CITS2211

Daniel Paparo

1 Lecture 1

1.1 Topics to be discussed

- Logic: Removes ambiguity from natural languages.
- Proofs: Chaining of logical deductions
- Sets and relations: A bunch of objects collected together.
- Countability: Difference between finite sets and infinite sets.
- Finite state machines: Referring to automata - abstracting towards finite state machines for tasks like processing context free languages
- Turing machines: Most complex automata

1.2 Propositions

Natural languages are not precise enough to describe logic in a way that is definite. Propositional logic is one such language that can be used to more accurately describe a set of propositions / statements in a logical way. These can be used to prove the correctness of a program - this will be done for critical software.

Logic: The study of correct and incorrect reasoning.

Logical languages require exact specification of parameters and propositions.

Proposition: a statement that is usually either true or false.

Primitive propositions can not be broken down any further, however can be combined with connectives to generate more complex structures.

Symbol	Joint	Meaning
$\vee$	Disjunction	Or
$\wedge$	Conjuction	And
$\neg$	Negation	Not
$\rightarrow$	Conditional	If/then
$\Longleftrightarrow$	Biconditional	If and only if

And these connectives must be arranged into well formed formula:

Well formed formulae
$\neg\alpha$
$(\alpha \vee \beta)$
$(\alpha \wedge \beta)$
$(\alpha \rightarrow \beta)$

Well formed formulae
$(\alpha \iff \beta)$

It is important to note that the main connective is the one with the least brackets around it. The meanings between propositions can be determined by using a truth table. A formal language must have the following:

- Permitted symbols
- Rules defining the use of symbols (syntax/grammar)
- A description of well formed formulas

To prevent ambiguity, the following order of operations can be assumed:

Order of operations
Brackets
Negation
Conjunction
Disjunction
Implication
Equivilance

2 Lecture 2

The concept of equivilance becomes important: we can simply use a truth table for both propositions and check for equality, however the following equivilances are useful for proofs:

Expression	Equivalent to	Name of Rule
		Commutativity
$P \vee Q$	$Q \vee P$	
$P \wedge Q$	$Q \wedge P$	
		Associativity
$(P \vee Q) \vee R$	$P \vee (Q \vee R)$	
$(P \wedge Q) \wedge R$	$P \wedge (Q \wedge R)$	
		Distributivity
$P \wedge (Q \vee R)$	$(P \wedge Q) \vee (P \wedge R)$	
$P \vee (Q \wedge R)$	$(P \vee Q) \wedge (P \vee R)$	

Expression	Equivalent to	Name of Rule
		De Morgan's laws
$\neg(P \vee Q)$	$\neg P \vee \neg Q$	
$\neg(P \wedge Q)$	$\neg P \wedge \neg Q$	
		Implication
$P \to Q$	$\neg P \vee Q$	
		Contrapositive
$P \to Q$	$\neg Q \to \neg P$	
		Excluded middle
$P \vee \neg P$	T	double negation
P	$\neg(\neg P)$	
		And absorption
$P \wedge T$	P	

The above equivilances are useful for simplifying complex if statements in programming, for example:

```
if(x>0 || (x <= 0 && y > 100))
```

could be rewritten as:

```
!(x <= 0 && y < 100)
```

Which can quiet easily be found graphically.

Propositions fall into three main categories:

2.0.1 Tautologies

These are compound propositions that hold true for all possible values:

P	$P \vee \neg P$
T	T
T	T
F	T
F	T

Tautologies can be simply found by usingi the axoioms of propositional logic, which can be used to derive all possible tautologies.

2.0.2 Contradictions

These are compound propositions that hold false for all possible values:

P	$P \wedge \neg P$
T	F
T	F
F	F
F	F

2.0.3 Contigent

These are every proposition that isnt either a tautology of a contradiction. T prove this is the case it will need to be simply proven with a truth table - that is that if there is at least one example of some values that give a true and one example that gives a false then the arugment will be contingent.

Tautologies can be found by using the axioms of propositional logic. These can be used to derive all tautologies. Alls sets of tautologies can be expressed with a more minimal set of axioms.

3 Lecture 3

3.1 Predicate logic

In predicate logic, the propositions are atomic statements that can either be $true$ or $false$. There are some limitations to propositional logic, however: What we need is the ability to make statments about objects, and assign them attributes.

