University of Portsmouth
BSc (Hons) Computer Science
Second Year

Discrete Mathematics and Functional Programming (`DMAFP`)

M21274
January 2024 - June 2024
20 Credits

Thomas Boxall
`up2108121@myport.ac.uk`

Contents

Part I

Discrete Maths

Page 1

Lecture - Sets

📅 2024-01-23 🕔 17:00 🎓 Janka

1.1 Introduction

Sets underpin maths and Computer Science. A set is a collection of objects, which are called the elements (also known as members of the set). For example, a set of the numbers 1, 3, 8; or the collection of students in a class born in March. There are two characteristics of sets:

1. There are no repeated occurrences of elements
2. There is no particular order of the elements

1.2 Set Notation

The elements of a set are enclosed in braces with their names being denoted by a *letter*, for example:

$$A = \{1, 2, 3\}, \quad C = \{Portsmouth, Brighton, London\}$$

There are two ways that we can describe the members of a set. We can *list the elements* which is mainly used for finite sets, for example:

$$A = \{3, 6, 9, 12\}$$

We can *specify a property* that all the elements in the set have in common. The '|' character is read 'such that', sometimes ':' is used in it's place. For example:

$$B = \{x | x \text{ is a multiple of 3 and } 0 < x < 15\}$$

We can also use *three dots* to informally denote a sequence of elements that we don't wish to write down, for example:

$$C = \{1, \ldots, 10\}$$

1.2.1 Sets of Numbers

There are some reserved letters to denote specific sets of numbers in maths. These are shown below:

- $\mathbb{N}$ (or N) is used for the set of natural numbers (integers >= 0). $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$
- $\mathbb{Z}$ (or Z) is used for the set of integers. $\mathbb{N} = \{\ldots, -1, 0, 1, \ldots\}$
- $\mathbb{Q}$ (or Q) is used for the set of rational numbers (number which can be expressed as a quotient or fraction). $\mathbb{Q} = \{0, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots\}$
- $\mathbb{R}$ (or R) is used for the set of real numbers. $\mathbb{R} = \{\ldots, -1, 0, \frac{1}{2}, \ldots\}$

1.2.2 Elements of a Set

We can use the $\in$ symbol to denote if an element is a member of a given set. For example, if x is a member of S - then we can say:

$$x \in S$$

The symbol $\notin$ denotes an element is not a member of a given set. For example, if y is **not** a member of S - then we can say:

$$y \notin S$$

1.2.3 Many Ways to Say The Same Thing

There are several ways of describing the same set, for example for the set S of *odd integers*:

$$\begin{aligned} S &= \{\ldots, -5, -3, -1, 1, 3, 5, \ldots\} \\ &= \{x | x \text{ is an odd integer } \} \\ &= \{x | x = 2k + 1 \text{ for some integer } k\} \\ &= \{x | x = 2k + 1 \text{ for some } k \in \mathbb{Z}\} \\ &= \{2k + 1 | k \in \mathbb{Z}\} \end{aligned}$$

The phrase "for some [integers K]", means "for all [integers k]"

1.2.4 Empty Sets

Where a set has *no elements*, it is called an empty set or null set. It's denoted with the $\emptyset$ symbol, for example:

$$\emptyset = \{\}$$

1.2.5 Finite & Infinite Sets

If the number of elements in the set is fixed (for example when counting the elements at a fixed rate for a set amount of time), then the set is *finite*. If the set X is finite, then we call $|X|$ the *cardinality* of X therefore:

$$|X| = \text{number of elements in } X$$

If the counting never stops then X is an infinite set.

1.2.6 Subsets

A subset is where one set's elements are entirely present in another set. There are three conditions we need to know about:

- $A \subseteq B$: A is a subset of B therefore every element in A is also in B.
- $A \nsubseteq B$: A is not a subset of B.
- $A \subset B$: A is a proper subset of B, therefore B has at least one additional element which is not in A.

1.2.7 Equality of Sets

Two sets are *equal* if they have exactly the same elements. This is denoted by writing $A = B$. Where $A = B$, the following conditions are also true:

- $A \subseteq B$ for every a if $a \in A$, then $a \in B$
- $B \subseteq A$ for every b if $b \in B$, then $b \in A$

1.3 Operations on Sets

Sets can have *operations* performed on them - this will change something about them.

1.3.1 Intersection

The intersection of two sets A and B is defined as:

$$A \cap B = \{x | x \in A \text{ and } x \in B\}$$

This is the set of elements which appear in both sets only. If we take a Venn Diagram with a set on either side - its the overlapped elements which would be returned from an intersection operation. For example if $A = \{a, b, c\}$ and $B = \{c, d\}$ then $A \cap B = \{c\}$.

