CA320 - Computability & Complexity

Complexity

Dr. David Sinclair

Big-O Notation

A function $g(N)$ is said to be $O(f(N))$ if exists constants c_0 and N_0 such that

$$g(N) < c_0 f(N),\ \forall N > N_0$$

Algorithms can be generally classified as:

- $O(1)$ Run time is independent of the size of the problem, N.
- $O(log(N))$ Occurs when a big problem is solved by transforming it into a smaller size by some constant fraction. (Logarithmic)
- $O(N)$ Occurs when each element of the problem requires a small amount of processing. (Linear)

Big-O Notation (2)

$O(Nlog(N))$ Occurs when a problem is broken into smaller subproblems, solving them independently, and combining the solutions. (Linearithmic)

$O(N^2)$ Occurs when an algorithm processes all pairs of elements. (Quadratic)

$O(2^N)$ Exponential run time. To be avoided. Characteristic of "brute force" approaches. (Exponential).

If each step takes 1ms (10^{-3}s) then:

N	$O(log(N))$	$O(\sqrt{N})$	$O(N)$	$O(Nlog(N))$	$O(N^2)$	$O(2^N)$
100	6.64ms	10ms	0.1s	0.2s	10s	$4x10^{19}$ years
200	7.64ms	14.1ms	0.2s	0.46s	40s	$5.1x10^{48}$ years

The age of the universe is about $14x10^9$ years!

Complexity Measures

There are 2 different kinds of complexity measures:

- *static measures* (program size) that are based on the structure of the algorithm, and
- *dynamic measures* (time and space) that are based on the algorithm and the inputs and hence are based on the behaviour of the computation.

Dynamic complexity measures are more descriptive and therefore are the interesting ones. A model-independent dynamic complexity measure Φ with respect to a set of algorithms $\{P_i\}$ should have the following properties.

- For each i, the domain of Φ_i coincides with that of P_i, and its codomain is a subset of N.
- A total computable predicate M exists such that

$$M(i, x, m) = true \Leftrightarrow \Phi_i(x) = m, \ \forall i, x, m$$

Complexity Measures (2)

TIME

Using the Turing machine as the model of computation, let TIME $= \{TIME_i\}$, where $TIME_i(x)$ denotes the number of steps executed by computation $T_i(x)$. If $T_i(x)$ does not halt, then $TIME_i(x)$ is considered to be undefined.

SPACE

SPACE must be defined carefully so that $SPACE_i(x)$ is undefined whenever $Ti(x)$ does not halt. If $T_i(x)$ does halt, after reading at most h(x) cells on any of the k tapes, it will halt in $N = |\Sigma|^{h(x)k} h(x)^k |Q|$ steps at most. N is the number of distinct global states of Ti(x). (An upper bound on space implies an upper bound on time.)

Nondeterministic TIME and SPACE

For nondeterministic Turing machine the time measure is $NTIME_i$, where $NTIME_i(x)$ is:

1. The number of steps in the shortest accepting computation path of $NT_i(x)$, if one exists and all paths halt.
2. The number of steps in the shortest computation path of $NT_i(x)$, if all paths halt and reject.
3. Undefined, otherwise.

NSPACE is similarly defined.

Classes of Languages

Languages can be grouped into *classes*. A class C is the set of all languages L over alphabet Σ that satisfy a predicate π.

$$C = \{L | L \subseteq \Sigma^* \wedge \pi(L)\}$$

The complement of a class C, coC, is defined as

$$coC = \{L | L \subseteq \Sigma^* \wedge \pi(L^c)\}$$

Example

The class T of tally languages is defined as $C = \{L | L \subseteq \Sigma^* \wedge \pi(L)\}$ where $\Sigma = \{0, 1\}$ and $\pi(L) = true \Leftrightarrow L \subseteq 0^*$. What is coT?

Resource-bounded Reducibility

A generalisation of reducibility is resource-bounded reducibility, $\leq_r$, where the reducibility function f can be computed in some predefined resource bounds. Given a class of languages C, and language L is said to be C-complete with respect to $\leq_r$ if $L \in C \wedge L' \leq_r L, \forall L' \in C$.

A class C is said to closed with respect to reducibility $\leq_r$ if for any pair of languages L_1 and L_2 such that $L_1 \leq_r L_2, L_2 \in C \Rightarrow L_1 \in C$.

Lemma

Let C_1 and C_2 be two classes of languages such that $C_1 \subset C_2$, and C_1 is closed with respect to a reducibility $\leq_r$. Then any language L that is C_2-complete with respect to $\leq_r$ does not belong to C_1.

The Class **P**

If each step of an algorithm requires only 10^{-9}s to execute then:

size n	n^2	n^3	2^n	3^n
10	0.1 μs	1 μs	1μs	59 μs
30	0.9 μs	27 μs	1s	2.4 days
50	2.5 μs	0.125ms	13 days	$2.3x10^7$ years

Therefore for practical reasons we consider all problems solved by algorithms that require a hyperpolynomial ($f(x) \geq n^k, \forall k$) number of steps to be computationally intractable.

