BSc (Hons) Computer Science

University of Portsmouth

Second Year

Discrete Mathematics and Functional Programming

M21274

Semester 2

Hugh Baldwin

up2157117@myport.ac.uk

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 $A \subseteq B$. 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 $A \subset B$. If A is not a subset of B, it can be written as $A \not\subseteq B$. 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 $A = B$. Alternatively, $A = B$ if the following conditions are true:

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

Operations

The intersection of two sets, A and B is every member in both sets, $A \cap B$. For example, the intersection of the sets $X = \{1, 2, 3, 4, 5\}$ and $Y = \{4, 5, 6, 7, 8\}$ is $X \cap Y = \{4, 5\}$. 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, $A \cup B$. For example, the union of the sets $X = \{1, 2, 3, 4, 5\}$ and $Y = \{4, 5, 6, 7, 8\}$ is $X \cup Y = \{1, 2, 3, 4, 5, 6, 7, 8\}$. 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, $A \backslash B$. For example, the difference of the sets $X = \{1, 2, 3, 4, 5\}$ and $Y = \{4, 5, 6, 7, 8\}$ is $X \backslash Y = \{1, 2, 3\}$. This is the effectively subtracting the sets, $X - Y$.

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 $\overline{A}$

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

Like binary arithmetic, these operations follow a few rules:

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

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

The Power Set

The power set is a set containing all subsets of the set, so if $S = \{a, b, c\}$, then the power set $P(S)$ would be $P(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$. If the set S has n members, $P(S)$ 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 $\text{range}(f) \subset \text{domain}(f)$. e.g. $\text{range}(f) = \{f(x) \mid x \in A\}$.

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 $f : A \to B$, $\text{range}(f) = B$. 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 I

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
It is also useful to know that $p \Rightarrow q$ is equivalent to $\neg p \vee q$

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 $\neg q \Rightarrow \neg p$.

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'

Lecture - Logic II - Quantified Statements

17:00 20/02/24 Janka Chlebikova

Propositional Logic

Propositional logic only applies to simple statements where the only possible states are either true or false. A statement such as "Mr Bean is a Mathematical Professor" (while obviously false) could conceivable be either true or false. However, a statement such as "They are a Mathematical Professor" is not a proposition, as its value depends upon the value of *They*, making it a predicate.

Predicate Logic

An atomic proposition is one where there is only one propositional variable, which can be either true or false. All predicates are statements containing one or more variables. If the values from a given domain are assigned to all the variables, the resulting statement is a proposition. To make a predicate into a proposition, you must replace all the variables with a value.

For example, the predicate $\mathrm{p} : \mathrm{x}\ \mathrm{is\ an\ integer\ less\ than}\ 80$ is not a proposition as the value of x is unknown. If we replace x with a value, for example an integer from the domain $\mathbb{Z}$ such as 10 then we can determine that 10 is less than 80 and therefore p is a true proposition.

A more formal example of this would be the predicate T where $\mathrm{T}(\mathrm{x},\mathrm{y},\mathrm{z}) : \mathrm{x} < \mathrm{y}+\mathrm{z}$ is not a proposition until you add the qualifier that $\mathrm{T}(\mathrm{x},\mathrm{y},\mathrm{z}) : \mathrm{x} < \mathrm{y}+\mathrm{z}\ \mathrm{where}\ \mathrm{x},\mathrm{y},\mathrm{z} \in \mathbb{Z}$

Quantifiers

A quantifier refers to quantities such as "some" or "all", which are used in statements such as "P is true *for some* value of x" or "T is true *for all* values of y". There are two main quantifiers, the "universal quantifier" (represented by $\forall$) and the "existential quantifier" (represented by $\exists$)

Universal Quantifier ($\forall$)

For a predicate P such as "For every x in the domain $\mathbb{Z}$, $\mathrm{P}(\mathrm{x})$", you could rewrite it using the universal quantifier as $\forall \mathrm{x} \in \mathbb{Z}, \mathrm{P}(\mathrm{x})$. This means that for the predicate to be true, every value in the domain, $\mathbb{Z}$ must have a corresponding, true proposition, $\mathrm{P}(\mathrm{x})$. That also means that you only need one counter-example to prove that the predicate is false.

For example, the predicate $\forall \mathrm{x} \in \mathrm{S}, \mathrm{x}^2 > \mathrm{x}$ is true for the set $\mathrm{S} = \{2,3,4,5,6\}$ because we can exhaustively prove that for every value of x, x^2 is greater than or equal to x. $2^2 > 2$, $3^2 > 3$, $4^2 > 4$, $5^2 > 5$, $6^2 > 6$. If the set S was expanded to include 1, then the predicate would be false, as $(1)^2 = 1, 1 \not> 1$.

Existential Quantifier ($\exists$)

For a predicate P such as "There exists an x in the domain $\mathbb{Z}$ such that $\mathrm{P}(\mathrm{x})$ is true", you could rewrite it using the existential quantifier as $\exists \mathrm{x} \in \mathbb{Z}, \mathrm{P}(\mathrm{x})$. This means that for the predicate to be true, there needs to be at least one value of x for which there is a corresponding, true proposition, $\mathrm{P}(\mathrm{x})$. It is much harder to disprove such a predicate, especially if the domain is infinite as in this case, and so it is usually necessary to disprove it logically.

For example, the predicate $\exists x \in \mathbb{Z}, x^2 > x$ is very easy to prove true, since we can find a single value of x, lets say 2, where $2^2 > 2$ is obviously true. It is also easy to disprove a statement such as $\exists x \in \mathbb{Z} \text{ such that } x^2 < -10$, since we can logically say that there is no number for which it's square is negative.

Negations of Quantified Statements

The negation of a quantified statement seems obvious at first glance, but the answer is not just to negate the proposition (e.g. "All mathematicians wear glasses" $\rightarrow$ "No mathematicians wear glasses"), but instead to give the statement matching the counter-example to that statement (e.g. "All mathematicians wear glasses" $\rightarrow$ "There is at least one mathematician who does not wear glasses"). More formally, the negation of a statement in the form "$\forall x \in S, P(x)$" is logically equivalent to a statement in the form "$\exists x \in S, \neg P(x)$", and the negation of a statement in the form "$\exists x \in S, P(x)$" is logically equivalent to a statement in the form "$\forall x \in S, \neg P(x)$".

Nested Quantifiers

Quantifiers can be nested, e.g. $\forall x \exists y$ or $\exists x \forall y$, which can be useful for reducing the ambiguity of a predicate. For example, the predicate "There is a student solving every exercise of the tutorials" could either mean "There is one student who solves every exercise of the tutorials" or "For any given exercise, there is a student who solves the exercise". Obviously the person who wrote the statement had only one of these in mind, and so it would be nice to explicitly define which of the two were intended.