1.3.2 Disjoint

If an intersection returns no elements, then the two sets are *disjoint*. This is shown by:

$$A \cap B = \emptyset$$

1.3.3 Union

The *union* of the two sets A and B is defined as:

$$A \cup B = \{x | x \in A \text{ or } x \in B\}$$

This is the set of elements which are in either A or B, this means elements which appear in both are returned. For example, if $A = \{a, b, c\}$ and $B = \{c, d\}$ then $A \cup B = \{a, b, c, d\}$.

1.3.4 Difference

The *difference* between two sets, A and B is defined as:

$$A \setminus B = \{x | x \in A \text{ or } x \in B\}$$

This is the set of elements which are in A but not in B, so could be represented as $A - B$. Note that $A \setminus B \neq B \setminus A$.

1.3.5 Counting Elements In a Set

If we take A and B to be finite sets, we can calculate the number of elements in the union of A and B. The correct way to count this is as follows:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

We have to minus $|A \cap B|$ from the sum because otherwise it is as though we are counting it twice due to the fact that we are summing the total number of elements in A and B.

1.4 Complement

If we consider that all subsets are the subset of a particular set, U for example (the universe of discourse), then the difference $U \setminus A$ is called the *complement* of A is shown as either $\overline{A}$ or A'. For example:

$$A' = \{X | x \in U \text{ and } x \notin A\}$$

1.5 Basic Set Properties

Sets have a number of basic properties - many of these are the same as that for Boolean Expressions

- $A \cup \emptyset = A$
- $A \cap \emptyset = \emptyset$
- $A \cup A = A$
- $A \cap A = A$
- Commutative
  - $A \cup B = B \cup A$
  - $A \cap B = B \cap A$
- Associative
  - $(A \cup B) \cup C = A \cup (B \cup C)$
  - $(A \cap B) \cap C = A \cap (B \cap C)$
- Distributive
  - $A \cap (B \cap C) = (A \cap B) \cup (A \cap C)$
  - $A \cup (B \cup C) = (A \cup B) \cap (A \cup C)$
- de Morgan's
  - $(A \cap B)' = A' \cup B'$
  - $(A \cup B)' = A' \cap B'$

1.6 Power Set

A *power set* is the collection of all subsets of a set, S which is denoted by $P(S)$. For example, if $S = \{a, b, c\}$ then:

$$P(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$$

1.7 Partition

A *partition* of the set S is a collection of non-empty subsets of set S where every element form S belongs to exactly one member of S. This means that the sets are mutually disjoint and that the union of all the sets in the collection results in the original set, S. For example, if $S = \{a, b, c, d, e, f\}$ then $\{\{a, e\}, \{c\}, \{f, d\}, \{b\}\}$ is a partition of S.

Page 2

Lecture - Relations

📅 2024-01-30 🕘 17:00 🎓 Janka

2.1 Ordered Pairs

An ordered pair of elements is a group of two elements which are in a specific order. They are written as (a, b) and the order matters - this means (a, b) is distinct from the pair (b, a). Note that ordered pairs use the brackets () while sets use curly braces {}

2.2 Cartesian Product

The Cartesian Product of two sets is the set of **all** ordered pairs where the first element is taken from set 1 and the second element from set 2. The formal definition is as follows:

$$A \times B = \{(a, b) | a \in A \text{ and } b \in B\}$$

For example - if $X = \{1, 2, 3\}$ and $Y = \{a, b\}$, then

$$X \times Y = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$$

2.3 Relations

A relation is the set of subsets from a cartesian product. For example, if we take $A = \{a, b, c, d, e\}$ and $B = 1, 2, 3$ then:

$$R_1 = \{(a, 1), (b, 1), (c, 2), (c, 3)\}$$
$$R_2 = \{(a, 3), (a, 1), (c, 2), (c, 1), (b, 2)\}$$

R_1 and R_2 are both examples of Binary relations from A to B.

2.3.1 Describing Relations

To describe a relation, we could list all of its elements, however this can be very long and obtuse so it's better & more common practice to use "the characteristics of their elements".

2.3.2 Relations On A Set

A relation *on a set* is where both sets are equal. For example $A = B$ then a relation on A is a relation from A to A, hence a subset of $A \times A$.

For example, let R be the relation on $A = \{1, 2, 3, 4\}$ as defined by:

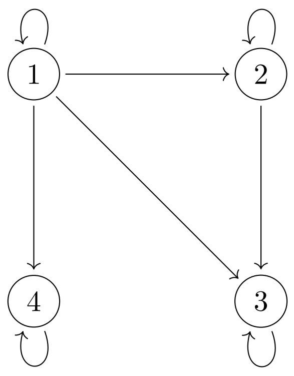
$$(x, y) \in R \text{ if and only if } x \text{ divides } y, \text{ for all } x, y \in A$$

Then we can conclude that R is:

$$R = \{(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 3), (4, 4)\}$$

2.4 A Digraph

We can use a *digraph* to picture a relation on a set. An example is shown below:



The dots (vertices) represents the elements of $A = \{1, 2, 3, 4\}$. If the element (x, y) is in the relation, an arrow (directed edge) is drawn from x to y.