The set of tractable problems belongs to the set of problems whose language is accepted by some deterministic Turing machine in a polynomial-bounded number of steps. Or formally, the languages that encode tractable problems belong to the class **P**, which is defined as

$$\mathrm{P} = \bigcup_{k \geq 0} DTIME[n^k]$$

Examples of Class **P** Languages

Greatest Common Divisor (GCD)

The greatest common divisor (GCD) of two integers a and b is defined to be that largest integer that divides both a and b. We can derive an algorithm for GCD by observing that if r is the remainder of a divided by b ($a \geq b$), then the common divisors of a and b coincide with the common divisors of b and r.

```
function GCD (a:integer, b:integer):integer
begin
   if b = 0 then GCD = a else GCD = GCD (b, a mod b)
end
```

Complexity analysis : Since $a_k = q * b_k + b_{k+1}$ and $b_{k-1} = a_k$ we have $b_{k-1} > b_k + b_{k+1} > 2b_k$.
Then $b = b_0 > 2b_1 > 4b_2 > 8b_3 > ... > 2^k$
and thus $k < \log b$.
Therefore the time complexity of GCD is linear with respect to the size of the input length.

Examples of Class **P** Languages (2)

Shortest Path

The Shortest Path Problem consists of determining whether a path of length of at most k exists between a pair of nodes n_1 and n_2 of an input graph G.

The following algorithm makes use of a dynamic programming technique. Denote by n the number of node of G and let $A^h(i,j)$ be the length of the shortest path from node i to node j going through no node with an index greater than h. Then,

$$A^{h+1}(i,j) = \min(A^{h+1}(i,h+1) + A^{h+1}(h+1,j), A^{h+1}(i,j))$$

that is, either a shortest path from i to j going through no node of index greater than $h+1$ passes through node $h+1$ or it does not. By construction, the entry $A^n(n_1,n_2)$ yields the value of the shortest path between n_1 and n_2. If the value is not greater than k, then the SHORTEST PATH instance admits a yes answer.

Examples of Class **P** Languages (3)

begin {input: G, n_1, n_2, k}
  n = number of nodes of G;
  for all $i, j \leq n$ do
    if G includes an edge $\langle i,j \rangle$
      then $A^1(i,j) = 1$
      else $A^1(i,j) = \infty$;
  for $h = 2$ to n do
    for $i \leq n$ do
      for $j \leq n$ do
        $A^h(i,j) = \min(A^{h-1}(i,h) + A^{h-1}(h,j), A^{h-1}(i,j))$;
  if $A^n(n_1,n_2) \leq k$ then accept else reject;
end

Because of the nested for loops, clearly the number of steps required by the algorithm is $O(n^3)$.

Examples of Class **P** Languages (4)

2-Satisfiability Problem

Given a set of boolean variables $x_1, x_2, \ldots, x_n$, a literal is either one of the variables x_i or the negation of the variable, $\overline{x_i}$. A *clause* is a disjunction (logical or) of a subset of the boolean variables. Each boolean variable is assigned a truth value, true or false. A clause is true is one of the literals in the clause is true otherwise the clause is false. A set of clauses is satisfiable is there exists an assignment of truth values to the variables that makes all the clauses true.

The satisfiability problem (SAT): Given a set of clauses, does there exist a set of truth values, one for each variable, such that every clause is satisfiabe?

The 2-Satisfiability problem (2-SAT): Given a Boolean formula f that is a conjunction of clauses containing exactly 2 literals, does an assignment of truth values satisfying f exist?

Examples of Class **P** Languages (5)

Algorithm:

The objective is to guess the value of an arbitrary variable and deduce the consequences of this guess for the other variables of f.

Initially all clauses are declared unsatisfied. A starting variable x is selected arbitrarily with x and $\overline{x}$ receiving the values true and false respectively. The assignment is then extended to as many variables as possible by repeated applications of the following step.

Take an arbitrary unsatisfied clause $(l_h \vee l_k)$. If one of the two literals has the value true then declare the clause satisfied. Otherwise if one of the literals, say l_h, has the value false, then assign to l_k and $\overline{l_k}$ the values true and false respectively. Declare the clause $(l_h \vee l_k)$ satisfied.

Examples of Class **P** Languages (6)

3 exclusive cases may occur:

1. During the execution of any of the steps, a conflict takes place were a literal is assigned the value true which was previously assigned false. This means the initial guess of the literal value was wrong and all the steps starting from the assignment of the values true to x are cancelled. This time x and $\overline{x}$ receive the values false and true respectively and the assignment procedure starts again. If a second conflict occurs, the algorithm terminates and f is declared unsatisfied.
2. No conflict occurs and all variables receive a value. Then the formula f is satisfiable.
3. No conflict occurs but some variables remain unassigned. We may ignore the clauses already satisfied. Clearly the reduced formula is satisfiable if and only if the previous one is also satisfiable. A guess is then made for an arbitrary variable and the assignment procedure is again applied to the reduced formula.