If we let $P(x, y)$ be the proposition "Student x solves the exercise y", S be the set of students and E be the set of all exercises, there are two possible options -

- $\exists x \in S \; \forall y \in E$ such that $P(x, y)$ is true. This is like the previously mentioned possible meaning, "There is one student who solves every exercise of the tutorials"
- $\forall y \in E \; \exists x \in S$ such that $P(x, y)$ is true. This is like the other possible meaning, "For any given exercise, there is a student who solves the exercise"

Lecture - Methods of Proof

17:00 27/02/24 Janka Chlebikova

A mathematical proof is a reasoned argument to convince someone of the truth of a hypothesis. More accurately, an theorem, or argument, is a collection of hypotheses, or premises, followed by a statement known as the conclusion. $(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \Rightarrow q$.

There are several methods of proof which are used depending upon the hypothesis. These include direct proof, proof by contradiction, proof by contrapositive and mathematical induction. There are many more methods of proof out there, but these are the only methods that are covered by this module.

Direct Proof

Theorems are often in the form $p \Rightarrow q$, and can be easily proved by supposing that p is true, and working to prove it as such. You start with one of the hypotheses and make a series of deductions until you reach a conclusion. For example, to prove the theorem below, you start by assuming that the hypotheses are true, and then work from there to reach a conclusion. If the conclusion is that the theorem is true, you have successfully proved the theorem.

Theorem 1. *For all integers* n *and* m*, if* m *is odd and* n *is even, then* $n + m$ *is odd.*

Proof. An integer, n, is even if and only if there exists an integer, k, such that $n = 2k$. Another integer, m is odd if and only if there exists an integer, l, such that $m = 2l + 1$.

As such,

$$\begin{aligned} n + m &= 2k + (2l + 1) \\ &= 2(k + l) + 1 \end{aligned}$$

$\therefore n + m$ is odd. QED.

(QED stands for “Quod Erat Demonstrandum”, or “What was to be demonstrated”)

Proof by Contradiction

Since the conclusion can only be either true or false, if you can prove that the theorem cannot possibly be false, you have arrived at a contradiction, and therefore proved that it is true. For the theorem below, you start by taking the inverse of the hypothesis, and attempt to prove it. Once you reach your contradiction, you can say that the theorem must be true.

Theorem 2. *For all* $n \in \mathbb{N}$*, if* n^2 *is even, then* n *is even.*

Proof. Assume that n is odd, but n^2 is even. Therefore, there exists some $k \in \mathbb{N}$ such that $n = 2k + 1$.

As such,

$$\begin{aligned} n^2 &= (2k + 1)^2 \\ &= 4k^2 + 2k + 2k + 1 \\ &= 4k^2 + 4k + 1 \\ &= 2(2k^2 + 2k) + 1 \end{aligned}$$

This results in a contradiction, as n^2 must be odd since it is in the form $2k + 1$.
$\therefore$ if n^2 is even then n must be even. QED.

Proof by Contrapositive

Rather than proving $p \Rightarrow q$, you can instead attempt to prove the logically equivalent proposition, $\neg q \Rightarrow \neg p$. You essentially use a direct proof of the contrapositive, and since it is logically equivalent to the original hypothesis, you have also proved the theorem. The example for this proof is the same as the proof by contradiction, e.g.

Theorem 3. *For all* $n \in \mathbb{N}$*, if* n^2 *is even, then* n *is even.*

Proof. Assume that n is odd, and n^2 is also odd. Therefore, there exists some $k \in \mathbb{N}$ such that $n = 2k + 1$ and some $l \in \mathbb{N}$ such that $n^2 = 2l + 1$.

As such,

$$\begin{aligned} n^2 &= (2k+1)^2 \\ &= 4k^2 + 2k + 2k + 1 \\ &= 4k^2 + 4k + 1 \\ &= 2(2k^2 + 2k) + 1 \end{aligned}$$

Thus if n is odd, n^2 is odd, and therefore the contrapositive theorem is true.
$\therefore$ if n^2 is even, then n must be even. QED.

Proof by Mathematical Induction

This is one of the most basic methods of proof, and is very useful for proving properties of natural numbers, or proving that an equation holds for an infinite set, such at $\mathbb{N}$ or $\mathbb{Z}$. To prove a theorem, you need the *Basis Step* and the *Inductive Step*. The basis step is a predicate that is defined for some integer n, e.g. "$P(a)$ us true for some integer, $a \in \mathbb{N}$" and the inductive step is a statement, e.g. "For all integers $k \geq a$, if $P(a)$ is true, then $P(k+1)$ is also true". This suggests that $P(n)$ is true for all $n \in \mathbb{N}$ where $n \geq a$.

Theorem 4. *For any integer,* $n \in \mathbb{N}$ *where* $n \geq 1$*, the sum of the first* n *natural numbers is* $S(n) = \frac{n(n+1)}{2}$.

In this case, the basis step would be "For $n = 1$, $S(1) = 1 = \frac{1(1+1)}{2} = 1$", and the inductive step would be "Assume that the sum, $S(n)$, of the first n natural numbers is $\frac{n(n+1)}{2}$. We need to prove that this is also true for the first $n+1$ natural numbers"

Proof. Assuming that

$$S(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

$$\begin{aligned} S(n+1) &= 1 + 2 + \cdots + n + (n+1) \\ &= S(n) + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) \\ &= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

$\therefore$ assuming that the formula is true for n, the formula is also true for $n+1$. QED.

(Dis)proving a Universally Quantified Statement

To disprove a universally quantified statement, you need to find a counterexample for $\forall x \in D : P(x)$ such that $\exists x \in D : \neg P(x)$. For example, the statement

$$\forall n \in \mathbb{N} : (2^n + 1 \text{ is prime})$$

is obviously false. A counterexample for this would be $n = 3$, since $2^3 + 1 = 9$ which is not a prime number. Only a single counterexample is required to disprove a universally quantified statement, which also means that you can prove a universally quantified statement by contradiction or counterexample.

Lecture - Graphs

17:00 12/03/24 Janka Chlebikova

Graphs

A graph, G, is a pair (V, E) of sets, V being the set of vertices and E being the set of edges. Each edge, $e \in E$ is denoted either by a set of vertices, e.g. $\{v_1, v_2\}$ or in shorthand as v_1v_2, from the set V. Each edge connects a distinct set of vertices from V. The vertices v_1 and v_2 are said to be incident with the edge v_1v_2, and are adjacent to each other.