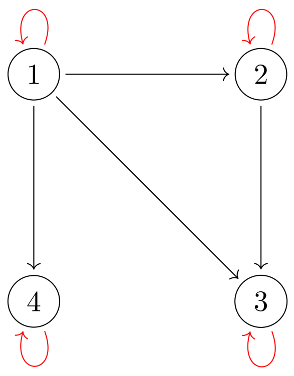
$$R = \{(1,1),(1,2),(1,3),(1,4),(2,2),(2,4),(3,3),(4,4)\}$$

2.4.1 Reflexivity

A relation is reflexive where for all elements in the set - there is an ordered pair in which both elements are the same, for example $(1, 1)$ or $(2, 2)$. In reference to the set $A = \{1, 2, 3, 4\}$, the relation:

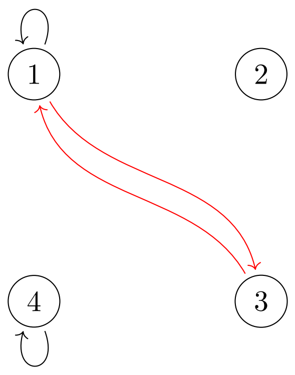
$$R = (1,1),(1,2),(1,3),(1,4),(2,2),(2,4),(3,3),(4,4)$$

On the following digraph, the red arrows are the ones which display the reflexivity.



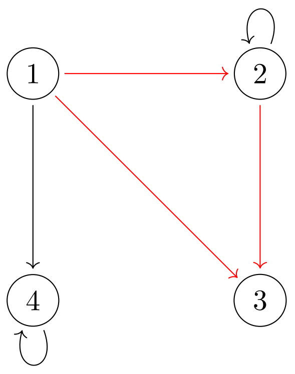
2.4.2 Symmetry

A relation is symmetrical where $(x, y) \in R$ and $(y, x) \in R$. If the condition is not true, then we do not have symmetry.



2.4.3 Transitivity

For a binary relation, R on set A; R is transitive if and only if for all $x, y, z \in A$ if $(x, y) \in R$ and $(y, z) \in R$ and $(x, z) \in R$. We initially assume that a relation is transitive and try to disprove it; if we are unable to disprove it then the relation is transitive. In the event that there is only one element in the relation - the relation will *always* be transitive.



2.4.4 Equivalence

Where a relation is reflexive, symmetric and transitive - it is classed as an equivalence.

2.4.4.1 Equivalence Class

To be continued.

Page 3

Lecture - Functions

2024-02-06 17:00 Janka

A *function* can be described in two ways. The mathematical definition is that “a function is a special type of relation in which a single input will have at most one output”. The alternative definition of a function is that it is a mysterious black box which takes an input and returns and output. The same function, with the same input will always return the same output.

If we take A and B to be nonempty sets, then: A is a (total) function f from A to B, $f : A \to B$, is a relation from A to B such that

$$\text{for all } x \in A \text{ there is exactly on element in } B, f(x)$$

associated with x by relation f. Note that the word ‘total’ is used to describe the above function which means that every input has a defined output. The function $f : \mathbb{Z} \to \mathbb{Z}$ which is defined by $f(x) = 2x$ is also an example of a total function.

It is also possible to have a ‘partial’ function, this is where some of the inputs do not have defined outputs. For example the function $f(\frac{1}{x})$ where $x = 0$, would be undefined therefore the function is classed as ‘partial’.

3.1 Describing A Function

There are a few different ways in which a function can be described.

3.1.1 By A Formula

This is the most common method used. The function f from $\mathbb{N} \to \mathbb{N}$ that maps every natural number x to its cube x^3 can be described as:

$$f(x) = x^3$$

3.1.2 By All Possible Associations

Whilst this is a valid method, it will generally not be used for efficiency reasons. The function g from $A = \{a, b, c\}$ to $B = \{1, 2, 3\}$ would be shown as:

$$g(a) = 1, g(b) = 1, g(c) = 2$$

3.2 Domain, Co-Domain & Range

The domain of a function is the set of all input values for which there is a defined output. For example if we let $f : A \to B$ then the subset $D \subset A$ of all elements for which f is defined is the domain. In the case of a total function, $D = A$ and in the case of a partial function, $D \subseteq A$

The co-domain of a function is the set of all possible output values; not just the ones which map to an input. For example, if we let $f : A \to B$ then the set B is the co-domain.

The range (also sometimes known as the image) of a function is the set of elements in the co-domain which map to an input. For example, if we let $f : A \to B$ then the range is denoted by $range(f)$. The range can also be expressed as:

$$range(f) = \{f(x)|x \in A\}$$

3.3 Properties of Functions

Functions have a number of properties.