Examples of Class **P** Languages (7)

```
begin {input: f}
  C = set of clauses of f ; declare the clauses of C unsatisfiable;
  V = set of variables of f ; declare the variables of V unassigned;
  while V contains a variable x do
    begin
      assign true to x; firstguess = true;
      while C contains an unsatisfied clause
        c = (l_h ∨ l_k) with at least one assigned literal do
          begin
            if l_h = true ∨ l_k = true then declare c satisfied;
            else if l_h = false ∧ l_k = false then
              begin
                if not firstguess then reject
                else
                  begin
                    declare the clauses of C unsatisfied and the variables of V unassigned;
                    assign false to x; firstguess = false;
                  end
              end
            else if l_h = false then assign true to l_k else assign false to l_k;
          end
            delete from C the satisfied clauses and V the assigned variables;
    end
  accept;
end
```

Polynomial-time Intractable Problems

The vast majority of problems that belong to the class **P** are computationally tractable, but there problems in **P** which are not tractable.

Problems in **P** are bounded by cn^k steps. If, for a given problem, c and k are very large then the problem is practically intractable.

Also there exists a small set of problems for which we can prove that the problem can be solved in a polynomial number of steps, but we can not describe the Turing machine which decides its language.

Polynomial-time Reducibility

How do we prove that an algorithm belongs to **P**? One way is to count the number of steps in the algorithm; another is to use the concept of polynomial reducibility.

Given two languages L_1 and L_2, L_1 is polynomially reducible to L_2, $L_1 \leq L_2$, if a function $f \in \mathbf{FP}$ exists such that $x \in L_1 \leftrightarrow f(x) \in L_2$.

Lemma

Let L_1 and L_2 be two languages such that $L_1 \leq L_2$ and $L2 \in \mathbf{P}$. Then $L_1 \in P$.

Proof.

Add a Turing transducer that computes the reduction to a Turing acceptor that decide L_2. The composition of two polynomials is still a polynomial. □

Polynomial-time Reducibility (2)

Lemma

Let B be any language included in **P** *such that $B \neq \emptyset$ and $B \neq \Sigma^*$. Then, for any language A included in* **P**, $A \leq B$.

Proof.

Let y be a word in B and z be a word in B^c. Then define a function f as

$$f(x) = \begin{cases} y, & \text{if } x \in A \\ z, & \text{otherwise} \end{cases}$$

The function f is polynomial-time computable since A is polynomial-time decidable, and $x \in A \leftrightarrow f(x) \in B$ is the definition of polynomial reducibility. □

The class **P** is closed with respect to reducibility.

The Class **NP**

By replacing the deterministic Turing machines by nondeterministic Turing machine, we can define another class of languages (and hence problems). The class **NP** is defined as:

$$\text{NP} = \bigcup_{k \geq 0} NTIME(n^k), k \geq 0$$

Since deterministic Turing machines are a restriction of nondeterministic Turing machines, $P \subseteq NP$. General consensus has it that **NP** is richer than **P**, because nondeterministic Turing machines are more "powerful" than deterministic ones since they execute several computation paths at the same time. A nondeterministic polynomial-time algorithm can be thought of as an algorithm that always chooses the correct alternative from many and runs in polynomial time.

The Class **NP** (2)

Is a nondeterministic Turing machine "more powerful" than a deterministic Turing machine?

Theorem

Given any k-tape Turing machine T with $k > 1$, it is always possible to derive an equivalent 1-tape Turing machine T that simulates the first t steps of T in $O(t^2)$ steps.

Proof.

We dedicate k consecutive cells of the single tape of T to storing the set of symbols contained in the i-th cells ($i = 0, 1, \ldots$) of the k tapes of T.
The k tape heads of T are simulated by making use of suitable marked symbols. For example, if the h-th tape head of T scanned a symbol a then the tape of T will contain the marked symbol a_h.
By convention, after a quintuple of T has been simulated the tape head of T is assumed to be positioned on the leftmost marked symbol.

The Class **NP** (3)

Proof (contd.)

The simulation proceeds as follows. First the k marked symbols are scanned sequentially rightwards to determine the quintuple of T to be simulated. Next, the tape is scanned k times rightwards to replace the old symbols with the new ones and to simulate the k tape moves. Assume that a left (or right) move must be simulated relative to tape h. This is accomplished by shifting the tape head k positions to the left (or right) and by replacing the symbol currently scanned, say a, with a marked symbol a_h. Since the tape is "semi-infinite" we need to be careful moving to the left. To prevent running past the left end of the tape we use a special end of tape symbol, say [, so that whenever to be marked is a [, T shifts back k positions to the right. A left move from cell 0 is replaced by a not moving the tape head.

The Class **NP** (4)

Proof (contd.)

The following configurations illustrate an example with 2 tapes where the tape heads are positioned on symbols c and f. The quintuple to be simulated replaces c with x, replaces f with y and performs a left move on tape 1 and a right move on tape 2.

	Before	After
tape 1	$[abq_icd\Delta$	$[abxq_jd\Delta$
tape 2	$[eq_ifgh\Delta$	$[q_jeygh\Delta$

The single tape simulation of this machine has the following configurations.