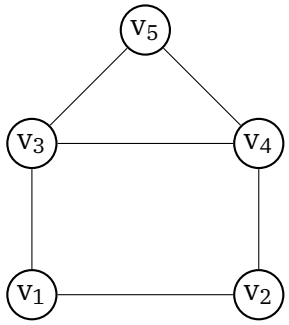


Figure 7.1: A graph with 5 vertices and 6 edges

In the case of the graph in figure 7.1,
$V = \{v_1, v_2, v_3, v_4, v_5\}$
$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_5\}\}$, or
$E = \{v_1v_2, v_1v_3, v_2v_4, v_3v_4, v_3v_5, v_4, v_5\}$

Multi/Pseudo-Graphs

A pesudograph, or multigraph, is a graph containing either an edge that connects a vertex to itself, or forms a 'loop' with two or more edges which connect the same two vertices together. These are not technically graphs, but will still be used in this course as they have some properties which are important to understanding graphs as a whole.

Vertices

There are several properties of vertices, but the one we are most interested in is the *degree* of the vertex. In the case of a graph, the degree of a vertex is the number of edges connected to it. This is denoted by $\deg(v_1)$, and a vertex with a degree of 0 is known as 'isolated'.

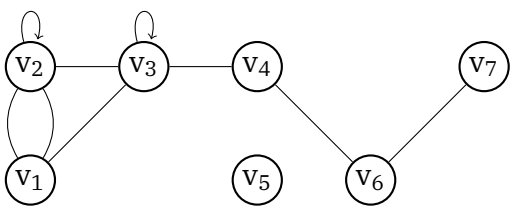


Figure 7.2: A graph with 7 vertices and 9 edges

For figure 7.2, the degree of each node is as follows:
$\deg v_1 = 3$
$\deg v_2 = 5$
$\deg v_3 = 5$
$\deg v_4 = 2$
$\deg v_5 = 0$
$\deg v_6 = 6$
$\deg v_7 = 1$

When $d_1, d_2, \ldots, n$ are the degrees of the vertices in a graph G and $d_1 \leq d_2 \leq \ldots \leq d_n$, the degree sequence of G would be $(d_1, d_2, \ldots, d_n)$. For example, the degree sequence of the graph in figure 7.2 would be $(0, 1, 2, 3, 5, 5, 6)$.

Euler Theorem

Euler's Theorem (sometimes known as the Handshaking lemma) states:

> In any graph, $G = (V, E)$, the sum of all vertex-degrees is equal to twice the number of edges,
> $$\sum_{v \in V} \deg v = 2|E|$$

This has several consequences, namely

- The sum of all vertex-degrees is an even number
- Adding a single edge increases the degree of 2 vertices by 1
- The number of vertices with odd degrees must be even

Special Graphs

Complete Graphs

For any positive integer, $n \in \mathbb{N}$, the complete graph with n vertices, denoted by K_n is the graph in which every pair of vertices are adjacent. More simply, every vertex is connected to every other vertex.

Figure 7.3: K_1

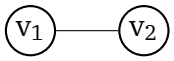


Figure 7.4: K_2

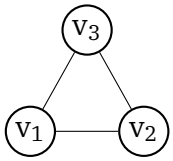


Figure 7.5: K_3

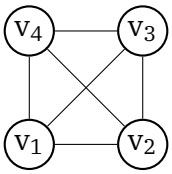


Figure 7.6: K_4

A complete graph with n vertices has $\frac{n(n-1)}{2}$ edges. This is because, as per Euler's theorem, $2|E| = (n-1) + \ldots + (n-1)$, with $(n-1)$ repeated n times. Therefore, $2|E| = n(n-1)$ and finally $|E| = \frac{n(n-1)}{2}$.

Bipartite Graphs

A bipartite graph is one whose vertices can be partitioned into two disjoint sets, V_1 and V_2 in such a way that every edge in E joins a vertex in V_1 with a vertex in V_2. This means that there are no edges connecting any vertices in the same set.

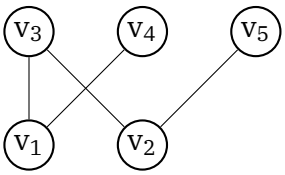


Figure 7.7: A bipartite graph with the sets $V_1 = \{v_3, v_4, v_5\}$ and $V_2 = \{v_1, v_2\}$

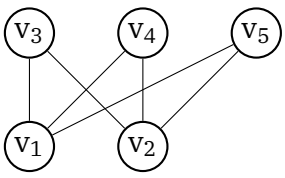


Figure 7.8: A complete bipartite graph, $K_{3,2}$

A complete bipartite graph is one in which every vertex in V_1 is connected to every vertex in V_2. This is denoted by $K_{m,n}$ for $|V_1| = m$ and $|V_2| = n$

Subgraphs

A graph H is a subgraph of graph G if and only if the vertex and edge sets of H are subsets of the vertex and edge sets of G, respectively. The way that the graphs are drawn is irrelevant, since any given graph could be drawn in a near infinite number of ways. The only thing that matters is that the structure of the graph is similar.

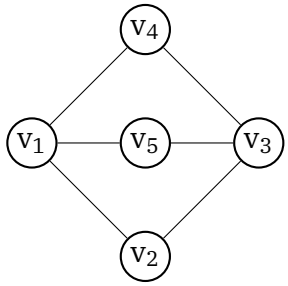


Figure 7.9: G

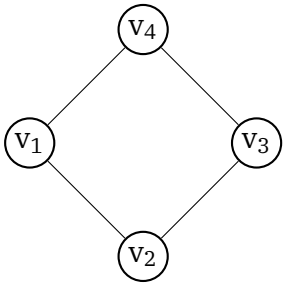


Figure 7.10: H_1

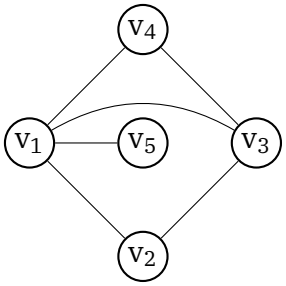


Figure 7.11: H_2

In this case, H_1 (as in figure 7.10) is a subgraph of G (as in figure 7.9), but H_2 (as in figure 7.11) is not a subgraph of G, since it has an extra edge connecting v_1 to v_3.

Isomorphic Graphs

There is not a terribly simple way of describing the concept of isomorphic graphs, but two graphs are isomorphic if it is possible to re-label the vertices of one graph to create the structure of the second graph.

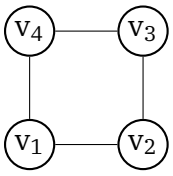


Figure 7.12: G_1

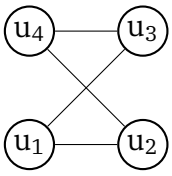


Figure 7.13: G_2

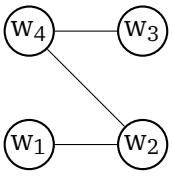


Figure 7.14: G_3

In this case, G_1 and G_2 are isomorphic, but G_1 and G_3 are not.

