Learning Frege

When you study the text “Learn you a Haskell,” you need to be aware of the subtle differences between Frege and Haskell. These differences exist to make Frege compatible with Java, particularly its data types.

Here are useful explanations specifically for that text: https://github.com/Frege/frege/wiki/LYAH-adaptions-for-Frege

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:

```
func1 x = x + 2
func2 x y = x*x + y*y
func3 a b = func1 a + func2 b a
```

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.:

```
fst3 :: (a, b, c) -> a
fst3 (x, y, z) = x
```

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

```
fst3(x, _, _) = x
```
Some Functions on Tuples

The Cartesian product is a good example of how to build lists of tuples using list comprehensions:

\[
\text{cart } \{x\} \times \{y\} = [(x, y) \mid x \leftarrow \{x\}, y \leftarrow \{y\}]
\]

> cart [1, 2], ['a', 'b', 'c']

> [(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]

Note that the assignment \(y \leftarrow \{y\}\) first goes through all elements of \(\{y\}\) before the value of \(x\) changes for the first time. The changes are “fastest” for the rightmost assignment and propagate to the left, like the digits of a decimal number when we count, e.g., from 100 to 999.

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:

\[
\text{fact } 0 = 1
\]
\[
\text{fact } n = n \cdot \text{fact } (n - 1)
\]

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

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

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:

\[
\text{fact } 0 = 1
\]
\[
\text{fact } n = n \cdot \text{fact } (n - 1)
\]

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

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

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:

\[
\text{fact } 0 = 1
\]
\[
\text{fact } n = n \cdot \text{fact } (n - 1)
\]

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

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

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:

\[
\text{fact } 0 = 1
\]
\[
\text{fact } n = n \cdot \text{fact } (n - 1)
\]

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

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

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:

\[
\text{fact } 0 = 1
\]
\[
\text{fact } n = n \cdot \text{fact } (n - 1)
\]

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:

\[
\text{iqGuards} :: \text{Int} \rightarrow \text{String}
\]
\[
\text{iqGuards } n
\]
\[
| n > 150 = \text{"amazing!"}
\]
\[
| n > 100 = \text{"cool!"}
\]
\[
| \text{otherwise} = \text{"oh well..."}
\]
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 all of it and work on some coding examples.

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

Please use the "apply" command in the UNIX system to get a directory for submitting your homework.