Before	After
$[[aebq_rf_2c_1gdh\Delta\Delta$	$[[aq_se_2byxgd_1h\Delta\Delta$

□

The Class **NP** (5)

Theorem

Given a nondeterministic Turing machine NT, it is always possible to derive an equivalent deterministic machine T. Furthermore, $\exists c$ such that if NT accepts x in t steps, then T will accept x in, at most, c^t steps.

Proof.

For any input x the aim is to systematically visit the computation tree associated with $NT(x)$ until T finds an accepting computation path (if it exists).

T needs to be careful since the computation tree may include computation paths of an infinite length. A simple depth-first search of the tree might end up following an infinite path and never get around to considering a finite accepting path.

The Class **NP** (6)

Proof (contd.)

The solution to this problem is to use a breadth-first search. T visits all the computation paths for, at most, one step and if an accepting state is encountered then T accepts x. Otherwise we visit all the computation paths for, at most, two steps, and so on. If NT accepts x then T will eventually accept x.
We still need to specify how to visit all the computation paths for, at most, i steps, for $i = 1, 2, \ldots$. Since we are now dealing with finite trees we can do a depth-first search. Since r is the degree of nondeterminism of NT then there are at most r^i partial computation paths. With each path we can associate a word with maximum length i over the alphabet $\Gamma = 1, 2, \ldots, r$. Let Γ^i represent all the words of length i over Γ. A deterministic Turing machine can generate all such words, in lexicographical order, in a linear number of operations with respect to i.

The Class **NP** (7)

Proof (contd.)

Then starting from the initial state T can simulate the execution path, pw, associated with each word w. Three situations can occur:

1. If pw does not represent a valid sequence of steps of NT then T moves to the next word in Γ^i.
2. If the first j symbols of $w, (0 \leq j \leq i)$, induce a sequence of steps that cause NT to accept x, then T also accepts x.
3. If the first j symbols of $w, (0 \leq j \leq i)$, induce a sequence of steps that cause NT to reject x, then T moves onto the next word in Γ^i.

The Class **NP** (8)

Proof (contd.)

Simulating *pw* also requires a linear number of steps with respect to i. Thus over all computation paths i steps can be visited in at most r^i steps where r is the degree of nondeterminism. Therefore if NT accepts x in t steps, then the total time spent by T in simulating NT is bounded by

$$\sum_{i=1}^{t} r^i \leq c^t$$

where c is a suitable constant.

□

Therefore a nondeterministic Turing machine doesn't "solve any more problems" than a deterministic Turing machine, it just does it more efficiently.

Checking solutions in **NP**

Theorem

A language L belongs to **NP** *if and only if a language* $L_{check} \in$ P *and a polynomial p exist such that*

$$L = \{x | \exists y . x, y \in L_{check} \wedge y \leq p(x)\}$$

Proof.

If $L = \{x | \exists y . x, y \in L_{check} \wedge y \leq p(x)\}$ where $L_{check} \in P$ and p is a polynomial, then the following nondeterministic algorithm decides L in polynomial time.

```
begin
   guess y in the set of words of length, at most, p(|x|);
   if < x, y >∈ L_check then accept;
     else reject;
end
```

Checking solutions in **NP** (2)

Proof (contd.)

Conversely, let L be a language in **NP**. Then a nondeterministic Turing machine NT exists that decides L in polynomial time. It is easy to verify that, for any x, each computation path of $NT(x)$ can be encoded into a word of length, at most, $p(|x|)$ where p is a polynomial. The language L_{check} is then defined as follows.
$\langle x, y \rangle \in L_{check}$ if and only if y encodes an accepting computation path of $NT(x)$.
Therefore $L_{check} \in$ P and for any x

$$x \in L \leftrightarrow \exists y.x, y \in L_{check} \wedge y \leq p(x)$$

□

This theorem tells us that **NP** is the class of problems for which it is easy to check the correctness of a claimed answer. We are not asking for a way to find a solution, only to verify that an alleged solution is correct.

Examples of Class **NP** Languages

Satisfiability Problem

Satisfiability is a generalisation of 2-Satisfiability introduced earlier. Here each clause C_i may include any number of literals rather than exactly two as in 2-SAT.

Solution: A nondeterministic algorithm for Satisfiability can be obtained by guessing any of the $2n$ assignments of value to the n variables of f and verifying whether it satisfies f:

```
begin {input: f }
   guess t in the set of assignments of values
                to the n variables of f;
   if t satisfies f then accept;
     else reject;
end
```

Examples of Class **NP** Languages (2)

Traveling Salesman Problem (TSP)

Given a completed weighted graph G and a natural number k, does a cycle exist passing through all the nodes of G such that the sum of the weights associated with the edges of the cycle is, at most, k?

Solution: A polynomial-time nondeterministic algorithm for the Traveling Salesman Problem (TSP) can be obtained by guessing any of the n! permutations of n nodes and verifying whether it corresponds to a cycle whose cost is, at most, k.