More formally, G is isomorphic to H if there is a bijective function such that $f : V(G) \to V(H)$, such that if u and v are adjacent in G, then $f(u)$ and $f(v)$ are adjacent in H, and if they are not adjacent in G, they are not adjacent in H either.

It is typically very difficult to prove if two graphs are isomorphic, and as such it is usually easier to prove by contradiction. If G and H are isomorphic, then G and H

- have the same number of vertices
- have the same number of edges
- have the same degree sequence
- are either both connected or not connected (a topic for the next lecture)

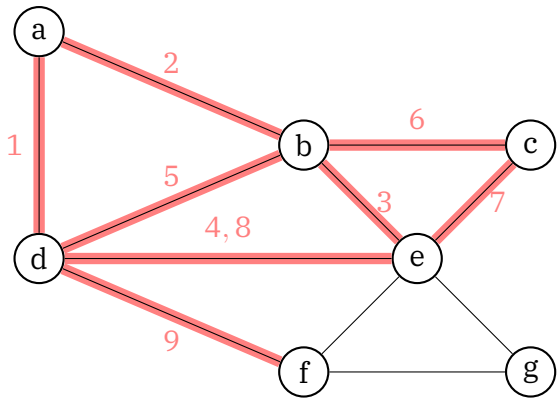
Therefore, if any of the above are not the case, you have proven that the graphs are not isomorphic.

Lecture - Walks, Trails and Paths

17:00 19/03/24 Janka Chlebikova

Walks

A walk is a sequence of vertices and edges, starting and ending with a vertex. There are several notations which can be used, a less formal one, such as a-ab-b-bc-c-ca-a, and the more formal one which I will stick to, being (a, b, c, a). The length of a walk is the number of edges traversed. A walk can traverse any given edge and number of times, anywhere from 0 to ∞. A walk can either be open or closed, depending upon if the walk ends at a different vertex or at the starting vertex, respectively.



The walk shown in red in Figure 8.1 is an open walk, and could be written either informally as d-da-a-ab-b-be-e-ed-d-db-b-bc-c-ce-e-ed-d-df-f, or formally as $(d, a, b, e, d, b, c, e, d, f)$.

Figure 8.1: An open walk of length 9 from d to f

Trails

A trail is a type of walk in which each edge is traversed at most once, and the edges are therefore distinct.

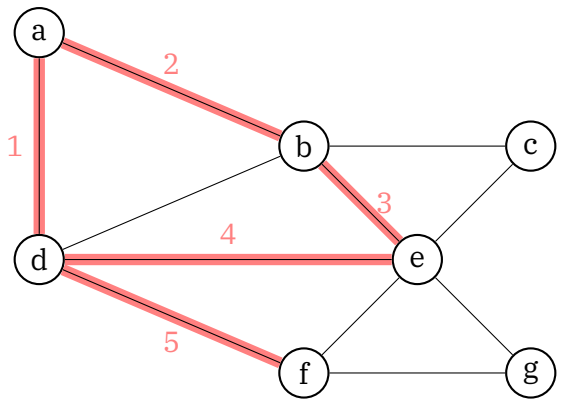


Figure 8.2: A trail of length 5 from d to f

Paths

A path is a type of walk in which each vertex is traversed at most once, and the vertices are therefore distinct.

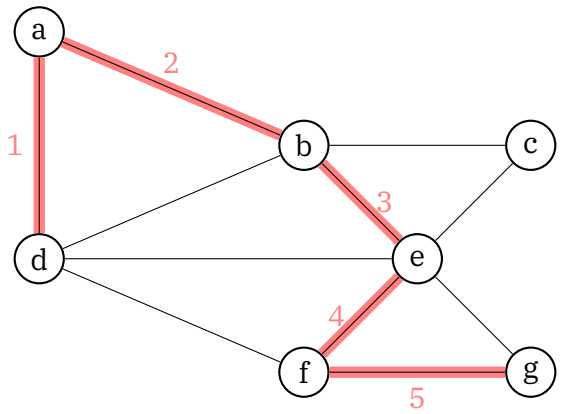


Figure 8.3: A path of length 5 from d to g

Circuits

A circuit is a closed walk which is also a trail (All edges are distinct).

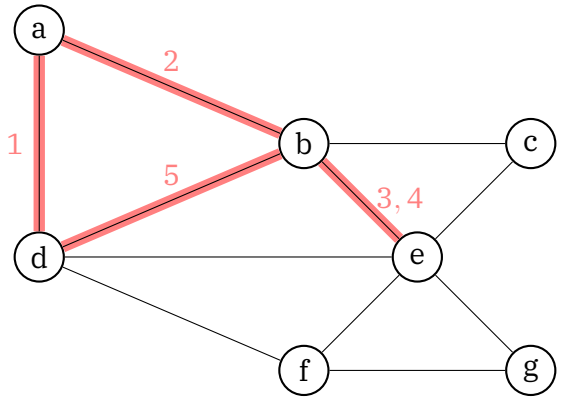


Figure 8.4: A circuit of length 5 from d to d

Cycles

A cycle is a closed walk which is also a path (All vertices are distinct)

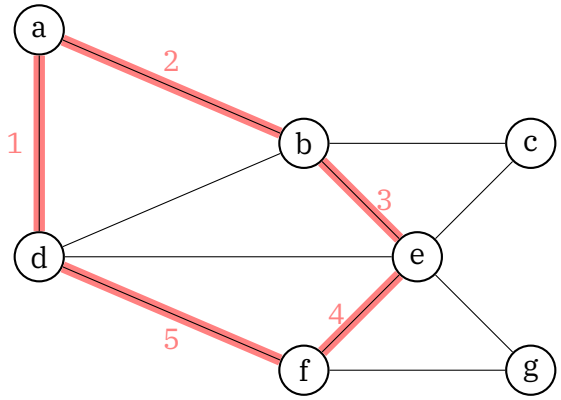


Figure 8.5: A cycle of length 5 from d to d

Connected Graphs & Bridges

A graph is connected if there is a path between any given pair of vertices, otherwise it is disconnected.

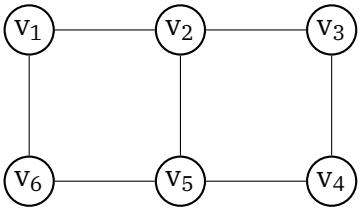


Figure 8.6: A connected graph with 6 vertices

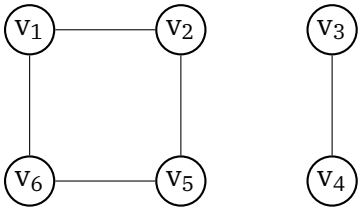


Figure 8.7: A disconnected graph with 6 vertices

A bridge is any edge within a connected graph which, if removed, would case the graph to become disconnected.