3.3.1 Injective

The function $f : A \to B$ is injective (or one-to-one) if there is only one input that maps to each output. It can mathematically be defined as:

$$\text{for all } x, y \in A \text{ if } x \neq y \Rightarrow f(x) \neq\Rightarrow f(y)$$

3.3.2 Surjective

A function $f : A \to B$ is surjective (or onto) if the $range(f)$ is the co-domain B. It can mathematically be defined as

$$\text{for all } y \in B \text{ there exists } x \in A \text{ such that } f(x) = y$$

A function which is not *onto* is *into*.

3.3.3 Bijective

A function $f : A \to B$ is bijective (or one-to-one correspondence) if it is both injective and surjective.

3.4 Composite Functions

A new function can be constructed by combining other simpler functions in some way. If we let $f : A \to B$ and $gB \to C$ be functions. The composition of g with f is the function denoted by $g \circ f : A \to C$ and defined by:

$$(g \circ f)(x) = g(f(x)) \text{ for all } x \in A$$

$(g \circ f)(x) = g(f(x))$ is read as g of f, which means do f first then do g.

3.5 Inverse Functions

An inverse function is where the output of function f can be fed into the input of function f^{-1} to get the original input of f. This is mathematically defined as: $f : X \to Y$ is a bijective function, then there is an inverse function $f^{-1} : X \to Y$ that is defined as:

$$f^{-1}(y) = x \text{ if and only if } f(x) = y$$

Page 4

Lecture - Logic I

📅 2024-02-13 ⏱ 1700 🎓 Janka

4.1 Reasoning

Reasoning is something that we are introduced to doing from a young age. Younger children will continually ask "why?" as they attempt to make sense of the world while growing up; as they grow older they will generally only want the facts however. *Logic* is the discipline which deals with the method of reasoning:

- in mathematics to prove theorems
- in computer science to verify the correctness of programs and to prove some theorems
- in the natural and physical sciences to draw conclusions from experiments
- and in our everyday lives to solve a multitude of problems!

Over time, people come to understand that the following statement is how logic & reasoning works:

"If X then Y" is true and "X" is true $\Rightarrow$ so "Y" must be true

4.2 Propositions

A *proposition* it a *statement* (which is a declarative sentence) that can either be true or false; however not both. A proposition will be exact, not wishy-washy. For example - $3 - x = 5$ is not a proposition as it has an unknown value of x; however "The earth is flat" is a proposition as it can, and only can, categorically be True or False.

4.2.1 Propositional Variables

Propositions can be quite long. Mathematicians like efficiency, therefore they apply a single letter variable to a propositional statement to make their lives easier. The letters $p, q, r, \ldots$ are used to denote propositional variables.

Statement can be combined with logical connectives to obtain compound statements. For example *p and q*

4.3 Logical Connectives

Logical Connectives are used to combine propositional statements together; they are very similar to Boolean Algebra[1]. The truth value of a compound statement depends only on:

[1]Covered in A-Level Electronics, A-Level Computer Science, 1st Year ArchOS Module

- the truth values of the statements being combined
- the types of connectives being used

4.3.1 Negation (not)

If p is a statement, the negation of p is the statement *not* p, denoted by $\neg p$. The truth table of negation is shown below:

p	$\neg p$
T	F
F	T

4.3.2 Conjunction (and)

If p and q are statement, the conjunction of p and q is the compound statement p *and* q, as denoted by $p \wedge q$

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

4.3.3 Disjunction (or)

If p and q are statements, the (inclusive) disjunction of p and q is the compound statement p *or* q as $p \vee q$

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

4.3.4 Conditional Proposition (implication)

If p and q are statements then the compound statement "*if p then q*" dented $p \to q$ (or $p \Rightarrow q$) is called implication. The hypotheses in the statement is p and the conclusion is denoted by q.

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

4.3.5 Conditional Proposition (bidirectional)

If p and q are statement, the compound statement “if and only if” (abbreviated to *iff*), denoted by $p \Leftrightarrow q$, is called the biconditional of p and q.

p	q	$p \Leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

4.4 Truth of Compound Properties

It is possible to combine the truth tables for single logical connectives to make *more complicated truth tables.* In similar fashion to Algebraic & standard numeric expression evaluation - there is a hierarchy of evaluation, as seen below (listed highest to lowest):

1. brackets
2. negation ($\neg$)
3. conjunction ($\wedge$)
4. disjunction ($\vee$)
5. implication ($\rightarrow$)
6. bidirectional ($\leftrightarrow$)

For connectives of equal priority - work from left-to-right through them.

4.5 Special Conditions

4.5.1 Tautology

A statement that is true for all possible values of its propositional variables is called a tautology. For example $p : p \vee \neg p$ is a tautology.

4.5.2 Contradiction

A statement that is false for all possible values of its propositional variables is called a contradiction. For example, any proposition $p : p \wedge \neg p$ is a contradiction. In a truth table, the final column (where the output is) will always be false.

4.5.3 Contingency

A statement that can be either true or false depending on the truth values of its propositional variables is called a contingency. For example the proposition "Murray will win the Wimbledon next year" us a contingency because the truth of the statement is dependent on the propositional variable.