Examples of Class **NP** Languages (3)

3-Colourability Problem

Given an undirected graph G does there exists a way to color the nodes red, green, and blue so that no adjacent nodes have the same color?

Solution: A polynomial nondeterministic algorithm for 3-Colourability can be obtained by guessing any of the 3^n colourings of a graph of n nodes and checking if it has the required property.

Examples of Class **NP** Languages (4)

We can reduce the 3-Colourability problem to the Satisfiability problem. Consider a graph G of four vertices, and the decision problem "Can the vertices of G be coloured in 3 colours such that no edge connects vertices of the same colour?".

We will use 12 Boolean variables. The variable $x_{i,j}$ corresponds to the assertion that vertex i has been coloured in the colour j $(i = 1, 2, 3, 4; j = 1, 2, 3)$.

We can construct the 3-Colourability problem as an instance of the Satisfiability problem with 34 clauses. The first 16 clauses are:

$$\begin{aligned} C(i) &= \{x_{i,1}, x_{i,2}, x_{i,3}\} \\ T(i) &= \{\overline{x}_{i,1}, \overline{x}_{i,2}\} \\ U(i) &= \{\overline{x}_{i,1}, \overline{x}_{i,3}\} \\ V(i) &= \{\overline{x}_{i,2}, \overline{x}_{i,3}\} \end{aligned}$$

Examples of Class **NP** Languages (5)

The 4 clauses $C(i)$ assert that each node has been coloured in at least one colour (though possibly more). The clauses $T(i)$ state that no node has colour 1 and colour 2. Similarly the clauses $U(i)$ state that no node has colour 1 and colour 3, and the clauses $V(i)$ state that no node has colour 2 and colour 3. Taken all together these 16 clauses state that each node has been coloured by one and only one colour.

The remaining clauses ensure that the two endpoints of an edge do not have the same colour. We define for each edge e of the graph G and colour $j(= 1, 2, 3)$ a clause $D(e, j)$ as follows. Let u and v be the endpoints of e, then

$$D(e, j) = \{x_{u,j}, x_{v,j}\}$$

asserts that not both endpoints of an edge have the same colour.

Examples of Class **NP** Languages (6)

The original instance of the 3-Colourability problem has now been reduced to an instance of the Satisfiability problem. Specifically, does there exist an assignment of value true and false to the 12 Boolean variables $x_{1,1}, ..., x_{4,3}$ such that each of the 34 clauses contains at least one literal whose value is true if and only if each pair of nodes connected by an edge of the graph G have different colours? The graph is 3-Colourable if and only if the clauses are satisfied.

If we have an algorithm to solve the Satisfiability problem, then we can also solve the 3-Colourability problem.

NP-complete Languages

Are all problems in NP equally hard, or are some more difficult than others? Are there problems in NP which are the most difficult?

Completeness Revisited

Given a complexity class C and a language L, for any language $L \in C$ we derive a reduction f which, given a model of computation M which decides L and an input x, computes a word $f(M, x)$ such that M accepts x if and only if $f(M, x) \in L$.
L is reducible to L and since L is an arbitrary language in C, L is C-complete.

Informally, we say that a decision problem in **NP**-complete if it belongs to **NP** and every problem in **NP** is quickly reducible to it. Or formally, is there a class of languages in **NP** which can be polynomially-time reduced to any other language in **NP**?

Implications of **NP**-completeness

Suppose we could prove that a certain decision problem Q is **NP**-complete. Then we could concentrate our efforts on finding a polynomial-time algorithm for Q. If we succeed in finding a polynomial-time algorithm for all instances of Q then we could derive a fast algorithm for every problem in **NP**.

Conversely, if we could prove that it is impossible to find a polynomial-time algorithm for some particular problem R in **NP**, then we cannot find a fast algorithm for any **NP**-complete problem either.

Is there any **NP**-complete problems? Stephen Cook and Leonid Levin independently proved that *Satisfiability* was **NP**-complete in 1971 and 1973 respectively.

Cook-Levin Theorem

Theorem

Satisfiability *is NP-complete.*

Proof.

We want to prove that *Satisfiability* is **NP**-complete, i.e. that every problem in **NP** is polynomially reducible to an instance of Satisfiability.

Let Q be some problem in **NP** and let I be an instance of problem Q. Since Q is in **NP** there exists a nondeterministic Turing machine that recognises encoded instances of the problem Q in polynomial time.

Let TMQ be such a Turing machine and let $P(n)$ be a polynomial in its argument n with the property that TMQ recognises every input x in time $P(n)$, where x is a word in the language Q and n is the length of x.

Cook-Levin Theorem (2)

Proof (contd.)

We intend to construct an instance $f(I)$ of Satisfiability (SAT), corresponding to each word I in the language Q, for which the answer to the decision problem "are all the clauses simultaneously satisfiable" is Yes. Conversely, if the word I is not in the language Q, the clauses will not be satisfiable.

To construct an instance of SAT means that we are going to define a number of variables, literals and clauses in such a way that the clauses are satisfiable if and only if x is in the language Q, i.e. the machine TMQ accepts x.