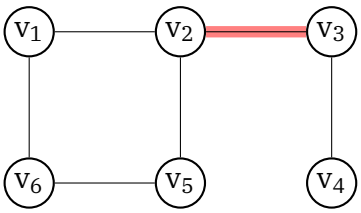


Figure 8.8: A connected graph with 6 vertices and 2 bridges

The graph shown in Figure 8.8 is a connected graph with a bridge. The edge v_2v_3 (highlighted in red) is a bridge as if it were removed there would be no path between the vertices v_1, v_2, v_5 and v_6 and the vertices v_3 and v_4. The edge between v_3 and v_4 is also an edge, as if it were removed there would be no path to v_4.

A graph can have multiple bridges, as there could be several sections which are only connected by a single edge, or as in the example in Figure 8.8, there could be a single node which is only connected to the rest of the graph by a single edge.

Eulerian Graphs

A Eulerian graph is one in which there is a circuit which contains every edge. This would be a closed walk where every edge is traversed exactly once. An easy way to tell if a graph is Eulerian, is if you are able to draw it without lifting your pencil off the paper. It also means that the degree of every vertex must be even. In this case, it is necessary and sufficient, so all you need to prove that a graph is Eulerian is to show that the degrees of all vertices is even.

Fleury's Algorithm

There is an efficient algorithm known as Fleury's Algorithm, which allows you to find an Eulerian circuit within an Eulerian graph. It is as follows:

0. Select a starting vertex
1. Select an edge to traverse, only select a bridge if there are no other edges
2. Remove the traversed edge, and any vertices with a degree of 0
3. Repeat steps 1-2 until all edges have been traversed, and you should end at the starting vertex

Since it does not matter which vertex you start from, or the order in which you traverse edges, there can be several Eulerian circuits in each Eulerian graph.

Semi-Eulerian Graphs

A connected graph with exactly two vertices of an odd degree is known as a semi-Eulerian graph and contains at least one open Eulerian trail, which includes every edge. These trails necessarily start and end at each of the vertices with an odd degree.

Hamiltonian Graphs

A graph is Hamiltonian if it has a cycle which contains every vertex exactly once, and is a closed path. Unlike Eulerian graphs, there is no known generalisation to determine if a graph is Hamiltonian or not. There are several algorithms which are able to construct a Hamiltonian cycle on a Hamiltonian graph, but unfortunately none of them run in a reasonable length of time, and most run in exponential time, with some even running in factorial time.

Adjacency Matrices

Another method for representing a graph is with an *Adjacency Matrix*. For a graph G with n vertices, $v_1, v_2, \ldots, v_n$, the adjacency matrix of G is the matrix $A = n \times n$, whose (i, j) entry is a_{ij} where $1 \leq i \leq n$.

For each a_{ij} in A, $a_{ij} = \begin{cases} 0 & \text{if } v_i v_j \text{ is not an edge} \\ 1 & \text{if } v_i v_j \text{ is an edge} \end{cases}$

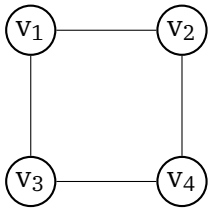


Figure 8.9: The graph, G, with 4 vertices and edges.

For the graph G, as in Figure 8.9, the adjacency matrix would be $A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$

The diagonal entries for a graph, e.g. a_{ii}, are always 0 for a proper graph, since a vertex is never connected to itself, which may not be the case for a pesudograph. The adjacency matrix is also always symmetrical along this line for an undirected graph, since the edge $v_i v_j$ is the same as the edge $v_j v_i$. The degree of v_i is the number of 1's in column i of A.

If you take the square of A, A^2, the entry (i, i) is the degree of v_i. (i, j) is the number of different walks of length 2 between v_i and v_j. This is the case for any power of A, and so it can be generalised as for any $k \geq 1$, (i, j) of A^k is the number of walks of length k between v_i and v_j.

Lecture - Trees

17:00 16/04/24 Janka Chlebikova

A tree is a subclass of graph which is connected, but has no cycles (known as acyclic). Alternatively, you could say that there is exactly one path between any two nodes in the graph. One basic property of such trees is that if the tree contains more than one vertex, it must contain a vertex of degree one. Said vertex is known as a leaf, or terminal vertex.

It is very easy to identify if a graph is a tree, even without drawing it. This is because a graph with n vertices is a tree if and only if it has $n - 1$ edges. If the graph is unconnected, you can automatically discount it from being a tree, since it has to be a connected graph.

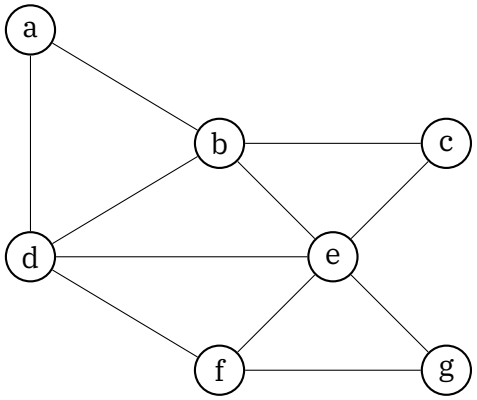


Figure 9.1: A graph, G

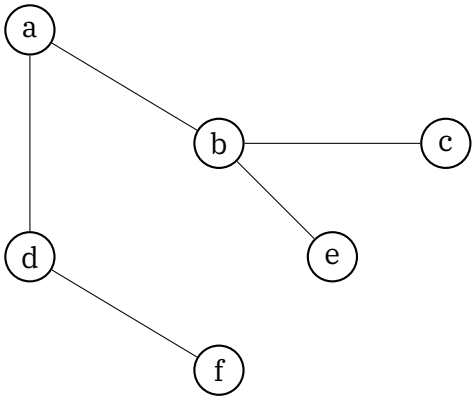


Figure 9.2: A tree, T of graph G

Spanning Trees

A spanning tree H, of the graph G, is a subgraph of G which contains every vertex in G, and is a tree. Any given graph can have multiple spanning trees, each of which is considered different if they are made up of different edges of G.

Finding a Spanning Tree

It is very easy to find a spanning tree of any given graph, G, by using the following method:

0. Check if G contains any cycles. If not, G is itself a spanning tree of itself
1. Delete any single edge in G that forms part of a cycle
2. Repeat step 1 until there are no edges which form part of a cycle. This means that you have found a connected subgraph of G with no cycles, and therefore a spanning tree of G

Another method is to use a depth-first search, which is useful if you need to implement this in code. The method is as follows:

0. Start at any vertex, and give it a label
1. Move to any unlabelled adjacent vertex
2. Label the vertex

3. Repeat steps 1-2 until there are no adjacent unlabelled vertices
4. Backtrack to the first vertex with any adjacent unlabelled vertices, then go back to step 1. If you reach the starting node without finding any adjacent unlabelled vertices, you have found a spanning tree

If you also keep track of the edges you used to traverse the graph, you have a list of vertices and edges which make up a spanning tree of the graph.