Predicate: THis is essentially a complete sentence/proposition with empty placeholders in which we can add variables/objects. The truth will epend on one or more terms.

Another motivation for predicates is the requirement for quantifiers. This can select some, all or none of the objects in a domain.

$\forall$: For all $\neg\forall$: not all $\exists$: there exists $\neg\exists$: there does not exist

In predicates there exists both constants (a, b, c, d) and variables (x, y, z). Normally constants are taken from the start of the alphabet and variables taken from the end of the alphabet.

Domain of discourse: This will essentially establish a domain in which are predicates hold.

In different domains the same predicates can have different truth values. There are cases in which variables can be **free** and **bound**. Predicates with all variables being bound are called **sentences**.

Bound variables: A variable that has been introduced by a quantifier within its domain of discourse. **Free variables**: A variable that has not been introduced in this domain of discourse.

The semantics of a sentence (the truth values) are derived from:

- The sementics of all the constants, variable and predicates
- The truth values for all the sub parts
- The truth tables for the connectives

All sentences, in predicate logic, are either true or false.

Swapping the order of quantifiers can give vastly different meanings. Negatated quantifiers give the following equivilances:

$$\neg\exists_x P_x \equiv \forall_x \neg P_x$$

$$\neg\forall_x P_x \equiv \exists_x \neg P_x$$

4 Lecture 4

4.1 Validity

Proving validity is far more difficult than proving invalidity. To prove invalidity we must simply find a counter example.

4.2 Satisfiability

Satisfiability: There are some values within the domain for which the statements will be true.

Satisfiability is used in terms of thousands of variables. Often we need to create some kind of situation for which the problem is solved for the given domain. To prove that somethng is satisfiable, we must simply present a single example for which is is true.

Example:

$$\forall_x . P(x) \rightarrow Q(x)$$

Can be satisfied simply by $P(x)$ being a subset of $Q(x)$.

These problems will almost always be solved by computer programs.

5 Lecture 5

Proofs are an important part of proving many things in computer science, including proving the correctness of a program.

Proof: An argument for a statement that must hold true if you accept the premises (axioms) and the rules that it uses

5.1 Logical deductions

Axiom: Is a proposition that is simply accepted to be true.

Proof: A sequence of logical deductions from axioms and previously proven statements.

Anticedents: Already proved and are above the line. The statement below the line is the conclusion.

Logical deductions or inference rules can be used to prove new propsitions.

We will typically look at the **Soundness** of an argument, this is that the following rules are true, and the **Completeness**, that any statement that is true can be derived by following the rules.

The addition of inference rules turns propositional logical into a formal system (that is; a calculus).

Argument: A collection of propositions, one of which, can be referred to as the conclusion which is justified by the premises.

An argument is considered **valid** when the conclusion is implied by the axioms.

6 Lecture 6

6.1 Direct proof

A direct proof is given in the general form:

$$P \rightarrow Q$$

In order to prove this we must give:

1. Assume that P holds
2. Make logical deductions for a few steps
3. End with a "therefore..."

6.2 Contraposetive

This is an extension of the direct proof in the form:

$$\neg Q \to \neg P \equiv Q \to P$$

This will then simply use the same proof method as a direct proof.

6.3 If and only if

Proving an "if and only if" statement. This can be done by breaking the biconditional up into its parts:

$$P \leftrightarrow Q \equiv P \to Q \land Q \to P$$

These will then simply be used to prove using the direct proof on both sides.

6.4 Proof by cases

Dividing a single proof into multiple smaller subproblems. This is a common theme in programming problems.

6.5 Proof by induction

With some parametized statements in the form:

- Suppose P is some predicate with the domain...

What we are saying is:

- I can prove this for the base case, and I can prove this for one case, and one case greater than it, therefore it is proven for all cases.

This works for using infinite reasoning and will depend greatly on the domain. Commonly the domino topping analogy is used for this. The steps to solve are as follows:

1. State that the proof uses proof by induction
2. Define a predicate $P(n)$
3. Prove that $P(0)$ is true
4. Prove that $P(n)$ implies $P(n+1)$ for any natural number.
5. invoke the induction by ending the proof with the statement *"So it follows that by induction $P(n)$ is true for all natural number*

7 Lecture 7

7.1 Proof by contradiciton

What we are essentially doing is a saying: *"This is not false therefore it must be true"*. We are proving P by assuming $\neg P$, then deriving a false.