What we must do then is to express the accepting computation of the nondeterministic Turing machine as the simultaneous satisfaction of a number of logical propositions. It is precisely here that the relatively simplicity of a Turing machine allows us to enumerate all of the possible paths to an accepting computation.

Cook-Levin Theorem (3)

Proof (contd.)

Now we will describe the Boolean variables that will be used in the clauses under construction.

$Q_{i,k}$ is true if after step i of the checking calculation it is *true* that the Turing machine TMQ is in state q_k, and *false* otherwise.

$S_{i,j,a} = \{\text{after step } i, \text{ symbol } a \text{ is in tape square } j\}$.

$T_{i,j} = \{\text{after step } i, \text{ the tape head is positioned over square } j\}$.

Let's count the variables that we've just introduced. Since the Turing machine TMQ does its accepting calculation in time $\leq P(n)$, it follows that the tape head will never venture more than $P(n)$ squares away from its starting position. Therefore the subscript j, which runs through the various tape squares that are scanned during the computation, can assume only $O(P(n))$ different values.

Cook-Levin Theorem (4)

Proof (contd.)

Index a runs over the letters in the alphabet that the machine can read, so it can assume at most some fixed number A of values.

The index i runs over the steps of the accepting computation, and so it takes at most $O(P(n))$ different values.

Finally, k indexes the states of the Turing machine, and there is only some fixed finite number, say K, of states that TMQ might be in. Hence there are altogether $O(P(n)^2)$ variables, a polynomial number of them.

The remaining task is to described precisely the conditions under which a set of values assigned to the variables listed above actually defines a possible accepting calculation for x . Then when set of satisfying values of the variables is found they will determine a real accepting calculation of the machine TMQ.

Cook-Levin Theorem (5)

Proof (contd.)

This will be done by requiring that the number of clauses be all true (satisfied) at once, where each clause will express one necessary condition.

1. At each step the machine is in at least one state.

Hence at least one of the K available state variables must be true. This leads to the first set of clauses, one for each step i of the computation.

$$\{Q_{i,1}, Q_{i,2}, \ldots, Q_{i,K}\}$$

Since i assumes $O(P(n))$ values, there are $O(P(n))$ clauses.

Cook-Levin Theorem (6)

Proof (contd.)

2. At each step the machine is not in more than one state.

Therefore, for each step i, and each pair (j', j'') of distinct states, the clauses

$$\{\overline{Q}_{i,j'}, \overline{Q}_{i,j''}\}$$

must be true. These are $O(P(n))$ additional clauses to add to the list.

Cook-Levin Theorem (7)

Proof (contd.)

3. At each step, each tape square contains exactly one symbol from the alphabet of the machine.

This leads to two lists of clauses which require:

1. That there is at least one symbol in each square at each step.
2. That there are not two symbols in each square at each step.

The clauses that do this are:

$$\{S_{i,j,1}, S_{i,j,2}, \ldots, S_{i,j,A}\}$$

where A is the number of letters in the machine's alphabet, and

$$\{\overline{S}_{i,j,k'}, \overline{S}_{i,j,k''}\}$$

for each step i, square j and pair (k', k'') of distinct symbols in the alphabet of the machine. This gives rise to $O(P(n)^2)$ additional clauses.

Cook-Levin Theorem (8)

Proof (contd.)

4. At each step the tape head is positioned over a single square.

First, for each step, the tape head is positioned on at least one square:

$$\{T_{i,1}, T_{i,2}, \ldots, T_{i,P(n)}\}$$

Second, for each step, the tape head is not positioned on two or more squares.

$$\{\overline{T}_{i,j'}, \overline{T}_{i,j''}\}$$

This gives another $O(P(n)^2) + O(P(n)^3) = O(P(n)^3)$ clauses.

Cook-Levin Theorem (9)

Proof (contd.)

5. Initially the machine in in state 0, the tape head is over square 1 and the input string x is in squares 1 to n.

$$\{Q_{1,0}\}$$
$$\{T_{1,1}\}$$
$$\{S_{1,1,x_1}\}$$
$$\{S_{1,2,x_2}\}$$
$$\ldots$$
$$\{S_{1,n,x_n}\}$$

Cook-Levin Theorem (10)

Proof (contd.)

6. At step $\leq P(n)$ the machine is in state q_Y.

$$\{Q_{1,q_Y}, Q_{2,q_Y}, \ldots, Q_{P(n),q_Y}\}$$

7. At each step the machine moves to its next global state (state, symbol, head position) in accordance with the application of its program module to its previous (state, symbol).

To find the clauses that will do this job consider first the following condition: the symbol in square j of the tape cannot change during step i of the computation if the tape head is not positioned on it at that moment.

Cook-Levin Theorem (11)

Proof (contd.)

This translates into the collection:

$$\{T_{i,j}, \overline{S}_{i,j,k}, S_{i+1,j,k}\}$$

of clauses, one for each triple (i, j, k) = (state, square, symbol). These clause express the condition that, at time i,

- either the tape head is positioned over square j ($T_{i,j}$ is *true*)
- or else the head is not positioned there, in which case
  - either symbol k is not in the jth square before the step
  - or else symbol k is (still) in the jth square after the step is executed.