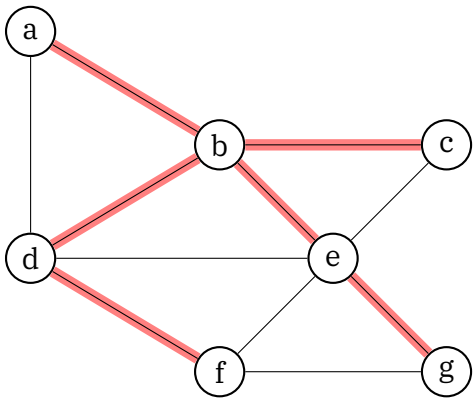


Figure 9.3: G, with a spanning tree, T, overlaid in red

Minimum Spanning Trees

If you now consider a weighted graph, it is obvious that some spanning trees have a lower total weight for all of their edges. The tree with the lowest of these is known as the minimum spanning tree. This is useful for optimising problems such as finding the most efficient path between locations, or the lowest cost way to connect users to a telephone system. While it is possible to use the previous methods to create a spanning tree in a weighted graph, it almost certainly will not be the minimum spanning tree.

For this purpose, there are two known efficient algorithms for finding (one of) the minimum spanning tree(s), namely Kruskal's Algorithm and Prim's Algorithm.

Kruskal's Algorithm

For a graph with n vertices,

1. 'Fill in' the edge with the lowest weight
2. Add the next lowest weight edge, if adding this won't create a circuit
3. Repeat step 2 until you have $n - 1$ edges

This will result in (one of) the minimum spanning tree(s)

Kruskal's algorithm is known as a 'Greedy Algorithm' since it makes decisions based upon what is best at that step, with no regard for the future implications.

Prim's Algorithm

To construct a tree, T of graph G,

0. Start with any vertex
1. Add the edge of lowest weight which is adjacent to T and connects vertices in T with those not in T
2. Repeat step 1 until all vertices are in T

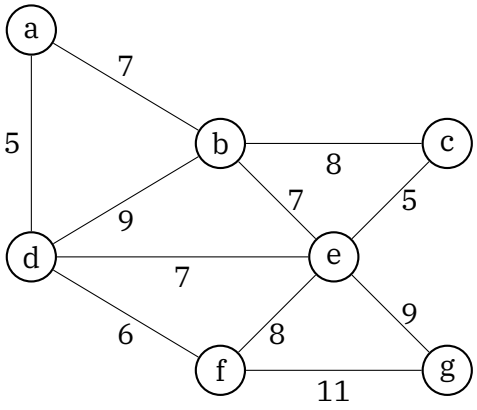


Figure 9.4: The weighted graph, G_2

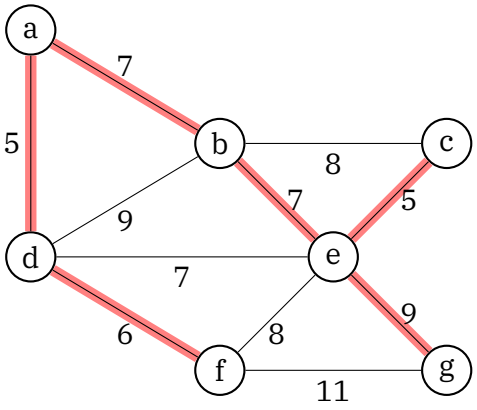


Figure 9.5: G_2, with the minimum spanning tree generated by Kruskal's Algorithm, T_2, total weight 39

Rooted Trees

A tree is a rooted tree if one of it's vertices is specified as the *root*. Each vertex in a tree has zero or more *children* (adjacent vertices moving away from the root), and each vertex (other than the root) has a *parent*, which is the vertex above it, in relation to the root. If two vertices have the same parent, they are known as *siblings*. In the case of T_2, we could say that a is the root, c and g share the parent, e, and as such are siblings.

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
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 12/02/24 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`

Lecture - List Patterns and Recursion

12:00 19/02/24 Matthew Poole

Patterns

Previously we have looked at patterns for defining functions, but they can also be used for pattern matching with lists. Patterns can include literal values, variables or wildcards, but they can also be tuples. For example, the pattern `(x, _) = x` would extract the first value from a tuple and discard the other value. This is exactly how the `fst` and `snd` functions are defined. These functions are projection functions and are polymorphic as they word for tuples of any type.

List Patterns

`[]` and `:` are both constructors which can be used to construct a list. For example, the list `[1, 2]` can be constructed using `[1, 2]`, `1:[2]` or even `1:2:[]`. The last example is how lists are represented internally within Haskell, and the other constructor is just syntactic sugar to make it easier to define lists.

You can also use these constructors within patterns, for example if you want to only deal with the first element of the list. For example, the `head` function in the prelude is defined as `head (x:xs) = x` where `x` matches the first element of the list, and `xs` matches the rest of the list. The naming convention of `x` and `xs` is standard, and means ”x and exes”. In this case, since we don’t care about the rest of the list, `xs` could be replaced with a wildcard. Similarly, the `tail` function uses the same trick but discarding the first element and returning the rest of the list. When using list patterns, if you want to match an empty list, that pattern must appear first as otherwise it will still match any other list pattern.

Recursion Over Lists

As with any other recursive functions, to define a recursive function over a list, you need a base case and recursive case. However, unlike other recursive functions the base case always considers the empty list, and the recursive case always considers a non-empty list, matched by `(x:xs)` and passes the tail of the list into the function call. If we want to recurse over a list and perform a calculation on each element but leave them in a list, you can prepend the element back into the list, within the recursive case. For example, if you wanted to double every item in the list you would use a base case which returns `0`, and a recursive case like `func (x:xs) = x * 2 : func xs`. This function is an example of primitive recursion, but there is also general recursion.

General Recursion (Over Lists)

An example of general recursion is the function `zip` from the prelude. This function joins two lists into a single list of tuples. It’s type definition is `zip :: [a] -> [b] -> [(a, b)]`. In this case, the lists don’t have to be lists of the same type, and if the second list is longer than the first, the remaining elements are just sent into the void. The easiest way to define this function would be to use a recursive function where the recursive case takes two non-empty lists `x:xs` and `y:ys`, and places `x` and `y` into a tuple, before recursing on the tail of the two lists. The base case would need to return an empty list (`[]`) if either of the input lists are empty. This could be written as three separate patterns, e.g.

```
zip (x:xs) [] = []
zip [] (y:ys) = []
zip [] []     = []
```

or using a single base case with wildcards, e.g.