1. P is assumed to be false, that is that $\neg P$ is true
2. It is shown that $\neg P$ implies two mutually contradictary assertions, $Q \wedge \neg Q$
3. Since Q and $\neg Q$ cannot both be true, theassumption that P is false must be wrong, and thus P must be true.

8 Lecture 8

A set is an unordered collection of objects without any repitions. Set theory extends predicate logic and introduces the new $\in$ symbol - this is the "element of" symbol.

The brackets $\{\}$ are used to denote a set. $\{1\} \in A$ is pronounced as "The set containing 1 is an element of A". Elipses can be quite useful to not need to write long lists when we can simply imply a pattern.

$\emptyset$: The empty set
$\mathbb{N}$: Natural numbers
$\mathbb{Z}$: Integers
$\mathbb{Q}$: Rationals
$\mathbb{R}$: Reals
$\mathbb{C}$: Complex numbers

The vertical $|$ can be used (sometimes the colon :) to be read as "such that". This creates a predicate like structure.

Subset: A is a subset of B is denoted by $A \subseteq B$. This can also include cases in which every element of A is also an element of B.

Proper subset: The set A is a proper subset of B, denoted by $A \subset B$ if $A \subseteq$ but $\exists_b \mid b \in B, b \notin A$

In order to test if A is a subset of B we can use the following algorithm:

```
for each in A:
    if x in A:
        continue
    else:
        not a subset
```

To test if something is an element we can use the following algorithm:

```
for each element of A:
    if a = x:
        it is an element

it is not an element
```

It is important to note that the empty set can be a member of another set - and more generally: a set can be a member of another set.

Union: The union of the sets A and B is elements appearing in A, B, or both:

$$A \cup B = def\{x \mid x \in A \vee x \in B\}$$

Intersection: The intersection of A and B is all elements appearing in both A and B:

$$A \cap B = def\{x \mid x \in A \wedge x \in B\}$$

To prove equality we must demonstrate that two sets are equal:

1. Prove that $A \subseteq B$
2. Prove that $B \subseteq A$

To do this it is necessary to show that an arbitrary element $a \in A$ is also contained in B.

8.1 Difference

8.1.1 Set difference

This is denoted by

$$A \setminus B$$

or

$$A - B$$

and features things that are "in set A but not in set B"

8.1.2 Symmetric difference

This is denoted by:

$$A \triangle B$$

and are elements that are in A, or elements that are in B, but not both. This can be shown in:

$$A \triangle B \equiv (A \cup B) \setminus (A \cup B)$$

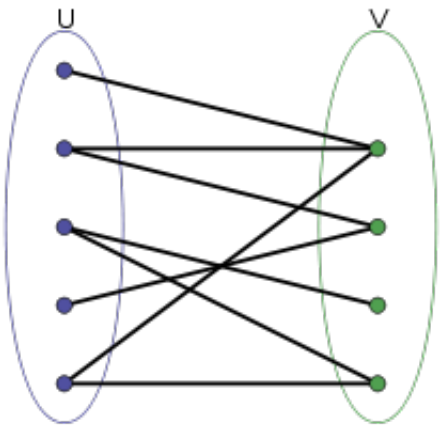
8.2 Complimentation

This requires the existance of a $\mathscr{U}$ universal set The universal set means that all sets must be a subset of $\mathscr{U}$. The compliment of A is:

$$\complement A = \{x \in \mathscr{U} \mid x \notin A\}$$

We can also work with De Morgan's laws:

$$\complement(A \cap B) \equiv \complement A \cup \complement B$$
$$\complement(A \cup B) \equiv \complement A \cap \complement B$$



Bipartite Graph

8.3 Power set

The powerset $\mathcal{P}$ is the set of all subsets of A as its elements:

$$\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$$

Note: that $\{x\}$ is called a **singleton**

8.4 Cartesian product

This is extremely important for computer science, and is defined in the following way:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

That is to say that: *"The cartesian product are the ordered pairs whose first coordinate is an element of A and the second is an element of B, and is all possible combinations"*

9 Lecture 10

9.1 Binary relations

These are purely relations between two things (called tuples). A subset of the cartesian product of two sets:

$$R \subseteq A \times B$$

in this example A is the domain and B is the co-domain. We can also use infix notation, where:

$$R \subseteq A \times B \equiv ARB$$

which is often easier to read. We can also use a **bipartite graph** representation as a representation of the relation:

