Pattern Matching

Pattern matching allows us to do things like this:

```haskell
testList [] = "empty"
testList (x:[]) = "single-element"
testList (x:xs) = "multiple elements. First one is " ++ show x
```

The last line demonstrates how we can use pattern matching to not only specify a pattern but also access the relevant input elements in the function expression. Since we do not use xs in that line, we could as well write

```haskell
testList (x:_) = "multiple elements. First one is " ++ show x
```

Haskell’s syntax allows us to write a quicksort algorithm very concisely and clearly:

```haskell
quickSort [] = []
quickSort (x:xs) = quickSort low ++ [x] ++ quickSort high
  where low = [y | y <- xs, y < x]
       high = [y | y <- xs, y >= x]
```

Note, though, that this quicksort algorithm is not as fast as if we had implemented it, for example, in C with elements remaining in the same memory block.

Recursion

Since variables in Haskell are immutable, our only way of achieving iteration is through recursion. For example, the reverse function receives a list as its input and outputs the same list but with its elements in reverse order:

```haskell
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Currying

As you know, you can turn any infix operator into a prefix operator by putting it in parentheses:

```haskell
(+) 3 4
7
```

Now currying allows us to place the parentheses differently:

```haskell
(+ 3) 4
7
```

By “fixing” the first input to (+) to be 3, we created a new function (+ 3) that receives only one (further) input.

Lambda Expressions

Sometimes we just need a small local function that is not used anywhere else in the program. Then it is convenient to use a lambda expression, which is a way of defining an anonymous function directly in the place where we use it. They have the following form:

`\input -> output`

For example:

`\x -> 3*x`

4

12
Lambda Expressions

More examples for lambda expressions:

- zipWith (+) [1..10] [11..20]
- [12,14,16,18,20,22,24,26,28,30]
- zipWith (\x y \rightarrow x^2 + y^2) [1..10] [11..20]
- [122,148,178,212,250,292,338,388,442,500]
- map (\x \rightarrow (x, x^2, x^3)) [1..10]
- [(1, 1, 1),(2, 4, 8),(3, 9, 27),(4, 16, 64),(5, 25, 125)]

Folds

Folds take a binary function f, a start value z, and a list:

- foldr f z [] = z
- foldr f z (x:xs) = f x (foldr f z xs)
- foldl f z [] = z
- foldl f z (x:xs) = foldl f (f z x) xs

Folds can be used to build a variety of functions, e.g.:

- sum = foldl (+) 0

(Point-free style: The last argument xs is identical on both sides of the equation and can be omitted.)

foldl Expression Evaluation

- foldl (+) 0 [1, 2, 3]
- foldl (+) (+) 0 1) [2, 3]
- foldl (+) 1 [2, 3]
- foldl (+) (+) 1 2) [3]
- foldl (+) 3 [3]
- foldl (+) (+) 3 3) []
- foldl (+) 6 []
- 6

The $ Operator

The $ operator is defined as follows:

- f $ x = f x

It has the lowest precedence, and therefore, the value on its right is evaluated first before the function on its left is applied to it.

As a consequence, it allows us to omit parentheses:

- negate (sum (map sqrt [1..10]))

Can be written as:

- negate $ sum $ map sqrt [1..10]

Function Composition

Similarly, we can use function composition to make our code more readable and to create new functions. As you know, in mathematics, function composition works like this:

- \( (f \circ g)(x) = f(g(x)) \)

In Haskell, we use the "." character instead:

- map (\xs \rightarrow negate (sum (tail xs)))
- [[1..5],[3..6],[1..7]]

Can be written as:

- map (negate . sum . tail)
- [[1..5],[3..6],[1..7]]
Data Types

Data types can be declared as follows:

```haskell
data Bool = false | true
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
  deriving Show
```

Then we can construct values of these types like this:
```
x = Circle 3 4 5
```

The addition `deriving Show` makes these values printable.

---

Records

If we want to name the components of our data types, we can use records:

```haskell
data Car = Car {company :: String, model :: String, year :: Int}
  deriving Show
myCar = Car {company="Ford", model="Mustang",
  year=1967}

company myCar
  "Ford"
```

---

Tree Data Type Example

```haskell
data BinaryTree a = 
  Leaf a | Branch (BinaryTree a) a (BinaryTree a)
mytree = Branch (Branch (Leaf 2) 7 (Leaf 5)) 3
  (Branch (Branch (Leaf 1) 3 (Leaf 4)) 8 (Leaf 9))
```

---

Tree Data Type Example (2)