Cook-Levin Theorem (12)

Proof (contd.)

It remains to express the fact that transitions from one global state of the machine to the next are the direct results of the operation of the program module, The three sets of clauses that do this are:

$$\{\overline{T}_{i,}, \overline{Q}_{i,k}, \overline{S}_{i,j,l}, T_{i+1,j+INC}\}$$
$$\{\overline{T}_{i,j}, \overline{Q}_{i,k}, \overline{S}_{i,j,l}, Q_{i+1,k'}\}$$
$$\{\overline{T}_{i,j}, \overline{Q}_{i,k}, \overline{S}_{i,j,l}, S_{i+1,j,l'}\}$$

Each clause can be interpreted:

"either the tape head is not positioned on square j, or the present state is not q_k or the symbol just read is not l, but if they are then ..."

Cook-Levin Theorem (13)

Proof (contd.)

There is a clause (as above) for each step $i = 0, \ldots, P(n)$ of the computation, for each square $j = 1, \ldots, P(n)$ of the tape, for each symbol l in the alphabet, and for each possible state q_k of the machine. In total a polynomial number of clauses. The new global state triple (k, l, INC) is as computed by the program.

Now we have constructed a set of clauses with the following property. If we execute a recognising computation on a string x, in at most $P(n)$ time, then this computation determines a set of (*true*, *false*) values for all the variables listed previously, in such a way that all of the clauses just constructed are simultaneously satisfied.

Cook-Levin Theorem (14)

Proof (contd.)

Conversely, if we have a set of values of the SAT variables that satisfy all of the clauses at once, then the set of values of the variables that would cause *TMQ* to do a computation that would recognise the string x. It also describes, in minute detail, the ensuing accepting computation that *TMQ* would do if it were given x.

Hence every language in **NP** can be reduced to SAT. It is not difficult to check through the above construction and prove that the reduction is accomplished in polynomial time. It follows that SAT is **NP**-complete.

□

P $\neq$ **NP**?

The fact that $P \subseteq NP$ is without doubt, but the really interesting question is:

Is P properly included in NP?

This $P \neq NP$ conjecture was first raised nearly 50 years ago and still not answered. We know that any nondeterministic Turing machine can be transformed into a deterministic Turing machine, but this machine will require an exponential number of steps. A major leap forward would be a proof, or disproof, of this conjecture.

$\mathbf{P} \neq \mathbf{NP}$? (2)

Reasons why we believe the $P \neq NP$ conjecture.

- No one has been able to find a polynomial-time algorithm for any of the over 3000 **NP**-complete problems
- Complexity theorists have developed a rich hierarchy of complexity classes, many of which would collapse together if $\mathbf{P} = \mathbf{NP}$
- There would be "no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss..." (Scott Anderson, MIT)

However, as yet, there is no proof that $\mathbf{P} \neq \mathbf{NP}$.

CA320 - Computability & Complexity

Context-Free Languages

Dr. David Sinclair

Context-Free Grammars

A *context-free grammar* (CFG) is a 4-tuple $G = (N, \Sigma, S, P)$ where

- N is a set of *nonterminal symbols*, or *variables*,
- Σ is a set of *terminal symbols*, and N and Σ are disjoint finite sets,
- S is a special nonterminal ($S \in N$) called the *start symbol*,
- P is a finite set of *grammar rules*, or *productions*, of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (N \cup \Sigma)$.

The set $V = N \cup \Sigma$ is called the vocabulary of G.

Context-Free Grammars (2)

A *derivation*, $\alpha \Rightarrow \beta$, is the application of one or more production rules starting with the string α and resulting in the string β.

Consider the following grammar.

$$\begin{aligned} S &\rightarrow SS \\ S &\rightarrow (S) \\ S &\rightarrow \epsilon \end{aligned}$$

An example derivation is:

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S((S)) \Rightarrow S(()) \Rightarrow (S)(()) \Rightarrow ()(())$$

If $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a *single step derivation* using $A \rightarrow \gamma$.

The grammar is a *context-free grammar* since the production rule, $A \rightarrow \gamma$, does not depend on the *context* surrounding nonterminal, A.

Context-Free Grammars (3)

Derivations requiring ≥ 0 and ≥ 1 steps are denoted by $\Rightarrow^*$ and $\Rightarrow^+$ respectively.

The *start symbol* denotes the entire set of strings that can be generated by G, $L(G)$.

$$L(G) = \{x \in \Sigma^* | S \Rightarrow^+ x\}$$

Example: *AnBn*

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow \epsilon \end{aligned}$$

This grammar yields derivations such as
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$.

$L(G) = \{a^k b^k | k \geq 0\}$

Context-Free Grammars (4)

Example: *Expr*

$$
\begin{array}{lcl}
S & \rightarrow & E\ O\ E \\
E & \rightarrow & id \\
E & \rightarrow & num \\
E & \rightarrow & (\ E\ ) \\
O & \rightarrow & + \\
O & \rightarrow & - \\
O & \rightarrow & * \\
O & \rightarrow & /
\end{array}
$$

This grammar generates simple expressions over number and identifiers.