What we are really trying to find is:
If $(a, b) \in R$ then a is related to b,
If $(a, b) \notin R$ then a is not related to b,

9.1.1 Types of relation

- One-to-one : both a and b only appear in the pair (a, b) and none other
- One-to-many : $a \in A$ appears atleast in one pair, but $b \in B$ appears in at most one pair.
- Many-to-one : $b \in B$ appears atleast in one pair, but $a \in A$ appears in at most one pair.
- Many-to-many : no restriction on the occurance of $a \in A$ or $b \in B$

9.1.2 Relations over A=B

in this case we have the relation of:

$$R \subseteq A \times A$$

9.1.3 Properties of a relation

A relation can have the following set of properties. Depending on what properties it posses it can be categorised further:

- A relation can be reflexive (**R**) - that is that "somthing is always related to itself"

$$\forall_a \in A.(a, a) \in R$$

- A relation can be symmetric (**S**) - that is that things are always related to eachother, as in siblings

$$\forall_{a,b} \in A.(a, b) \in R \rightarrow (b, a) \in R$$

- A relation can be transetive (**T**) - such as in "melbourne is reachable from sydney, and cairns is reachable from melbourne, therefore cairns is reachable from sydney"

$$\forall_{a,b,c}((a, b) \in R \land (b, c) \in R) \rightarrow (a, b) \in R$$

- A relation can be antisymmetric (**A**) - that is that there are "no two way streets, except self loops"

$$\forall_{a,b}((a, b) \in R \land (b, a) \in R) \rightarrow a = b$$

What we are really doing with these tuples is looking at them and then seeing what type of relation they are.

9.1.4 Closure of a relation

When we are looking at the closure of a relation, what we are really saying is "what would we need to add, in order to make this property hold". We can have:

- Reflexive closure, and is given by:

$$S = R \cup \{(x, x) : x \in X\}$$

- Transitive closure
- Symmetric closure, and is given by:

$$S = R \cup \{(x, y) : (y, x) \in X\}$$

It is important to note that we cannot have antisymmetric closure.

9.2 Functions

Functions turn up in all scientific fiends. The function simply acts as a black box.

Function: A kind of relation that has a domain A and a codomain B that is a relation of $f \subseteq A \times B$ with the property of every member of the domain being mapped to exactly one member of the codomain - this can be many-to-one or one-to-one

An obvious draw from this is that there is exactly one arrow leading from every element of the domain A onto the codomain B in the Bipartite graph. The function is shown as:

$$f : A \to B$$

and is pronounced as "f is a function which maps A to B". Functions can be listed exhaustively, however this will only work for small domains. Instead we can use rules to express this, and we find the following properties:

- **Injunctivity**: the function maintain distinctness - i.e must be a one-to-one relationship.
- **Surjunction**: the function covers accross the entire codomain
- **Bijunction**: the function is both a surjunction and an injunction

10 Lecture 11

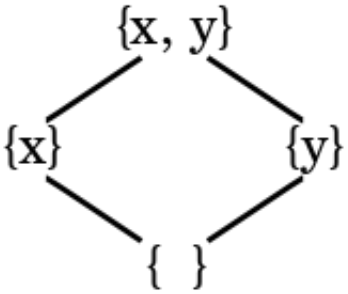
10.1 Partial orders

This is any binary relatipon that is reflexive, transitive and antisymmetric. An example of this is the "less than" relation.

Note: if aRb and bRa then we say that a and b are comparable.

Total order: A partial order R on a set A with the additional property that for any $a, b \in R$, a and b are comparible.

	Equivilance Relation	Partial Order
Reflexive	Y	Y
Transitive	Y	Y
Symmetric	Y	N
Antisymmetric	N	Y



Hasse diagram

An example of a partial order is the "less than or equal to" relation. This also has the additional property: "If a is not less than or equal to b, then b is less than or equal to a.

Incomparible: Neither $A \subseteq B$ nor $B \subseteq A$ holds.

A total order, when graphed, will end up being a line.

10.1.1 Hasse diagram

This is a variant of the directed graph diagrams for partial orders (only). The idea is to remove any edges which can be deduced from the properties of a partial order. That is that:

- Loops do not need to be drawn
- No need to draw $A \to C$, if $A \to B$ and $B \to C$ as we can infer transitivity
- get rid of the arrows, lines only required

Diagrams are not sufficient to prove something, they only provide a good way of finding counter exampels. We must use rules and logical deduction to discover facts about the relation.