4.5.4 Contrapositive

The contrapositive of a conditional statement $p \to q$ is $\neg q \to \neg p$. The conditional statement is logically equivalent to its contrapositive.

4.6 Logical Equivalence

Two statements are said to be *logically equivalent*, $\equiv$, if and only if they have identical truth values for each possible value of their statement variables. Logical equivalence corresponds to $=$ with numbers.

Shown below are the rules of Logical Equivalence. There is not a requirement to memorise these as it should be possible to derive them in the exam when required. Note that they are the same as Boolean algebra's laws, except with funky symbols.

- $p \land p \equiv p$
- $p \lor p \equiv p$
- $p \land T \equiv p$
- $p \land F \equiv F$
- $p \lor T \equiv T$
- $p \lor F \equiv p$
- $\neg(\neg p) \equiv p$
- $p \lor (\neg p) \equiv T$
- $p \land (\neg p) \equiv F$
- $p \land q \equiv q \land p$ (commutativity)
- $p \lor q \equiv q \lor p$ (commutativity)
- $(p \lor (q \land r)) \equiv ((p \lor q) \land (p \lor r))$ (distributivity)
- $(p \land (q \lor r)) \equiv ((p \land q) \lor (p \land r))$ (distributivity)
- $p \to q \equiv (\neg p \lor q)$
- $\neg(p \lor q) \equiv ((\neg p) \land (\neg q))$ (De Morgan's Law)
- $\neg(p \land q) \equiv ((\neg p) \lor (\neg q))$ (De Morgan's Law)

4.6.1 Example of logical equivalence proof

Prove that $(p \to q) \lor (p \to r) \equiv p \to (p \lor r)$

$$\begin{aligned}
(p \to q) \lor (p \to r) \equiv & \\
& \equiv (\neg p \lor q) \lor (\neg p \lor r) && \text{logical equivalence law} \\
& \equiv \neg p \lor q \lor \neg p \lor r && \text{all are `or' so remove brackets} \\
& \equiv \neg p \lor q \lor r && \text{get rid of second} \neg p \\
& \equiv \neg p \lor (q \lor r) && \text{add brackets in} \\
& \equiv p \to (q \lor r) && \text{logical equivalence law again}
\end{aligned}$$

4.7 Necessary and Sufficient Condition

A necessary condition is a condition such that statement B cannot be true without statement A being true. However it is possible for statement A to be true to even if statement B is not true.

A sufficient condition is a condition such that knowing statement A is true guarantees that statement B is true.

A statement (A) is said to be "necessary and sufficient" for the statement B when B is true if and only if A is also true.

Part II

Functional Programming

Page 5

Lecture - Introduction to Functional Programming

2024-01-22 1200 Matthew

5.1 Introduction

Functional Programming is a different programming paradigm. There are all sorts of different ways we can classify a programming language, paradigm being one of them. More details on this in another module.

5.1.1 Imperative vs Functional Programming

Before we go too deep into Functional Programming, we will first look at the structure of Imperative Programming.

Imperative Programming is a paradigm where the execution of the program consists of executions of *statements*, which each impact the program's *state*. *Side Effects* can be caused by the statements; these are things that the program does where it cannot guarantee the outcome - for example get the current temperature, ask the user to enter a number or getting the system time.

Pure Functional Programming does not have a state, does not have statements and does not have side effects. However - side effects are a "necessary evil" so they get brought back in isolated from the main program. Once side effects are introduced, our functional programming becomes impure.

5.2 Functional Programming

In Functional Programming, there are three key terms - expression, evaluation and value. An expression is some text which has a value, for example `2 * 3 + 1`; a value is the thing which the expression has, for example `7`; and evaluation is the process used to obtain a value from an expression.

We will start our FunProg journey looking at Mathematical Functions, which we can think of as a box which maps argument values to a result value. A Haskell program is mainly made up of Function definitions, for example

```
square :: Int -> Int
square n = n * n
```

The first line above starts with the function name, then lists the parameters (a single `Int`), then finally the result type. The second line is the actual function, starting with the name, then the names assigned to the parameters, then the value which gets returned. The double colon (`::`) is read as 'is type of'.

5.3 Data Types: A Brief Introduction

Haskell includes a number of basic data types which we can make use of.

