Hugh Baldwin
up2157117

Discrete Mathematics and Functional Programming

M21274
TB2

University of Portsmouth
BSc Computer Science
2nd Year

Contents

Part I

Discrete Mathematics

Lecture - Sets

17:00 23/01/24 Janka Chlebikova

A set is a collection of objects, known as elements or members (I will stick to members). Each member only appears once in the set. There is no particular order for members of a set, so there are several different ways to represent the same set. The members of a set can be just about anything, as long as they all abide by the same rules, and are in some way related.

Notation

There are several ways of noting a set, such as writing out all of the members of the set, or by using a rule which describes all of the members of a set.

For example, the following sets are equivalent

- $A = \{1, 2, 3, 4, 5\}$
- $A = \{x \mid 0 < x \leq 5\}$

If the object, x is in the set S, you would write it as $x \in S$. If not, it would be written as $x \notin S$ You can also describe a set by specifying a property that the members share, e.g.

- $B = \{3, 6, 9, 12\}$
- $B = \{x \mid x \text{ is a multiple of } 3, \text{ and } 0 < x \leq 15\}$
- $$\begin{aligned} S &= \{\dots, -3, -1, 1, 3, \dots\} \\ &= \{x \mid x \text{ is an odd integer}\} \\ &= \{x \mid x = 2k + 1 \text{ for some integer } k\} \\ &= \{x \mid x = 2k + 1 \text{ for some } k \in \mathbb{Z}\} \\ &= \{2k + 1 \mid k \in \mathbb{Z}\} \end{aligned}$$

The Number Sets

Some letters are reserved for specific sets of numbers which can be used elsewhere to simplify definitions, the following are the most commonly used number sets

- $\mathbb{N}$ is used for the set of natural numbers, $\mathbb{N} = \{0, 1, 2, 3, 4, \dots\}$
- $\mathbb{Z}$ is used for the set of integers, $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
- $\mathbb{Q}$ is used for the set of rational numbers, $\mathbb{Q} = \{0, \frac{1}{2}, \frac{1}{3}, \dots\}$
- There is also the empty or null set, $\emptyset$ which contains no items, so $\emptyset = \{\}$

A set can either by finite or infinite, and the cardinality of a set is the number of members, e.g. $|S| =$ the number of members of S. For example, $\mathbb{N}$ and $\mathbb{Z}$ are infinite sets, and the set $A = \{1, 2, 3\}$ is a finite set with a cardinality of 3, so $|A| = 3$

Subsets

If every member of A is also a member of B, A is said to be a subset of B, which can be written as $\mathrm{A} \subseteq \mathrm{B}$. If B also has at least 1 member which is not a member of A, then A is a proper subset of B, which can be written as $\mathrm{A} \subset \mathrm{B}$. If A is not a subset of B, it can be written as $\mathrm{A} \not\subseteq \mathrm{B}$. Since the null set, $\emptyset$ contains no elements, it is a subset of every other set.

Equality of Sets

If two sets, A and B are equal, they have exactly the same members, which can be written as $\mathrm{A} = \mathrm{B}$. Alternatively, $\mathrm{A} = \mathrm{B}$ if the following conditions are true:

- $\mathrm{A} \subseteq \mathrm{B}$, and so for each x, if $x \in \mathrm{A}$ then $x \in \mathrm{B}$
- $\mathrm{B} \subseteq \mathrm{A}$, and so for each y, if $y \in \mathrm{B}$ then $y \in \mathrm{A}$

Operations

The intersection of two sets, A and B is every member in both sets, $\mathrm{A} \cap \mathrm{B}$. For example, the intersection of the sets $\mathrm{X} = \{1, 2, 3, 4, 5\}$ and $\mathrm{Y} = \{4, 5, 6, 7, 8\}$ is $\mathrm{X} \cap \mathrm{Y} = \{4, 5\}$. If there are no common members, then the two sets are said to be disjoint. You can remember this by the fact that $\cap$ looks like an n, and therefore is the I**n**tersection of two sets.

