \end{align*}
  \]
Empty List Pattern

• You can specify an empty list as a parameter. The definition will be used if it is executed with an empty list

mySum :: Num a => [a] -> a
mySum [] = 0
mySum str = head str + mySum (tail str)
Practice

• Write a Haskell method to convert a string containing a binary number to an int number

    bin2dec "0101" returns 5

    sumList :: [Char] -> int
Splitting Complex Values

• The parameters to a function can be specified by their parts
• A list parameter can be specified as (cat:dog)
• The "::" operator is normally used to add a new element to the beginning of a list
• Here it specifies the input list is composed of a head named cat and the tail named dog
Using Split Parameters

• Consider a function that uses the head and tail of a list
• You can describe the list as a head:tail

\[
\text{sumsqr} :: \text{Num} \ a \Rightarrow [a] \rightarrow a \\
\text{sumsqr} \ [] = 0 \\
\text{sumsqr} \ (\text{cat}:\text{dog}) = \text{cat} \times \text{cat} + \text{sumsqr} \ \text{dog}
\]
Values in Split Parameters

• You can put a constant in part of a split parameter to make a pattern

mytrim (' ' : cat) = mytrim cat
mytrim dog = dog
And More

• You can split an input parameter into more than two parts

  myFunc (ant : bird : cat) = bird

• This returns the second element of a list
• (ant : bird : cat) can be read as a list where ant is the head of a list where bird is the head and cat the tail
Don't Care Variables

• Sometimes you are not going to use a part of the complex type, but need to specify that is there to completely describe the type

• The variable name _ or underscore represents a variable that will not be used

myFunc ( _ : bird : _ ) = bird
Practice

• Write a method that takes the union of two lists
• Note that the union should not contain duplicates
Guards

• You can put conditions on the parameters of a function

grade score
| score >= 90 = "A"
| score >= 80 = "B"
| score >= 70 = "C"
| otherwise = "You flunk"
Calculated Guards

• The guard equations can involve more complicated equations

<table>
<thead>
<tr>
<th>grade</th>
<th>right</th>
<th>total</th>
<th>condition</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>100.0</td>
<td>total</td>
<td>&gt;= 90</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>80</td>
<td>100.0</td>
<td>total</td>
<td>&gt;= 80</td>
<td>&quot;B&quot;</td>
</tr>
<tr>
<td>70</td>
<td>100.0</td>
<td>total</td>
<td>&gt;= 70</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>otherwise</td>
<td>&quot;You flunk&quot;</td>
</tr>
</tbody>
</table>
Simplified Guards

• Equations used in the conditions of a statement can be defined to simplify the guards

\[
\text{grade right total}
\begin{align*}
| & \text{score } \geq 90 = "A" \\
| & \text{score } \geq 80 = "B" \\
| & \text{score } \geq 70 = "C" \\
| & \text{otherwise } = "You flunk"
\end{align*}
\]

where score = \(100.0 \times \text{right} / \text{total}\)
Temporary Bindings in Functions

• You can specify variable values in an expression before the definition of a function

```plaintext
functionName parm = let what = ever in function def

border len width = let parim = 2 (len + width) in 3 * parim
```
Haskell Reading

• Read the Haskell tutorial at learnyouahaskell.com
• Learn Haskell up to and including Higher order functions in chapter 6