5.3.1 Boolean

The `Bool` data type has the values `True` and `False`. As well as having the boolean operators `&&` (and); `||` (or); and `not` (not). These can be seen implemented in the following function which implements the *exclusive or* operator (which gives True when exactly one of its arguments is True):

```
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

5.3.2 Int & Float

Haskell includes a number of different numerical data types - we'll start of using the `Int` and `Float` types. `Int` is a fixed-space integer data type; and `Float` is a floating point data type. Operators for these data types include:

- arithmetic operators: `+`, `-` and `*`
- The `Float` data type includes floating point division `/`
- The `Int` has integer division and remainder functions `div` and `mod`
- relation operators: `==`, `/=` (not equals), `<`, `>`, `<=` and `>=`

The operators use the standard precedence rules (as experienced in other languages).

5.4 Conditional Expressions

Haskell includes a conditional expression which takes the form:

```
if condition then m else n
```

where `condition` is a boolean expression and `m` & `n` are expressions of the same type Where `condition` is true - the expression evaluates to `m`; and where `condition` is false - the expression evaluates to `n`.

In the next lecture, we'll see an alternative to conditional expressions.

Page 6

Lecture - Introduction To Functional Programming II

📅 2024-01-29 🕘 1200 🎓 Matthew

6.1 Evaluation and Calculation

In a similar way to that of tracing an imperative program to understand the effect of executing their statements on the program state; we can evaluate expressions of a functional program step-by-step to understand the operation of it. This is also known as calculation.

The process of calculation will be explained with the example function:

```
twiceSum x y = 2 * (x + y)
```

which we can evaluate using the example inputs `4` and `(2 + 6)`. The complete calculation of the expression is as follows:

```
twiceSum 4 (2 + 6)
⇝ 2 * (4 + (2 + 6))       def of twiceSum
⇝ 2 * (4 + 8)             arithmetic
⇝ 2 * 12                  arithmetic
⇝ 24                      arithmetic
```

6.2 Guards

Guards are Boolean expressions used in function definitions to give alternative results dependent on the parameter values. The following function gives the largest of the two `Int` values:

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

We saw the `if ... then ... else ...` construct last week, which could be used in place of guards. However guards are the preferential thing to use where there are more than one case - as they allow for this easier. Shown below is a function, `maxThree`, which returns the largest of three `Int` values passed in:

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
| x >= y && x >= z   = x
| y >= z             = y
| otherwise          = z
```

Calculations involving guards are represented slightly differently, note the `??` denoting when the function is inside the guard. Shown below is the calculation of `maxThree`:

```
maxThree 3 2 5
?? 3 >= 2 && 3 >= 5       first guard
?? ⇝ True && 3 >= 5       def of >=
?? ⇝ True && False        def of >=
?? ⇝ False                def of &&
?? 2 >= 5                 second guard
?? ⇝ False                def of >=
?? otherwise              third guard
?? ⇝ True                 def of otherwise
⇝ 5
```

6.3 Local Definitions

Single line definitions in Haskell can be slightly unwieldy to read, write and understand. For this reason - we may want to breakdown the definition's expression to make it easier to read. For example the function:

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

can be broken down to the following:

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

From the above example, we see that the main expression uses the local definitions and the local definitions use the function's parameters. The local definitions can only be used within the functions that they are defined within; they are "hidden" from the rest of the program. The local definitions can appear in any order within the `where`.

Page 7

Lecture - Pattern Matching & Recursion

2024-02-12 1200 Matthew

7.1 Modules

As with Python and other popular programming languages, Haskell supports modules (libraries) which provide pre-written and tested code to do certain things. As with other languages, when using a module in Haskell - we have to import it with the `import` command. The first line below shows importing the entire module and the second line shows importing only two functions:

```
import Data.Char
import Data.Char (toUpper, toLower)
```

Haskell's standard library which is auto-imported into every other module & the interpreter is called the *standard prelude*.

7.2 Functions & Operators

Haskell includes both functions and operators. Functions (`sqrt`, `mod`, etc) are used in prefix notation (i.e. `mod n 2`). Operators (`+`, `-`, `**`, etc) are used in infix notation (i.e. `1 + x`); apart from the unary minus operator which is a prefix operator.

It is possible to use any binary (two-argument) function as an operator by surrounding it with back-quotes (`` ` ``). For example ``n `mod` 2`` is equivalent to `mod n 2`. Similarly, we can use an operator as a function by encasing the operator in parentheses (`()`). For example `(+) 1 x` is equivalent to `1 + x`.

7.3 Pattern Matching

So far, we have seen two ways of defining functions:

- using single equations
- using guards

We will now add a third function definition mechanism to our options: pattern matching.

Pattern matching consists of a sequence of equations; each *pattern* (on the left hand side) is associated with a different result (on the right hand side). An example of this is seen below, defining the `not` function which is included in the Prelude.

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

It is also possible to use a wildcard to simplify pattern matching. This can be seen below where the Boolean *or* operator is redefined.

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

Alternatively, we can also use *named parameters* which can take a value from the pattern and use it in the output. An example of this is shown below

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

7.4 Recursion

As with other programming languages, Haskell supports recursion. A recursive definition is one that is defined in terms of itself.

Recursion is a critical component in Haskell as pure functional programming does not support loops as these are imperative constructs (as they operate on a program's state). We shall recuse lots especially when using lists.