```haskell
nodeCount (Leaf x) = 1
nodeCount (Branch left x right) =
  1 + nodeCount left + nodeCount right

treeHeight (Leaf x) = 0
treeHeight (Branch left x right) =
  1 + max (treeHeight left) (treeHeight right)
tree2list (Leaf x) = [x]
tree2list (Branch left x right) =
  (tree2list left) ++ [x] ++ (tree2list right)
foldTree f z (Leaf x) = f x z
foldTree f z (Branch left x right) =
  foldTree f (f (x (foldTree f z right)) left
```

---

Input/Output with “do”

Purely functional code cannot perform user interactions such as input and output, because it would involve side effects. Therefore, we sometimes have to use impure functional code, which needs to be separated from the purely functional code in order to keep it (relatively) bug-safe. In Haskell, this is done by so-called Monads. To fully understand this concept, more in-depth study is necessary. However, in this course, we do not need to perform much input and output. We can use a simple wrapper (or “syntactic sugar”) for this – the “do” notation.
Input/Output with "do"

In a do-block, we can only use statements whose type is "tagged" IO so that they cannot be mixed with purely functional statements.

Example for a program performing input and output:

```haskell
main = do
  putStrLn "Hello, what's your name?"
  name <- getLine
  putStrLn ("Hey " ++ name ++ ", you rock!")
```

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.

A Trip to Grid-Space World

• Grid-space world is an extremely simple model of our own world.
• It is a three-dimensional space with a floor that is divided into cells by a two-dimensional grid.
• The cells can be empty or contain objects or agents.
• There can be walls between sets of cells.
• The agents are confined to the floor and can move from cell to cell.
• A robot in grid-space world can sense whether neighboring cells are empty or not.

Perception and Action

Organisms in the real world have to do two basic things in order to survive:
• They have to gather information about their environment (perception) and
• based on this information, they have to manipulate their environment (including themselves) in a way that is advantageous to them (action).

The action in turn may cause a change in the organism's perception, which can lead to a different type of action.

We call this the perception-action cycle.

Perception and Action

Complex organisms do not just perceive and act, but they also have an internal state that changes based on the success of previous perception-action cycles. This is the mechanism of learning.

We will first consider a very simple robot that lives in grid-space world and has no internal state.

The grid has no tight spaces, that is, spaces between objects and boundaries that are only one cell wide.
Perception and Action

The robot can move to a free adjacent cell in its column or row. Consequently, there are four possible actions that it can take:

- **north**: moves the robot one cell up
- **east**: moves the robot one cell to the right
- **south**: moves the robot one cell down
- **west**: moves the robot one cell to the left

Immediate Perception-Action

Now that we specified the robot’s capabilities, its environment, and its task, we need to give “life” to the robot. In other words, we have to specify a function that maps sensory inputs to movement actions so that the robot will carry out its task. Since we do not want the robot to remember or learn anything, one such function would be sufficient. However, it is useful to decompose it in the following way (next slide):

Immediate Perception-Action

The functional decomposition has two advantages:

- **Multiple action functions** can be added that receive the same feature vector as their input,
- It is possible to add an internal state to the system to implement memory and learning.

The Robot’s Perception

For our robot, we define four different features $x_1, \ldots, x_4$ that are important to it. Each feature has value 1 if and only if at least one of the shaded cells is not free:

Immediate Perception-Action

The robot’s action:

- **move north** if $x_1 = 0$ and $x_2 = 0$ and $x_3 = 0$ and $x_4 = 0$
- **move east** if $x_1 = 1$ and $x_2 = 0$
- **move south** if $x_2 = 1$ and $x_3 = 0$
- **move west** if $x_2 = 1$ and $x_1 = 0$
Production Systems

- **Production systems** are a standardized way to represent action functions.
- A production system consists of an ordered list of production rules (productions).
- Each rule is written in the form condition \( \rightarrow \) action.
- A production system is therefore written like:
  - \( c_1 \rightarrow a_1 \)
  - \( c_2 \rightarrow a_2 \)
  - \( \ldots \)
  - \( c_m \rightarrow a_m \)
- The action of the first rule whose condition evaluates to 1 is executed.

Using Boolean notation, the production system for our boundary-following robot looks like this:

\[
\begin{align*}
x_4x_1 & \rightarrow \text{north} \\
x_3x_4 & \rightarrow \text{west} \\
x_2x_3 & \rightarrow \text{south} \\
x_1x_2 & \rightarrow \text{east} \\
1 & \rightarrow \text{north}
\end{align*}
\]