The union of two sets, A and B is every member in either set, $\mathrm{A} \cup \mathrm{B}$. For example, the union of the sets $\mathrm{X} = \{1, 2, 3, 4, 5\}$ and $\mathrm{Y} = \{4, 5, 6, 7, 8\}$ is $\mathrm{X} \cup \mathrm{Y} = \{1, 2, 3, 4, 5, 6, 7, 8\}$. You can remember this by the fact that $\cup$ looks like a U, and therefore is the **U**nion of two sets.

The difference of two sets, A and B are all members of the first set which are not members of the second set, $\mathrm{A} \backslash \mathrm{B}$. For example, the difference of the sets $\mathrm{X} = \{1, 2, 3, 4, 5\}$ and $\mathrm{Y} = \{4, 5, 6, 7, 8\}$ is $\mathrm{X} \backslash \mathrm{Y} = \{1, 2, 3\}$. This is the effectively subtracting the sets, $\mathrm{X} - \mathrm{Y}$.

If we consider all of the sets to be a subset of a particular set, U which contains all of the members of the "Universe of Discourse", then the complement of a set, A is any members of U which are not in A. This is represented as either A' or $\overline{\mathrm{A}}$

All of these operations can be represented using a Venn diagram.

Like binary arithmetic, these operations follow a few rules:

- Commutative - $\mathrm{A} \cup \mathrm{B} = \mathrm{B} \cup \mathrm{A}$ and $\mathrm{A} \cap \mathrm{B} = \mathrm{B} \cap \mathrm{A}$
- Associative - $(\mathrm{A} \cup \mathrm{B}) \cup \mathrm{C} = \mathrm{A} \cup (\mathrm{B} \cup \mathrm{C})$ and $(\mathrm{A} \cap \mathrm{B}) \cap \mathrm{C} = \mathrm{A} \cap (\mathrm{B} \cap \mathrm{C})$
- Distributive - $\mathrm{A} \cap (\mathrm{B} \cup \mathrm{C}) = (\mathrm{A} \cap \mathrm{B}) \cup (\mathrm{A} \cap \mathrm{C})$ and $\mathrm{A} \cup (\mathrm{B} \cap \mathrm{C}) = (\mathrm{A} \cup \mathrm{B}) \cap (\mathrm{A} \cup \mathrm{C})$
- de Morgan's - $(\mathrm{A} \cap \mathrm{B})' = \mathrm{A}' \cup \mathrm{B}'$ and $(\mathrm{A} \cup \mathrm{B})' = \mathrm{A}' \cap \mathrm{B}'$

To get the cardinality of the union of two finite sets, you might think it would just be $|\mathrm{A} \cup \mathrm{B}| = |\mathrm{A}| + |\mathrm{B}|$, however, this results in counting $|\mathrm{A} \cap \mathrm{B}|$ twice, and so the correct cardinality is $|\mathrm{A} \cup \mathrm{B}| = |\mathrm{A}| + |\mathrm{B}| - |\mathrm{A} \cap \mathrm{B}|$

The Power Set

The power set is a set containing all subsets of the set, so if $\mathrm{S} = \{a, b, c\}$, then the power set $\mathrm{P}(\mathrm{S})$ would be $\mathrm{P}(\mathrm{S}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$. If the set S has n members, $\mathrm{P}(\mathrm{S})$ has 2^n members.

Partitions

The collection of nonempty subsets of S is a **partition** of the set S, if and only if every element in S belongs to exactly one member of the partition. This means that the partition sets are mutually disjoint, and the union of all sets in the partition is equal to S

Lecture - Relations

17:00 30/01/24 Janka Chlebikova

Ordered Pairs

A set is an unordered collection of members, but sometimes it is useful to consider the order of members. In this case, you can use an ordered pair, which is two members written as (a, b). Since they are ordered pairs, (a, b) is distinct from (b, a)

Cartesian Product

With the sets A and B, $A \times B$ is the Cartesian product, where $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$. This is the set of all ordered pairs, in which the first item is from the first set, and the second item from the second set. For example, if $X = 1, 2, 3$ and $Y = a, b$, then $A \times B = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$