7.4.1 Recursive Definition of Factorial

To illustrate recursion in Haskell, we will examine an example of the Factorial function. $fact(n)$ is the product of the integer n and all the integers below it $n-1$, $n-2$, ... For example:

$$fact(3) = 3 \times 2 \times 1 = 6$$

Note that the factorial of a number $n > 0$ can be defined in terms of the factorial of $n-1$, for example

$$fact(4) = 4 \times fact(3)$$

This leads us to the following recursive function definition

```
fact :: Int -> Int
fact n
    | n > 0        = n * fact (n - 1)
    | n == 0       = 1
    | otherwise    = error "undefined for negative ints"
```

7.4.2 Primitive vs General Recursion

Primitive recursion is recursion where

- the base case considers the parameter value 0
- the recursive case considers how to get from value $n-1$ to n

General Recursion is a recursive function where there is not a precise halting case, but there is a halting condition. For example, in the `divide` function below - the halting case is where `n < m` - not when either of them are at exact values.

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

Page 8

Lecture - Tuples, Strings & Lists

2024-02-12 12:00 Matthew

8.1 Characters & Strings

Haskell comes with both the `String` and `Char` types. In Haskell, a single quote (') is used to denote characters and double quotes (") for strings. For example:

```
ghci> :type 'a'
'a' :: Char

ghci> :type "Sam"
"Sam" :: String
```

The `Char` module defines some useful functions on characters. Such as converting to uppercase or checking if a provided character is a digit.

```
ghci> import Data.Char

ghci> toUpper 'a'
'A'

ghci> isDigit 'a'
False
```

8.2 Tuples

As seen in other languages, we can combine multiple pieces of data into one data type. For example, we may want to combine a name (represented as a string) and a mark (represented as an integer) for example `("Dave", 74)` which would have the tuple type `(String,Int)`. This example can be furthered to a small program which takes two Tuples representing a student and outputs the name of the student with the higher mark:

```
betterStu :: (String,Int) -> (String,Int) -> String
betterStu (s1,m1) (s2,m2)
    | m1 >= m2  = s1
    | otherwise = s2
```

We can define a *type synonym* which can be used in the place of the raw definition of the tuple. For example, if we define the type synonym on the first line below, we can then re-define `betterStu` to use that.

```
type StudentMark = (String, Int)
betterStu :: StudentMark -> StudentMark -> String
betterStu (s1,m1) (s2,m2)
    | m1 >= m2  = s1
    | otherwise = s2
```

Tuples can also be used to enable a function to return more than one value.

8.3 Lists

Lists are the main data structure in Haskell. They are used to store any number of data values *of the same type*. For example the first list below is a list of integers and the second example is a list of strings.

```
[12, 64, -92, 85, 12]
["This", "is", "a", "list"]
```

The type of a list (`l`) is denoted by `[l]`. For example:

```
ghci> :type [True, False, False]
[True, False, False] :: [Bool]
```

The empty list (`[]`) is an element of any list type.

8.3.1 Strings as a List of Chars

Strings in Haskell are simply lists of characters, therefore the type `String` is declared as:

```
type String = [Char]
```

This means that the two expressions below are the same

```
['h', 'e', 'l', 'l', 'o']
"hello"
```

This also means that the operations that we will cover later in this lecture (for example concatenation of lists) therefore also apply to strings.

8.3.2 Lists from Ranges

It is possible to use the operator `..` to generate lists. For example:

```
[3 .. 9]     = [3, 4, 5, 6, 7, 8, 9]
[3.1 .. 9]   = [3.1, 4.1, 5.1, 6.1, 7.1, 8.1, 9.1]
['a' .. 'z'] = "abcdefghijklmnopqrstuvwxyz"
```

We can also add an argument to give steps different from `1` as seen below:

```
[3, 5 .. 15]     = [3, 5, 7, 9, 11, 13, 15]
[0, 0.1 .. 0.5]  = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

8.3.3 List Comprehension

We can use list comprehension to build one list from another list. For example, if we define:

```
aList = [1, 2, 3, 4, 5]
```

THen the list comprehension shown on the first line below will have the value on the second line below:

```
[2*i | i <- aList]

[2, 4, 6, 8, 10]
```

We read this as "take all `2*i` where `i` comes from `aList`". The `<-` symbol represents the set member symbol ($\in$) from Maths.

List comprehension is extremely powerful as can be seen in the following example:

```
bList = [2, 3, 6, 9, 4, 7]

ghci> [mod i 2 == 0 | i <- bList]
[True, False, True, False, True, False]
```

We can take this another step further where we can add a test at the end of the generator. The following example has given all the values that are less than 5 from `cList`

```
cList = [2, 3, 6, 9, 4, 8]

ghci> [i * 2 | i <- cList, i < 5]
[4, 6, 8]
```

We can use list comprehension and a pattern on the left hand side of the `<-` as seen below:

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

ghci> addPairs [(1, 2), (4, 8), (6, 3)]
[3, 12, 9]
```

8.4 Polymorphic Functions

The *Prelude* comes with a number of functions built in, we will now examine some of these. While doing so, we will examine *polymorphism*.