10.1.2 POSET

This stands for "partially ordered sets". The most prominant partially ordered sets are:

- $\mathbb{N}$ with the relation $\leq$
- Subsets of some set with the relation $\subseteq$
- A subset of $\mathbb{N}$ with the relation $|$

Covers: A collection of sets in which X is contained as a subset of the sets.

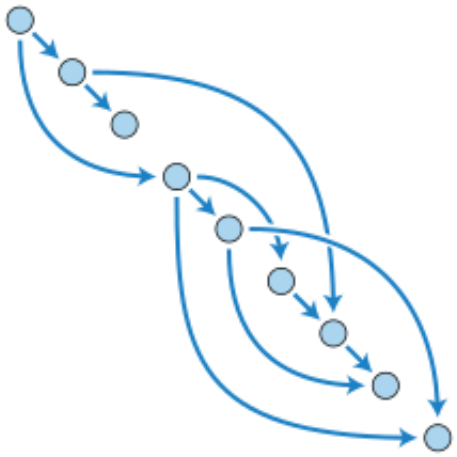
DAG: A directed acyclical graph

10.2 Counting

Fundamentally we are interested with finding the size of a space or time complexity of something we are running.

Multiplication principal: If there are n_A possible outcomes of an event A, and n_B possible outcomes of event B (givent that these are independent events), then:

$$n_A \times n_B$$



A directed acyclical graph

possible outcomes can arise from this sequence of events.

This can be extended to many events simply through multiplying. In its basic form, we get the equation:

$$n^m$$

11 Lecture 12

| means "the divisor of"

For all equivilance relations the following must hold:

- $a \in [a]_R$
- $\forall_{x,y} y \in [a]_R \rightarrow (x, y) \in R)$
- $b \in [a]_R \rightarrow [b]_R = [a]_R$
- $b \notin [a]_R \rightarrow [b]_R \cap [a]_R = \emptyset$

What we can do is list off all of the available values by hand and then select those that are in the relation.

Clique: A graph in which every node is directly checked to all other nodes

For an infinite set we must prove these symbolically. Being in the same equivilance relation means that they will be in the same set. An equivilance class will form a partition as this will divide the set into discrete steps.

12 Lecture 13

Cardinality: A bit like the size, but works with more sets

For a finite set: the cardinality is simply the number of members within it. For infinite sets = $|\mathscr{U}|$. Note that cardinality is denoted by: $|something|$.

We can use a function to count the cardinality of a set.

- Surjunction: If A covers all of B then $|A| \geq |B|$

- Injunction: If all of A maps to B, but still has spaces left over, then $|A| \leq |B|$
- Bijunction: All A maps individually to B: $|A| = |B|$

12.1 Definitions for an infinite set

12.1.1 A

The set A is infinite iff there does not exist any $k \in \mathbb{N}$ such that there is a surjunction:

$$f : \mathbb{N}_k \to A$$

12.1.2 B

The set A is infinite iff it has a proper subset with the same cardinality. Tht is that A is infinite iff:

$$\exists_B : (B \subset A) \wedge (|A| = |B|)$$

We are using zFc : there are two sets we are allowed to start with:

- $\emptyset$: the empty set
- $\mathbb{I}$: the infinite set

There are two rules for the empty set:

- The empty set is a member of the infinite set
- If we pick some set x which is a member of $\mathbb{I}$ then the set $\{x\}$ is also a member of $\mathbb{I}$

The implication if this is that sets will have the same cardinality even when they are seemingly of "different sizes", as in the case:

$$|\mathbb{N}_{>0}| = |\mathbb{N}|$$

12.2 Countability

Something is countable if for set A there is a biunction $f : \mathbb{N}_0 \to A$ or A is finite.

From this we can prove that $\mathbb{Z}\$$ is countable. That is we must be able to list all function in the same order.

13 Lecture 14

To prove that something in uncountable we must prove that something is not in our list of lists. It is good to think about proof by contradictions as an argument. There are two obvious facts about countability:

- If $S \subseteq T$ and T is countable, then so is S - That is that any subset of a countable set is countable itself
- If $S \subseteq T$ and S is uncountable then so is T - that is that a superset of any uncountable set is itself uncountable.