Relations

If A is the set of all students taking DMaFP and B is the set of all modules offered by the School of Computing, then the relation T can be defined between A and B as "If the student, $x \in A$ is registered on the module, $y \in B$ then x is related to y by the relation T", e.g. $(\text{Hugh Baldwin}, \text{Ethical Hacking}) \in T$. The order matters, as T is a relation from the set A to the set B

To put it another way, if a set is a subset of the Cartesian product of A and B, then it is a relation between A and B. If $T \subseteq A \times B$ and $(a, b) \in T$, we can say that a is related to b by T, and therefore aTb.

Relations can also be described "by the characteristics of their members". For example, if $A = \{1, 2\}$ and $B = \{1, 2, 3\}$, we can define a relation from A to B as follows: $x \in A \text{ is related to } y \in B \text{ if and only if } x \leq y$. With this definition, we can see that $(1, 3) \in R$ since $1 < 3$, but $(2, 1) \notin R$ since $2 > 1$. The full list of members is $R = \{(1, 1), (1, 2), (1, 3), (2, 2), (2, 3)\}$.

If $A = B$, then a relation **on** A is a relation from A to A, and so is a subset of $A \times A$.

Basic Properties of Relations

Reflexivity

A relation is reflexive, if and only if $(x, x) \in R$ for all $x \in A$. For example, the relation

$$R = \{(1, 1), (1, 2), (1, 3), (2, 2), (3, 3)\}$$

on the set $A = \{1, 2, 3\}$ is reflexive. Another example is that the relation $S = (x, y) \mid x, y \in A \text{ and } x \leq y$ is reflexive on the set $A = \{1, 2, 3\}$ since $(1, 1), (2, 2), (3, 3) \in S$

Symmetry

A relation is symmetric, if and only if for all $x, y \in A$, if $(x, y) \in R$ then $(y, x) \in R$

Transitivity

A relation is transitive, if and only if for all $x, y, z \in A$, if $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$

Equivalence

A relation is an equivalence relation if and only if it is Reflexive, Symmetric and Transitive. Suppose that A is a set and R is an equivalence relation on A, for each element $a \in A$, the equivalence class of a, $[a]$ is the set of all elements in A such that x is related to a by R: $[a] = \{x \mid x \in A \text{ and } (x, a) \in R\}$. Since R must be a symmetric relation, we can also write $(a, x) \in R$. For example, for the set $A = \{0, 1, 2, 3\}$ and relation $R = \{(0, 0), (1, 1), (1, 3), (2, 2), (3, 3), (3, 1)\}$ the equivalence class for 1 is $[1] = \{x \mid x \in A \text{ and } (x, 1) \in R\} = \{1, 3\}$. A set of the equivalence classes is also a partition of the set.

Lecture - Functions

17:00 07/02/24 Janka Chlebikova

Functions

A function is a special type of relation, specifically one in which each member of the input set is related to at most one member of the output set. This is known as a function from A to B. A more formal definition of a function is "With the non-empty sets A and B, a function from A to B, $f : A \to B$ is a relation from A to B such that for each $x \in A$ there is exactly one element in B, $f(x) \in B$, associated with x by relation f"

Total and Partial Functions

A total function is one in which every member of the input set, A, has a corresponding member in the output set, B. For example, the function $f : \mathbb{Z} \to \mathbb{Z}$ defined by $f(x) = 2x$ is a total function, as every possible integer has a corresponding integer output.

For a partial function on the other hand, each member of the input set A may or may not have a corresponding member in the output set B. An example of this is the function $f : \mathbb{Z} \to \mathbb{Q}$ defined by $f(x) = \frac{1}{x}$ is a partial function as the input value 0 has no defined output, since $\frac{1}{0}$ is undefined.

