Function Application

In Haskell, function application has precedence over all other operations. Since the compiler knows how many arguments each function requires, we can do the following:

\[
\begin{align*}
\text{func1 } x &= x + 2 \\
\text{func2 } x \ y &= x^*x + y^y \\
\text{func3 } a \ b &= \text{func1 } a + \text{func2 } b \ a
\end{align*}
\]

No parentheses are necessary – the known signatures of func1 and func2 define how to compute func3.

If – Then - Else

Remember that execution of purely functional Haskell code involves only function evaluation, and nothing else.

Therefore, there is an if – then – else function, but it always has to return something, so we always need the "else" part.

Furthermore, it needs to have a well-defined signature, which means that the expressions following "then" and "else" have to be of the same type.

Some Functions on Lists

- head \(xs\): Returns the first element of list \(xs\)
- tail \(xs\): Returns list \(xs\) with its first element removed
- length \(xs\): Returns the number of elements in list \(xs\)
- reverse \(xs\): returns a list with the elements of \(xs\) in reverse order
- null \(xs\): returns true if \(xs\) is an empty list and false otherwise

Some Functions on Tuples

- fst \(p\): Returns the first element of pair \(p\)
- snd \(p\): Returns the second element of pair \(p\)

This only works for pairs, but you can define your own functions for larger tuples, e.g.:

\[
\begin{align*}
\text{fst3 } (a, b, c) &= a \\
\text{fst3 } (x, y, z) &= x
\end{align*}
\]

You can always replace variables whose values you do not need with an underscore:

\[\text{fst3}(x, _, _) = x\]
Pattern Matching
You can define separate output expressions for distinct patterns in the input to a function. This is also the best way to implement recursion, as in the factorial function:

\[
\text{fact } \emptyset = 1 \\
\text{fact } n = n \times \text{fact } (n - 1)
\]

Similarly, we can define recursion on a list, for example, to compute the sum of all its elements:

\[
\text{sum}' [] = 0 \\
\text{sum}' (x:xs) = x + \text{sum}' xs
\]

Guards
In pattern matching, you have to specify exact patterns and values to distinguish different cases.

If you need to check inequalities or call functions in order to make a match, you can use guards instead:

\[
\text{iqGuards} :: \text{Int} \rightarrow [\text{Char}] \\
\text{iqGuards } n \\
\begin{align*}
| & n > 150 = \text{"amazing!"} \\
| & n > 100 = \text{"cool!"} \\
| & \text{otherwise} = \text{"oh well..."}
\end{align*}
\]

Laziness of Haskell
One important thing to know about Haskell is that it performs lazy evaluation of expressions.

Here, "lazy" means that expressions are only evaluated when absolutely necessary for the current computation.

For example, this allows us to work with infinite lists without getting stuck in an infinite computation (\text{take} \ n \ xs \ returns \ the \ first \ n \ elements \ of \ list \ xs):

\[
\text{>take 10 [1..]} \\
[1,2,3,4,5,6,7,8,9,10]
\]

Some Functions We Wrote in Class
\[
\text{euclid } x \ y = s qX + s qY \\
\text{where } s qX = x \times x \\
\text{sqY} = y \times y \\
\text{isPrime } n = n > 1 \land \text{null } [x | x <- [2 .. \text{div} \ n \ 2], \mod n x == 0] \\
\text{myzip } [] \ [] = [] \\
\text{myzip } (x:xs) \ [] = [] \\
\text{myzip } [] \ (y:ys) = [] \\
\text{myzip } (x:xs) \ (y:ys) = (x, y):\text{myzip } xs \ ys
\]

Reading
For this course, you should understand the material in "Learn you a Haskell" in Chapters 1-6, Chapter 8 until the end of "type parameters" and the "Hello, World!" section of Chapter 9.

Please read this material and experiment with it as far as you get. In class we will cover most of it and work on some coding examples.

Then you will be ready to tackle AI problems with some powerful programming tools.