13.1 Cardinality of an infinite set

The cardinality of an infinite set is:

$$|\mathbb{N}_{n>0} = \aleph_0$$

13.1.1 Cantors theorem

Fpor any set A, there is no function $f : A \to \mathcal{P}(A)$ that is a surjunction. That is that:

$$|A| < |\mathcal{P}(A)| < |\mathcal{P}(\mathcal{P}(A))|$$

Therefore there are an infinite number of different sized infinities. The power set will always be equal to or larger than the original set. It is important to note that unioning an infinite set with a finite set will not increase the cardinality.

14 Lecture 15

We need to look at issues such as memory, information, nondeterminism and complexity. We look at computation so that we know what can and cannot be computer.

Finite state machine (FSM): Essentially a cinoyter with a very sall memory and a finite number of states

14.1 Finite state machines

A parking ticket machine is an example of a finite state machine. These react to some stimuli, normally in the form of money being inserted into the machine. All finite state machines require some form of stimuli. Finite state machines are defined by:

- Σ : The alphabet of the machine
- Q : the set of all states
- $Q_0 \in Q$: the starting state
- $F \subseteq Q$: the set of accepting states
- $f : Q \times \Sigma \to Q$: the transformation function

Output can be implicit in the state or the transition, however we do not cover this in this unit. The FSM will require a start state, denoted by the arrows. THe accepting state will be denoted by a double circled state in which the FSM will be satisfied with the outcome of the input.

14.1.1 Trace

The trace can be used to describe the behaviour of an FSM. The trace of an FSM M is a finite sequence of states and transitions labels that can be used to describe the path of transitions for any particular input.

- The trace is written as $s_0, i_1, i_2, ..., i_n, s_n$
- The first state, s_0 must be the start state of M.
- Every triple $(s_j, i_{j+1}, s_{j+1}$ must be of the state transition relation of M

What the trace is essentially showingg is the path that some input will follow within our FSM.

14.1.2 Accepting states

The FSM accepts the input if the final state is an accepting state. All other inputs are not accepted as a member of the language.

It is important to state any assumptions that have been made in an FSM question. For tests, if there is any ambiguity then state the assumptions that have been made.

14.1.3 Language and recognisers

A set of strings accept by a FSM is known as a language that is recognised by the FSM:

$$L(M) \subseteq \Sigma^*$$

The question might be asked: "What language cant be recognised by our FSM

15 Lecture 16

15.1 Non Deterministic FSM

A regular FSM will have only a single trace per input. A NFSM allows for multiple outputs for a single input at each state and as such will have a tree structure for the trace - there are multiple potential traces for a NFSM. A NFSM can have multiple transitions or ϵ (the empty input symbol) at any state in particular. These are mostly the same as any other FSM. The NFSM requires everything that a DFSM requires. In fact, a deterministic FSM is actually a member of the NFSM. If there is at most one trace for an input sequence then it is a DFSM, otherwise it is a NFSM.

It is important to note that every NFSM can be expressed as a DFSM, except they will become very ugly very quickly. The power of this is that "guessing" (in a NFSM) and "calculating" (in a DFSM) have the exact same degree of power.

16 Lecture 17

16.1 Regular expressions

Regular expressions offer a way to compactly define a language of a FSM. These regular expressions can be used to search for things following the rules of the expression. There exists "regex" which are used in many programming languages, and will be a superset of the features of regular expressions, however we will not be covering those in this course.

The following are regular expressions:

- Use the $\emptyset$ symbol and ϵ empty character
- Any symbol $i \in \Sigma$ is a regular expression
- If A and B are reguar expressions then so is AB, $A + B$ and A^*

Brackets can be used for grouping and removing ambiguity. The following rules can be applied for a regular expression:

- $\emptyset$ is the empty language
- ϵ is the language containing only the empty string
- i represents the language defined by $\{i\}$

- AB represents all strings in the form $\alpha\beta$ in which $\alpha \in A$ andn $\beta \in B$
- $A + B$ is the union of A and B which generates a new set from this
- A^* represents all possible concatenations of A

Not all languages are regular. We can capture some as sets, but we can use the diagonalisation argument to show what is and is not regular. Regular expressions are countable. Its important to note that once we give something a definition we make it countable (in this context) as all strings are only finitely long, and as such can be trivially tested with the diagonalisation argument. Almost anything that we can write down is countable.