Domain, Co-Domain and Range

The domain of a function is the set of all inputs for which there is a defined output. This is the case with both total and partial functions, as the domain is a subset of the input set, e.g. $D \subset A$. In the case of a total function, $D \subseteq A$ and therefore $D = A$, but for a partial function $D \subset A$. For the total function $f : \mathbb{Z} \to \mathbb{Z}$ defined by $f(x) = x^2$, the domain is $\mathbb{Z}$, but for the partial function $g : \mathbb{Z} \to \mathbb{Q}$ defined by $g(x) = \frac{1}{x}$, the domain is $\mathbb{Z} - \{0\}$ as $\frac{1}{0}$ is still undefined.

The co-domain of a function is the domain of the output. e.g. for the function $f : A \to B$, the domain is A and co-domain is B. B contains all possible outputs of the function, as well as any other members of B.

The range of a function is the subset of the co-domain for which each member is associated with an input member of the domain, and therefore $\mathrm{range}(f) \subset \mathrm{domain}(f)$. e.g. $\mathrm{range}(f) = \{f(x) \mid x \in A\}$.

Function Properties

There are three main properties which a function can be: injective, surjective and/or bijective.

Injective

A function is injective (or one-to-one) if it maps each member of the input set A to a unique member of the output set B. So, for all $x, y \in A$, if $x \neq y$ then $f(x) \neq f(y)$. A function cannot be injective if there is some two values, $x, y \in A$ for which $f(x) = f(y)$.

Surjective

A function is surjective if the range of the function is equal to it's co-domain. e.g. for the function $f : A \to B$, $\mathrm{range}(f) = B$. To put it more technically, for all $y \in B$ there exists $x \in A$ such that $f(x) = y$.

Bijective

A function is bijective if it is both injective and surjective.

Composite Functions

A composite function is one which is defined in terms of another. For the functions $f : A \to B$ and $g : B \to C$, the composition of g with f is the function $g \circ f$ such that $g \circ f : A \to C$ and is defined by $(g \circ f)(x) = g(f(x))$ for all $x \in A$ and therefore the value of $f(x)$ must be calculated before that of $g(f(x))$. The function $g \circ f$ is pronounced as “g of f”.

Inverse Functions

For a bijective function, $f : X \to Y$, there is an inverse function, $f^{-1} : Y \to X$, which is defined as $f^{-1}(y) = x$ if and only if $f(x) = y$. For example, if the function $g : A \to B$ gives $g(a) = 1$ and $g(b) = 2$, then the inverse function $g^{-1} : B \to A$ must give $g^{-1}(1) = a$ and $g^{-1}(2) = b$.

Arity

The arity of a function or operator is the number of members of the domain which are used to calculate the output value. Functions with an arity of 1 are known as unary (this includes functions like $f(x)$ as well as the unary minus (the $-$ from -1)), 2 are known as binary, 3 as ternary, etc. For example, the function $f : A \to B$ defined by $f(x, y) = x \times y$ is a binary function and therefore a arity of 2.

Lecture - Logic

17:00	13/02/24	Janka Chlebikova

Propositions

A proposition is a statement that is either true or false, but not both. The letters p, q, r, s, etc. denote propositional variables, each of which has one of two truth values, true or false. Statements can be combined using logical connectives to create compound statements.

Logical Connectives

The three main logical connectives are not ($\neg$), and ($\wedge$), or ($\vee$). These work in the same way as they do in other places, such as boolean algebra, but use different and objectively the correct symbols.

Not (Negation)

The negation of a statement p is the statement not p, or $\neg p$

And (Conjunction)

The conjunction of two statements, p and q, is the compound statement p and q, or $p \wedge q$.

Or (Disjunction)

The (inclusive) disjunction of two statements, p and q, is the compound statement p or q, or $p \vee q$. There is also an exclusive disjunction, which means that only one of the two statements can be true, and so is exclusive.