If we consider the `length` function, which gives the number of elements in a list (returning an `Int`), which works for any type of lists. For this reason, we may think of it to have the following types:

```
length :: [String] -> Int
length :: [Bool] -> Int
```

However! The actual type of `length` is given as:

```
length :: [a] -> Int
```

Here, we are using a *type variable* which stands for an *arbitrary type*. By convention `a`, `b`, `c`, ...are used as type variables.

The expressions `[a] -> Int` is known as the *most general type* for `length`

8.4.1 List Functions

We can define polymorphic functions, by not giving a type declaration. Haskell will try and infer the mst general type by looking at the structure of the function.

8.4.1.1 Adding Elements To Front Of List

For lists in Haskell, one of the most used operations is `:`. This adds an element to the front of the list and is of type `a -> [a] -> [a]`. For example:

```
ghci> 3:[5, 7, 2]
[3, 5, 7, 2]
```

8.4.1.2 List Concatenation

The `++` operator can be used to join two lists together:

```
ghci> "hello" ++ "world"
"helloworld"
```

8.4.1.3 Return Element From Specific Position

The `!!` operator returns an element at a given position. For example:

```
ghci> ["fish", "ham", "cheese", "spam"] !! 2
"cheese"
```

Note that eventhough indexing a list like this is common in other languages, we will not tend to use this operator that often.

8.4.1.4 Checking If List Is Empty

The `null` function tests whether a list is empty. For example:

```
ghci> null [1, 2]
False
```

Page 9

Lecture - List Patterns and Recursion

2024-02-19 1200 Matthew

9.1 Patterns, a Recap

Many of the functions we have been using so far have been defined using patterns. These can include literal values, variables, and wildcards. All of these can be seen below in the redefinition of `or` from lecture 3.

```
or :: Bool -> Bool -> Bool
or True _    = True
or False a   = a
```

Patterns can also include Tuples. For example the `fst` and `snd` functions from the Prelude:

```
fst (x,_)  = x
snd (_,y)  = y
```

Note that these projection functions are polymorphic - they work for tuples of data of any kind.

9.2 Lists and List Patterns

As we saw in the last lecture, the : operator will append whatever precedes it to the list which succeeds it. For example:

```
ghci> 4:[7,3]
[4,7,3]
```

This means that every lit can be built from `[]` and `:`, which are constructors for lists.

Note that the : operator is right-associative meaning that it works from right to left adding the right-most to the second right-most, and so on. For example:

```
ghci> 1:2:[]
[1,2]
```

We can use these constructors in patterns. Where we use : in list patterns when we want to deal with the first element and the rest of the list separately. For example, the following example (from the Prelude) returns the first element of a list:

```
head :: [a] -> a
head (x:xs) = x
```

Note that the naming convention `x:xs` is standard; where we say “x, exes”.

It is also possible to use wildcard matching:

```
head :: [a] -> a
head (x:_) = x
```

9.3 Recursion over Lists

As we already know from recursion in the previous weeks, we require a base case and a recursive case. When recursing over lists - the base case considers the empty list `[]` and the recursive case gives the result for any non-empty list (one matched by `x:xs`) from the result of the tail of the list `xs`.

If we take the example of the implementation of a Prelude defined function `sum`.

```
sum :: [Int] -> Int
```

The function is defined to return a sum of a list of integers.

THe base case of our function will be when the list is empty, as the sum of `[]` is obviously `0`. The recursive case will be where `xs` is greater than 0. Therefore, we can define the function as

```
sum []       = 0
sum (x:xs)   = x + sum xs
```

9.3.1 General Recursion over Lists

The above recursive function we have explored is of the type 'primitive recursion'. It is also possible to use general recursion over lists.

For example, we take the Prelude function `zip` which has the following definition:

```
zip :: [a] -> [b] -> [(a,b)]
```

This function joins two lists into a single list of tuples:

```
ghci> zip ['r', 'h', 'a'] [4, 7, 2]
[('r',4),('h',7),('a',2)]
ghci> zip [5, 7, 1, 5] ['a', 'b']
[(5,'a'),(7,'b')]
```

Note that the lists don't have to be of the same length, and that if the lists are different lengths then the last few elements of the longer elements are dropped.

The easiest way to define `zip` is by beginning with the recursive case of two non-empty lists `x:xs` and `y:ys`. In this case we can place `x` and `y` into tuple, and zip up the tails `xs` and `ys` which give us:

```
zip (x:xs) (y:ys) = (x.y) : zip xs ys
```

Now we have one part of the program left - handling when one argument is `[]`. There are three possibilities for this:

- `y` is empty
- `x` is empty
- `x` and `y` are empty

These can all be written together with a wildcard:

```
zip _ _ = []
```

which gives us the entire function:

```
zip :: [x] -> [y] -> [(x,y)]
zip (x:xs) (y:ys) = (x.y) : zip xs ys
zip _ _ = []
```