To do a diagonalisation argument we need to do something that will not be diagonisable using our given rules

16.1.1 Kleene's theorem

Kleene inveted the regular expression. He determined that a language is regular iff it is a language recognised by some finite state machine:

For example:

$$0^n 1^n$$

will give:

$$01$$

$$0011$$

$$000111$$

$$00001111$$

This obviously cannot be stored in a finite state machine as it will require some kind of method of storing the number of 0s on the left, which can be recalled to find the number of 1s on the right.

To show kleenes theorem:

1. Give a regular expression and its accompanying FSM
2. Give a FSM and its accompanying RE

In this unit we will only be dealing with smaller problems.

17 Lecture 18

FSM can be used to effectively complete a simple machine, but there exists other types of autonoma.

17.1 Pumping lemma

The pumping lemma can be defined as:

if L is a language then $\exists$ an integer p called the pumping length of L such that $\forall$ words $w \in L$ where $|w| > p$ $\exists$ expression $w = xyz$ where:

- $\forall_i \geq 0. xy^i z \in L$

- $|y| \geq 1$
- $|xy| \leq p$

If L is regular then any sufficiently long word will contain a non empty substring that can be repeated an arbitrary number of times. Infinite languages can contain an infinite number of characters, and this would require some kind of loops from within the FSM. That is that for any regular language there must exists some loops within the FSM in order for it to be an infinite languge. What the pumping lemma finds is the minimum length before we start repeating portions.

$$Regular \rightarrow Pumping\ lemma$$

This can be used to prove that a language is not a regulalar language, however it obviously cannot be used to prove that a language is indeed a regular language. This is due to the disjunction as apposed to a bijunction.

17.2 Context free languages

Most programming languages are CFLs. CFLs were initially defined in an attempt to understand natural human languages. For example, in english the following normally applies:

$$sentence \rightarrow (noun\ phrase)(verb\ phrase)$$

This is different to a regular expression as an RE typically works with individual characters in the word. The CFL ends up with an overall heirarchy of individual sentences.

Example:

In java a variable name must start with a letter or underscore, followed by any other legal character:

All regular languages are CFLs. CFLs are recognised by **pushdown automata**.

These are expressed in grammar.

Example:

$$Program \rightarrow (some\ rules)$$

Which translates to "a program is comprised of the following rules".

Grammars are defined by a pushdown automata in much the same way that a regular expression is defined by a FSM.

18 Lecture 19

Note that context free languages are easier than pumping lemmas and as such will show up in our assessments more.

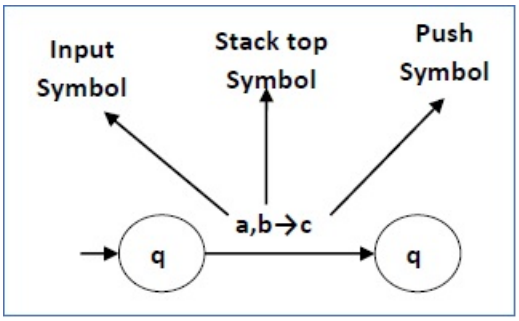
Example:

The language $\{0^n 1^n \mid n \geq 0\}$ is generated by:

$$A \rightarrow 0A1 \mid \epsilon$$

These are called *context free* as A has no knowledge of where it appears in the sequence. The following will always be included in the CFL:

- A grammar grows all the strings of the language



A pushdown automata

- A gramar is a collection of substitution rules called productions
- Each rule has a left hand side symbol, an arrow and a right hand side
- Variable symbols are called *non-terminals* and are usually represented by a capital letter
- Other symbols are from the alphabet of the language called *terminals* are usually represented by a lower case letter
- One symbol is designated the start variable usually written S
- Strings in the language are grown by starting with the start symbol and then replacing non-terminals acording to the production rules.