Conditional Propositions

Implication

An implication compound proposition ($\Rightarrow$) means an 'if-then' relation, e.g. if it is raining (p), then I get wet (q). $p \Rightarrow q$. Since it is an implication, q can be true regardless of the value of p, but if p is true then q must also be true. This can be represented using the following truth table:

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

As suggested, if p is false, then $p \Rightarrow q$ is true, since there is no implication as to the value of q

Bi-conditional

A bi-conditional compound proposition ($\Leftrightarrow$) means an 'if and only if' relation, and as such q can be true if and only if p is true. This can be represented using the following truth table:

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

Negation of Conditional Propositions

The negation of a conditional proposition, $p \Rightarrow q$ would be $p \wedge \neg q$ since

$$\neg(p \Rightarrow q) \equiv \neg(\neg p \vee q) \equiv \neg(\neg p) \wedge \neg q \equiv p \wedge \neg q$$

Contrapositives

The contrapositive of a conditional proposition, $p \Rightarrow q$ would be $\neg p \Rightarrow \neg q$.

Truth of Compound Propositions

It is possible to construct truth tables for more complex compound statements, as we did for boolean algebra. The order of precedence for connectives is as follows:

- Brackets
- Not ($\neg$)
- And ($\wedge$)
- Or ($\vee$)
- Implication ($\Rightarrow$)
- Bi-conditional ($\Leftrightarrow$)

For multiple of the same precedence, you can calculate them in any order (left to right or right to left).

Statement Properties

Tautology

A statement is a tautology if it is true for all possible values of it's propositional variables, e.g. $p \vee \neg q$ is a tautology since it is always true, as demonstrated by the below truth table:

p	$\neg p$	$p \vee \neg p$
T	F	T
F	T	T

Contradiction

A statement is a contradiction if it is false for all possible values of it's propositional variables, e.g. $p \wedge p$ is a contradiction since it is always false, as demonstrated by the below truth table:

p	$\neg p$	$p \wedge \neg p$
T	F	F
F	T	F

Contingency

A statement is a contingency if it can be either true or false depending upon the values of the propositional variables.

Logical Equivalence

Two or more propositions with the same truth values are said to be logically equivalent, since either one will produce the same output given the same propositional variables. For example, $p \Rightarrow q \equiv \neg p \vee q$ since

p	q	$p \Rightarrow q$	$\neg p$	$\neg p \vee q$
T	T	T	F	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

Like boolean algebra, propositions follow a few basic rules,

- Commutative - $p \wedge q \equiv q \wedge p$, $p \vee q \equiv q \vee p$
- Distributive - $(p \vee (q \wedge r)) \equiv ((p \vee q) \wedge (p \vee r))$
- De Morgan's - $\neg(p \vee q) \equiv \neg p \wedge \neg q$, $\neg(p \wedge q) \equiv \neg p \vee \neg q$

You can determine if two propositions are logically equivalent in one of two ways, either by using truth tables or by using basic logical equivalences (the above rules) to simplify the propositions and make them easier to discuss. Using truth tables can be complex and cumbersome since if there are more than two propositional variables, there will be many rows in each table.

Necessary and Sufficient Conditions

The sufficient condition of an 'if-then' statement is r, for s where 'if r then s'

The necessary condition of an 'if-then' statement is r, for s where 'if s then r' or 'if not r then not s'

Part II

Functional Programming

Lecture - Intro to Functional Programming

12:00 22/01/24 Matthew Poole

- For this module, we will be using the GHC (Glasgow Haskell Compiler), or more specifically it's interactive shell, GHCi

Imperative VS Functional Programming

- Most programming languages are imperative
  - Such as Python, JavaScript, C, etc
- Functional programming is another programming paradigm, which is based upon the mathematical concept of a function
- Imperative programming has state, statements (or commands) and side effects
- Pure Functional programming has no state, statements, or side effects
- A side effect is the change of state caused by calling a functional assigning a variable, etc
  - This means that it is not always possible to predict the result of running a program, even with access to it's source code
- Since most programs need to cause a side effect (usually outputting data), most functional programming languages are not purely functional, but tend to organize the code such that only one part causes side effects