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

Lecture - Higher Order Functions

12:00 26/02/24 Matthew Poole

In most functional programming languages, it is possible to treat a function as a piece of data, passing it as a variable into another function, or returning it from a function. Any function which takes another function as an argument, or returns a function is known as a Higher Order Function. A simple example of a higher order function is one which takes a function and an integer, and applies the function to the integer twice, once to the integer, and then again to the result.

```
twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)
```

As you can see, the syntax is very easy and the type definition allows you to define the specific function signatures that your higher order function can use.

Function Composition

Haskell includes the function composition operator, `.`, which is the equivalent of $\circ$ in maths. For example, with the functions `f` and `g` the effect of $(f \circ g)(x)$ is given by `(f . g) x = f (g x)`. The output type of the first function must be the same as the input type of the second function, otherwise a type error will occur.

The composition operator also allows you to easily define functions in terms of other functions, for example the `twice` function could be rewritten as

```
twice :: (Int -> Int) -> Int -> Int
twice f = f . f
```

Any function defined only in terms of other functions is known as a function-level definition.

Partial Applications

Another feature of many functional programming languages is the ability to partially apply functions. This can be combined with function-level definitions in order to simplify the definition of very basic functions. For example, if there is a `multiply` function, which takes two integers and returns the product, we could define a `double` function as

```
double = multiply 2
```

This is a much simpler way of defining the function, as the unnecessary variable is never defined, and Haskell just applies the function `multiply 2` to the variable.

Higher Order Functions in the Prelude

Mapping

The `map` function takes a function and a list as arguments, and applies the function to each element of the list. It is defined as a recursive function, with the pattern `map :: (a -> b) -> [a] -> [b]` such that

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

This allows you to very quickly and simply apply a function to every element of a list, without writing your own list comprehension each time you need to do it.

Filtering

Another common one is filtering a list. The `filter` function takes a 'property' and a list as arguments, and returns a list containing only the elements of the original list which match that 'property'. The property is effectively a function which returns a boolean value. The prelude definition is as follows

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

Some examples of a property would be `(>0)` or `isDigit`, which would return all values greater than 0, and all digits, if used in conjunction with `filter`.

Folding

The final common one is folding a list. This is essentially taking a list, and *folding* all of the values into a single value. This time, the function takes a function, list and a value to return for an empty list. The prelude definition is as follows

```
foldr :: (a -> a -> a) -> a -> [a] -> a
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)
```

This could be used to add all of the values in a list, `foldr (+) 0 [1,2,3] -> 6`, or to concatenate all of the strings in a list, e.g. `foldr (++) [] ["a","b","c"] -> "abc"`

Lecture - Algebraic Types

12:00 11/03/24 Matthew Poole

As with most programming languages, it is possible to define your own data types in Haskell, known as algebraic types. As shown previously, you can define simple datatypes using the `type` keyword, e.g. `type HouseNumber = Int`. However, it would be hard to represent more complex structures, such as binary trees or graphs, using only simple data types. Algebraic types allow you to represent arbitrary types. These are defined using the `data` keyword, which allows you to define a type, along with all of the values which are valid for that type. For example, if you wanted to define a datatype to store every day of the week, you could do so with the following code
`data Day = Mon | Tue | Wed | Thur | Fri | Sat | Sun`. In this case, the constructors are the data values, or members of the type. This is known as an enumerated type. The boolean type could be defined as `data Bool = True | False`.

Functions with Enumerated Types

The simplest way to define a function on an enumerated type is by using pattern matching. By default, there are no actual operators, such as == for the type. It is possible to define them manually, but this would be very tedious, and there are easier ways of doing so.

You can get Haskell to automatically define the == operator by declaring that the type is a member of the Eq **type class**. We do this by adding `deriving (Eq)` to the type declaration, such as

```
data Day =  Mon | Tue | Wed | Thur | Fri | Sat | Sun
            deriving (Eq)
```

This adds obvious implementations for == and `/=`. There are other type classes which automatically add other useful operators, such as

- `Ord`, which defines the operators <, <=, > and >=
- `Show`, which defines a function `show`, that converts values to strings
- `Read`, which defines a function `read`, that converts strings back into values (e.g. parsing)

Product Types

A product type is an algebraic type with a constructor with two arguments. This was previously done by giving a type name to a tuple, for example

```
type StudentMark = (String, Int)
```

could be redefined as

```
data StudentMark = Student String Int
```

where `Student` is the name of the constructor. This allows you to define types in essentially the same way, but with labels which allow you to tell more easily which type a given item is.

Sum Types

You can combine enumerated and product types to create more complex types where each member can have different variables of different types. For example, a shape type could be defined as

```
data Shape = Circle Float | Rectangle Float Float
```

This would allow you to create two instance of a shape, such as `Circle 9.0` and `Rectangle 4.5 6.0`, both of which could be used as the input to a function with the input type `Shape`. For example, you could use pattern matching to create an `area` function which has a different definition for each variety of shape, i.e.

```
area :: Shape -> Float
area (Circle r)       = pi * r * r
area (Rectangle w h)  = w * h
```

Recursive Types

Algebraic types can also be defined in terms of themselves. For example, if you wanted to represent a binary tree, you could define each node as a type which is either null, or a node with a value, and left and right child. In the case of Haskell, you would define it as a tree, which can either be null or a value and left and right subtree, i.e.

```
data Tree = Null | Node Int Tree Tree
```

Most functions using a recursive type would use null as the base case, and a non-null type for the recursive case.

Lecture - Input/ Output

12:00 18/03/24 Matthew Poole

Real-time input and output is important for any program which requires user-interaction. This is an issue for functional programming languages, since they are typically immutable and therefore don't have great support for using unpredictable values. Another problem is that functions are meant to have a property known as referential transparency. This means that, given the same set of arguments, a function should always return the same value. This is a required property of any pure functional language, and therefore and language with support for real-time input is an impure functional language.

A common approach used by functional languages, is to provide a set of 'functions', along the lines of `inputInt :: Int`, which reads an integer from the standard input and returns the value. This approach breaks the rule of referential transparency since the function returns a different value each time it is called, which is a side effect of reading from the standard input.

Haskell's I/O Approach

Haskell uses a method of I/O known as the *monadic approach*, as it is based upon the mathematical concept of a 'monad'. The input/ output is viewed as a series of actions which happen in sequence. Haskell provides the generic type `IO a`, which represent I/O actions of type a.

A value of type `IO a` is an action which, when executed, performs some input/ output and then returns a value of type a. It also provides a method for sequencing these actions, which allows them to be executed one after the other in a specific order. Effectively, there is another very simple imperative programming language within Haskell which is used for writing I/O programs. As such, a typical Haskell program consists of a set of function definitions which abide by referential transparency and are purely functional, and a set of I/O programs. The I/O programs are able to call the pure functions, as well as each other, but pure functions are only able to call other pure functions, and never an impure I/O program. These programs are an even higher level of abstraction, as they are syntactic sugar for a set of purely functional expressions.