We can define the context free language using:

- V : a finite set of variables
- Σ : a finite set of terminates.
- R : a finite set of rules
- $S \in V$: the start variable

18.1 Pushdown automata

These are used in a similar way to the FSM with the addition of a stack in which we can store things until we require them. The pushdown automata is defined by:

- Q : the finite set of states
- Σ : a finnite alphabet of input symbols
- Γ : is a finite stack alphabet
- σ : the transition function
- q_0 : the start state
- F : accepting states

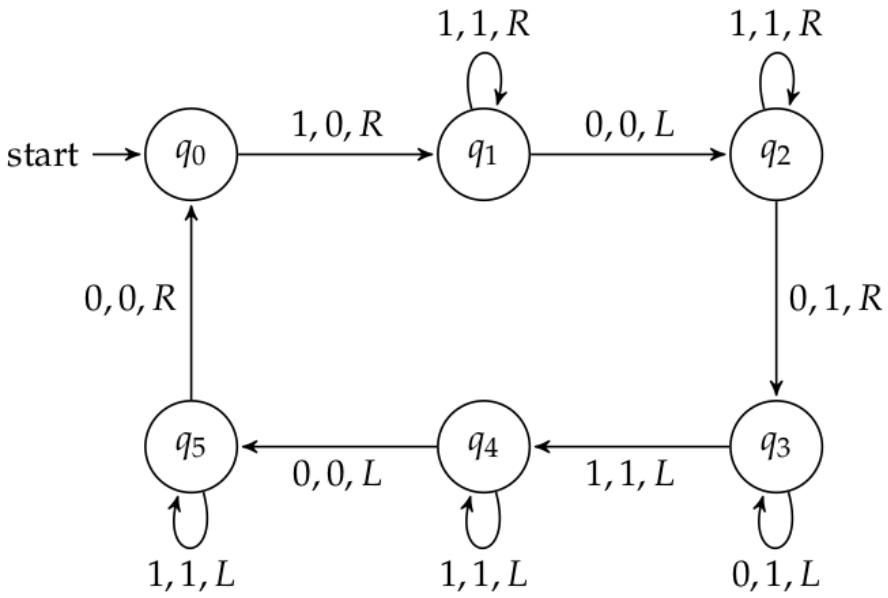
18.1.1 The stack

We can pop or push to the stack. The $ denotes the bottom of the stack.

19 Lecture 20

It is important to figure out the computability and feasability to determine if we are able to:

- See if a program enters an infinite loop
- See if the program contains malware



A turing machine

19.1 Turing machines

The turing machine solves the finite memoryless nature of the FSM. A turing machine contains:

- A finite state machine
- A memory tape of infinite length

The machine will halt when there rae no available outgoing edges on the tape. At each step of the turing machine it will:

- The symbol on the current tape is read
- A symbol is written on the tape cell
- The FSM moves to the the next state
- The read write head is moved one cell left or right

A turing machine is defined by:

- Q : the finite set of states
- Σ : a finnite alphabet of input symbols
- Γ : is a finite stack alphabet
- σ : the transition function
- q_0 : the start state
- F : accepting states
- q_{accept} : The accept states
- q_{reject} : The reject states

We are assuming rejection if there are no valid outgoing states or we end up in an infinite loop. Turing machines can both accept languages and conduct computations, delivering some output.

There is indeed some languages that the turing machine cannot identify. An obvious observation is that the number of turing machines is countable. It is important to note that the number of functions is greater than the number of computable functions (that is that it is uncountable).

20 Lecture 21

It is important to note that not all problems that are computable of any interest. A function that can be computed by some turing machine is turing complete.

Turings description fo computability appears to be the only valid description, as all other attempts to aptly describe this phenomena end up at the same conclusion.

Universal turing machine: A turing machine that can simulate some arbitrary turing machine. This is to say that a universal turing machine can acccept another turing machine as input (typically through the input of a string with the parameters) and can simulate it within itself.

Undecidability: Cannot be answered mechanically. That is that it is impossible to determine an algorithm that can solve the issue - such as the halting problem.

Turing machine model: A TM comprises of a finite state machine plus an input tape of symbols which can be read and written and moved over one square at a time

Halting and looping: A TM halts if it reaches a state and input value for which no rule exists to make a further move. A TM loops by running forever, if it never reaches a halting state.

Specifying a turing machine: Can eb done by encoding the states and rules of the TM as a string of symbols which can then be written on an input tape.

20.0.1 The halting problem

The problem essentially boils down to: does the TM halt on a given input by accepting or rejecting that input:

$$HALT_{TM} = \{(M, w) \mid M \textit{ is a } TM \textit{ and } M \textit{ halts on input } w\}$$

The halting problem is undecidable, in that there can be no algorithm (TM) that can decide whether a program will halt or not, before it has even been computed.