Functional Programming Languages

- There are two types of functional programming languages
- Pure
  - Languages such as Haskell
  - Has absolutely no state or side effects
- Impure
  - Languages such as ML, Clojure, Lisp, Scheme, OCaml, F#
  - Has some state or side effects, either everywhere or in a specific part of code
- There are also some functional constructs in major imperative languages such as Python, JavaScript, and more

FP Basics

Expressions

- An expression is a piece of text which has a value
- To get the value from the expression, you evaluate it
- This gives you the value of the expression
- e.g.
- ```
  Expression -> evaluate -> Value
  2 * 3 + 1  -------------> 7
  ```

Functions

- A function whose output relies only upon the values that are input into it
- The result will always be the same, given the same values
- This is the same as a mathematical function, which is where the name Functional Programming comes from

Haskell Basics

- In Haskell, all functions have higher precedence than operators
- This means that you have to explicitly use brackets to ensure the correct order of operations
  ```

Lecture - Intro to Functional Programming II

12:00 22/01/24 Matthew Poole

Tracing a Functional Program

In an imperative program, it is obvious that you trace the program by determining the effect of each statement on the overall state of the program. However, with a functional program, each step of the tracing process is evaluating an expression, known as calculation. For example, we can trace the following program

```
twiceSum :: Int -> Int -> Int
twiceSum x y = 2 * (x + y)

twiceSum 4 (2 + 6)
```

by replacing each of the parameters of `twiceSum` as below

```
twiceSum 4 (2 + 6)
~> 2 * (4 + (2 + 6))
~> 2 * (4 + 8)
~> 2 * 12
~> 24
```

As above, in Haskell, the arguments are passed into the function verbatim, so the first step of executing a function is usually evaluating the arguments. This means that Haskell uses Non-Strict Computation - The arguments are passed into the function before being evaluated. Some functional programming languages use Strict Computation, meaning that the arguments are evaluated before being passed into the function. It doesn't really make a difference, other than that you may be asked to use a specific method in questions on exams, etc.

Guards

Guards are boolean expressions which can be used when defining a function to give different results, depending upon the input or a property thereof. This is especially useful for **Guarding** against invalid inputs. The syntax in Haskell is as below

```
maxVal :: Int -> Int -> Int
maxVal x y
  | x >= y    = x
  | otherwise = y
```

If the first guard is true, then the corresponding result is returned. If it's false, the next guard is evaluated, and corresponding value returned. You can also create a "default" case which is used if none of the other guards are true, which uses the keyword `otherwise`. Guards can also be used instead of a chain of if statements, which is easier to understand, and simpler to create in the first place.

Local Definitions

If your function uses a very complex mathematical calculation, you may want to break the calculation into several steps. In Haskell, you can do this using Local Definitions. Using the `where` keyword, you can define what are effectively local variables. This is useful as you can break down complex calculations into multiple, less complex and easier to understand ones. For example, the following function,

```
distance :: Float -> Float -> Float -> Float -> Float
distance x1 y1 x2 y2 = sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

could also be written as

```
distance x1 y1 x2 y2 = sqrt (dxSq + dySq)
  where
  dxSq = dx^2
  dySq = dy^2
  dx = x1 - x2
  dy = y1 - y2
```

The order of local definitions is irrelevant, and will still work if you reference a definition before it is actually defined. Local definitions are only usable within the function they are defined, hence “local”, but are able to reference each other, and the parameters of the function. As well as “variables”, you can also define “functions” as local definitions. This is useful to simplify repetitive code, which is used only within the function itself. Local definitions can also be used in conjunction with guards, in which case they are defined after the guards.

Lecture - Pattern Matching and Recursion

12:00 05/02/24 Matthew Poole

Importing Libraries

As with any other language, you can import files in Haskell. There are some standard libraries included in GHC, such as the `Data.Char` library, which includes functions for manipulating strings, such as `toUpper` and `toLower`. (Astounding features, I know). To import an entire module, you use `import Data.Char`, but to import only specific functions you can use `import Data.Char (toUpper, toLower)`. There is also a "standard prelude" which is imported automatically by the interpreter. It includes the definitions of standard functions, such as `mod` as well as commonly used types.