Prelude Functions

There are two built-in functions defined in the prelude for reading input. Namely: `getLine :: IO String`, which reads a line from the standard input, and returns a string; and `getChar :: IO Char`, which reads a single character from the standard input.

The prelude also defines a method of writing to the standard output. There are three functions which are able to write to the standard output: `putStr` and `putStrLn` which write a string and a string with a newline to the standard output, respectively; and `print`, which is a polymorphic function, which writes the value of any type which extends Show to the standard output.

Since every function has to have a return type, Haskell provides the *one-element* type called `()`, which contains a single value, also `()`. All this really means is that the I/O program returns nothing of interest. This is similar to the `void` type that most imperative languages use to represent a function with no return value. Haskell effectively gets this value for free, as it is simply a tuple with no elements.

'do' Notation

As previously stated, an I/O program is a set of actions with a specific order. To write a program, you have to use the do keyword to specify that a function is an I/O program. For example, if you wanted to write a

program which displays a message asking the user to 'Enter a string', gets a string of user input (but does nothing with it), then displays a 'done' message, you would define it as below:

```
readALine :: IO ()
readALine = do
  putStrLn "Enter a string"
  getLine
  putStrLn "Done"
```

Notice that every line consists of an action of type `IO a` for some common `a`.

Usually, you'll actually want to do something with the user input, and so you 'capture' it into a named value (effectively a variable) using `<-`. This appears to be like an assignment operator as in imperative languages, but since this is still Haskell, a named value is immutable, as everything else is. If we wanted to write a similar I/O program as before, but this time, output the value the user entered, we can capture the value they enter, e.g.

```
readALine :: IO ()
readALine = do
  putStrLn "Enter a string"
  line <- getLine
  putStrLn ("You entered " ++ line)
```

If you wanted to get an integer from user input, you would read a value and capture it, and then pass it into the `read` function for an integer, e.g. `do str <- getLine` and then `read str :: Int`. Now you have a problem, since `read` is a pure function, it cannot be called directly from an I/O program. To get around this, Haskell defines a function `return :: a -> IO a`, which does no I/O operations, but simply returns the value of the argument you give it. Putting this all together, you could create an I/O program `getInt` which reads an integer from the standard input, as bellow

```
getInt :: IO Int
getInt = do
  str <- getLine
  return (read str :: Int)
```

File I/O

The prelude also defines the following functions which allow you to read from or write to files.

- `readFile :: String -> IO String`
- `writeFile :: String -> String -> IO ()`
- `appendFile :: String -> String -> IO ()`

For each function, the first argument is the path to the file, and for write and append, the second argument is the string which should be written or appended to the file, respectively.

Conditional Code

The conditionals `if`, `then` and `else` can be used within a `do` construct to allow for more complex functions with flow control. Alternatively, you can move more of the logic to pure functions, allowing you to use guards and pattern matching for more efficient and readable functions, and simply read in the value and then call the pure function with that value.

Local Definitions

As with pure functions, you can use local definitions (with the keyword `let`) in I/O programs to reduce the number of IO functions which need to be used. This is because you can replace an assignment with a local definition, e.g. replace `response <- return (even 2)` with `let response = even 2`.

Recursion

IO programs, like pure functions, can be recursive. The syntax is exactly the same with both, since you are simply calling a function within a function, it just happens to be the same function you are already in.

Haskell Programs

To create an actual program with Haskell, you have to define a main function, the entry point for the program. This is similar to most imperative languages, but typically just immediately calls a pure function. The main function is defined as

```
main :: IO [type]
main = do
  [do stuff]
```

This allows you to run your code as an actual program, using the command `runhaskell <filepath>`

Lecture - Functional Programming in Python

12:00 22/04/24 Matthew Poole

Most modern imperative programming languages include a subset of functional programming concepts, but these are usually limited to certain environments. For example, Python includes list comprehensions, lambda expressions and higher-order functions, some very important functional programming concepts.

Functions are Data

In Python, any function definition creates a new variable, which acts as a pointer to the function. This is useful, as it allows you to define new functions based upon existing ones, but it also means that it is possible to accidentally cause variable hiding by using the same name for a function and variable. It is possible to assign variables to the value of a function-valued variable, which then acts as a pointer to that function, but it is also possible to assign a different value to the function-valued variable, causing that function to be lost.

List Comprehensions

List comprehension in Python is very similar to Haskell, as they create new lists from the values of an old list, and a function. For example, `[i*2 for i in list1]` would return a new list containing each value from `list1` multiplied by 2. You can also add a test at the end, using the `if` keyword, e.g. `[i*2 for i in list1 if i>2]`.

...Using Tuples

In Python, a tuple is effectively an immutable list, which typically contains only a few values. This also works in a similar way to Haskell, in that you must define the 'pattern' which a tuple follows, before you can take any values from it.

Lambda Expressions

As in Haskell, Python includes anonymous functions known as *lambda expressions*. A simple example of this would be `lambda x : x * 10`, which represents a function which multiplies its argument by 10. Higher-order functions can also be used as an argument for a lambda expression, allowing you to return a function from a function.

Higher-Order Functions

Python includes built-in functions such as `map`, `filter` and `reduce`, which take a function and an list and apply the function to each element of the list. The `map` and `filter` functions are the same as those in Haskell, and the `reduce` function acts very similarly to the `foldr` function. The `reduce` function can also take a third argument, which becomes the starting value for the application of the function to the first element in the list.

Other Benefits

Another benefit of higher-order functions is that any local variables used in a returned function continue to exist, even if the function they were defined in has already exited. This allows you to define static variables for a function without having to make them global. A returned function which uses this is known as a *closure*.

Recursion

It is usually said that a recursive function is easier to read than an equivalent iterative function, but the argument could be made that recursive functions are almost always slower than their iterative equivalent. This is often caused by repeatedly computing intermediate results, which can be solved using higher-order functions and a dictionary. This is often called *Memorization*.

Dictionaries

In Python, a dictionary is an unordered collection of data, whose values are accessed via a key. Dictionary literals are defined as a sequence of `key:value` pairs between braces, e.g. `{"eggs": 2, "cheese: 1}`. The `in` operator gives an easy way to test if a dictionary contains any given key.

Memorization

A general purpose memorization function is used, which is passed the function to be optimised, and returns a new function, which we can use to overwrite the old function-valued variable, creating a runtime-optimised function. An example of this optimisation function is below–

```
def memorize(f):
  cache = {}
  def g(arg):
    if arg in cache:
      return cache[arg]
    else:
      cache[arg] = f(arg)
      return cache[arg]

  return g
```

You would then define a function, e.g. `def fibonacci`, and then assign the function-valued variable to the optimised function, e.g. `fibonacci = memorize(fibonacci)`.