Haskell includes functions, which are used as with prefix notation, e.g. `mod n 2` and operators which are used with infix notation, e.g. `2 - 1`. There is an operator which uses prefix notation, the unary minus which is used to represent a negative number. You can use any binary function (one with two arguments) as an infix operator, by surrounding it with backticks, e.g. `mod n 2` could also be written as `n 'mod' 2`. You can also use an operator as a function by surrounding it with brackets, e.g. `1 + x` could also be written as `(+) 1 x`.

Pattern Matching

There are two ways of defining functions - using single equations and using guards - which have already been covered, but there is another way, which is using pattern matching. Patterns work in a similar way to guards, and one example is the function `not` which is defined in the prelude as

```
not :: Bool -> Bool
not True    = False
not False   = True
```

This definition is a sequence of equations. For each pattern (on the left) there is a result (on the right). When the function is called, the input is checked against each pattern, and if it matches, that pattern's output is returned.

You can also use the wildcard pattern `_`, which matches any value. This is often useful for simplifying complex patterns. For example, if we wanted to redefine the boolean or operator, `||`, we could define it as

```
(||) :: Bool -> Bool -> Bool
True || True     = True
True || False    = True
False || True    = True
False || False   = False
```

but this is very complex, and defines 3 redundant patterns. We could instead use the wildcard, and define it as

```
(||) :: Bool -> Bool -> Bool
False || False   = False
_ || _           = True
```

I would argue this is not necessarily more readable, but it is more compact and technically more efficient

Recursion

For and while loops are very much imperative constructs, as they operate on the state of the program. This means that they cannot exist in pure functional programming. Therefore recursion is a fundamental concept in the functional paradigm. Recursion is used heavily throughout functional programming, but especially when a list or other iterable data type is involved.

One common example of iteration is to calculate the factorial of a number. Since the factorial of a number, n is defined as $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$ and by convention, $n! = 1$, we can define the factorial of n where $n > 0$ in terms of the factorial of $n - 1$. e.g. $3! = 3 \times 2!$. This gives us a very simple recursive algorithm, which could be defined as follows in Haskell

```
fact :: Int -> Int
fact n
  | n > 0   = n * fact (n - 1)
  | n == 0  = 1
```

This definition, despite being correct, will fail for negative integers as there is no guard for that case. To fix this, you could add the following `otherwise` guard to give an error message

```
  | otherwise = error "Undefined for negative integers"
```

General Recursion

The previous example of a recursive function was, in fact, a primitive recursive function, e.g. the base case considers the value of 0 and the recursive case considers how to get from $n - 1$ to n. Another example of a primitive recursive function is to perform multiplication with addition, e.g.

```
mult :: Int -> Int -> Int
mult n m
  | n == 0  = 0
  | n > 0   = m + mult (n - 1) m
```

Since this function also has a base case of `n == 0`, it is primitive. A general recursive function is one in which the base case is not checking for a value of 0. For example, if we were to implement integer division using subtraction, the base case would be where the divisor is greater than the dividend. e.g.

```
divide :: Int -> Int -> Int
divide n m
  | n < m       = 0
  | otherwise   = 1 + divide (n - m) m
```

Lecture - Tuples, Lists and Strings

12:00 12/02/2024 Matthew Poole

Characters and Strings

Haskell includes both `Char` and `String` data types. The `Data.Char` module includes some useful functions for working with characters, such as `toUpper` and `toLower`, which do as they suggest. There is also the function `isDigit`, which is useful for checking if a string or character can be parsed into a number. As with most programming languages, Strings are defined as a list of characters.

Tuples

Tuples are used to combine several pieces of data into a single value which can be passed between functions more easily. For example, if we wanted a function to return someone's name, and the score they got in a test, it could return that using a tuple on the form `("Thomas", 68)`. The specific type of that tuple is `(String, Int)`, a singular data type. This allows you to use it as an input or return type for any function, for example, the following function takes two student's scores and returns the name of whichever student got the higher score

```
betterScore :: (String, Int) -> (String, Int) -> String
betterScore (name1, score1) (name2, score2)
  | score1 >= score2  = name1
  | otherwise         = name2
```

When using tuples or other composite data types in Haskell, it is a good idea to define a type synonym, such as `:type StudentMark = (String, Int)`. This can then be used in the code rather than writing the full type definition each time, for example, `betterScore` can be re-written as follows

```
betterScore :: StudenMark -> StudentMark -> String
betterScore (name1, score1) (name2, score2)
  | score1 >= score2  = name1
  | otherwise         = name2
```

Polymorphic Functions

A polymorphic function is one which has multiple definitions with different input types. For example, the length function defined in the prelude works on any type of list, and always returns the number of items in the list. The length function actually uses a type variable, which can take an arbitrary type which can be referenced in the function. The type definition of length is `length :: [a] -> Int`, which is known as the function's **most general type**. You can define a polymorphic function in several ways, but if you don't include a type definition, and just defined the function itself, e.g. `square n = n * n`, then Haskell attempts to infer the most general type by analysing the structure of the function. In this case, it is inferred as `square :: Num a => a -> a`, which means that a can be any numeric data type.

Lists

Lists are used to store any number of values of the same type, as in any other language. In Haskell, they are the main data structure, and are used extensively in actual programs. Lists are defined as in most other languages, e.g. `[1, 2, 3, 4, 5]`. The data type of a list can be defined in a function definition by surrounding the data type with square brackets, e.g. `stringList :: [String]`. The empty list (`[]`) can be of any data type. Strings in Haskell, as with most languages, are defined as a list of characters, quite literally defined as `:type String = [Char]` in the prelude. This means that any operation working on lists will also work on strings, such as concatenating.

When creating a list, you can also use a range format to populate a list, for example `[1 .. 5]` is the same as writing out `[1, 2, 3, 4, 5]`. This also works with floats and characters and therefore strings, as `['a' .. 'z']` gives a string containing the entire alphabet.

List Comprehension

A list comprehension is effectively a method of mapping one list onto another. For example, if we have the list `a = [1, 2, 3, 4, 5]`, then the comprehension `[ 2*i | i <- a ]` would give the value `[2, 4, 6, 8, 10]`. The data type of the output list does not have to be the same as the input list, which allows you to check through a list and return a list of boolean values all at once.

You can also add a test at the end of the generator (`i <- a`) which will only add a value to the output list if the input value passes the test. For example, if we modified the previous comprehension to be `[ 2*i | i <- a, i < 5]` it would only return the values `[2, 4, 6]`, as only the input values `1, 2 & 3` pass the test.

Rather than a single variable, you can use a pattern on the left side of the `<-` to extract multiple values. For example, if we have a list of tuples and wish to add the two values together, we could use the comprehension `[ i+j | (i,j) <- b ]`.

You can also use comprehensions within functions. If you wanted to define a function to do that pair addition, you can write it as follows

```
addPairs :: [(Int, Int)] -> [Int]
addPairs pairs = [ i+j | (i,j) <- pairs ]
```

List Functions

Every list function is polymorphic, and can be used on any type of list, as long as both input lists are of the same type. More specifically, they all have the same type definition of `[a] -> [a] -> [a]`

The list function `:` is probably one of the most used list functions, and adds an element to the front of a list, e.g. `3:[5, 7, 2]` returns `[3, 5, 7, 2]`.

The `++` operator joins two lists together, e.g. `[1, 2, 3] ++ [4, 5, 6]` returns `[1, 2, 3, 4, 5, 6]`. Since strings are lists of characters, this is also how you concatenate strings together, e.g. `"hello " ++ "there"` returns `"hello there"`.

The `!!` operator returns the element at a given position in a list, e.g. `["one", "two", "three"] !! 2` returns '"three"', since Haskell uses 0-indexed lists. This is not used very often in Haskell, but is still useful to know just in case it's ever needed.

Finally, the `null` function checks if a list is empty, e.g. `null [1, 2]` returns `False` and `null []` returns `True`