Derivations

A *derivation* is a sequence of steps where in each step a non-terminal is replaced by the left-hand side of a productions rule that starts with the non-terminal.

If the leftmost non-terminal is always chosen to be replaced then it is a *leftmost derivation*.

If the rightmost non-terminal is always chosen to be replaced then it is a *rightmost derivation*.

A derivation is also called a *parse*. The process of discovering a derivation is called *parsing*.

Parse Tree

Consider the CFG grammar *Expr*.

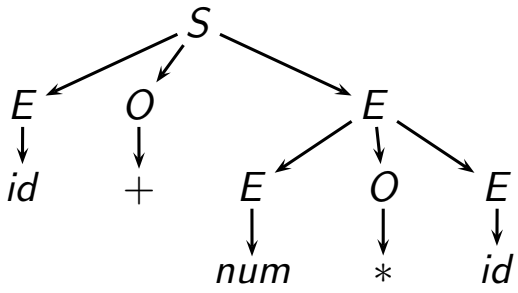
$$
\begin{aligned}
S &\Rightarrow E\ O\ E \\
&\Rightarrow id\ O\ E \\
&\Rightarrow id\ +\ E \\
&\Rightarrow id\ +\ E\ O\ E \\
&\Rightarrow id\ +\ num\ O\ E \\
&\Rightarrow id\ +\ num\ *\ E \\
&\Rightarrow id\ +\ num\ *\ id
\end{aligned}
$$

This can be denoted as $S \Rightarrow^* id + num * id$.

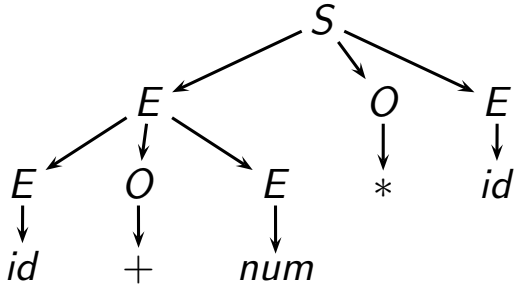
Because we always choose the leftmost non-terminal this is a leftmost derivation.

Parse Tree (2)

This derivation can also be represented as a *parse tree*.

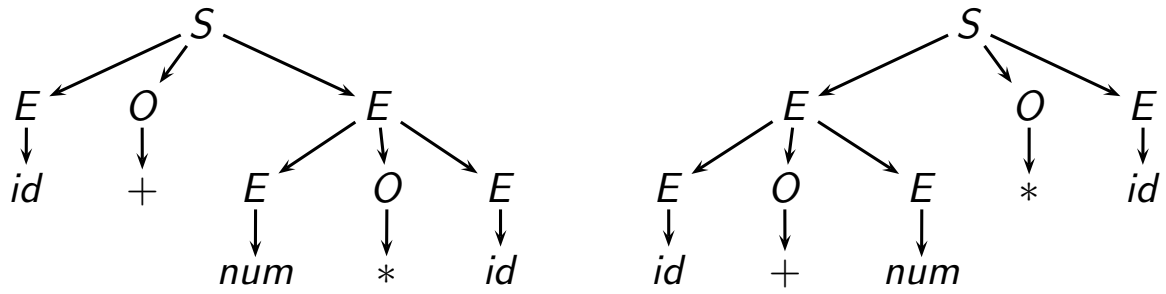


But what if we choose a rightmost derivation of the same expression.



Ambiguity

What is the implication of the fact that there at least 2 parse trees for the same expression?



A CFG G is *ambiguous* if $\exists x \in L(G)$ such that x has more than one derivation tree. This is equivalent to saying it has more than one distinct leftmost or rightmost derivations tree.

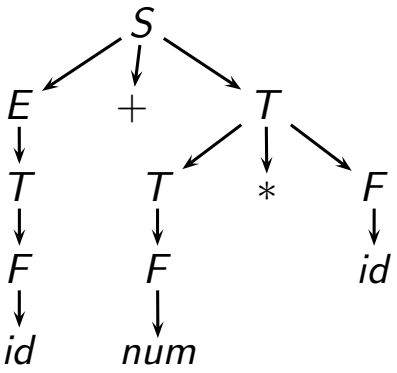
Ambiguity (2)

Sometimes redesigning the grammar can remove the ambiguity. The following grammar, *Expr1*, does not have the ambiguity of *Expr*.

Expr1

$$
\begin{array}{lcl}
S & \rightarrow & E \\
E & \rightarrow & E + T \\
E & \rightarrow & E - T \\
E & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & T / F \\
T & \rightarrow & F \\
F & \rightarrow & id \\
F & \rightarrow & num \\
F & \rightarrow & (E)
\end{array}
$$

The only parse tree for $id + num * id$ is:



Pushdown Automaton

We know that the language *AnBn* is not a regular language and cannot be recognised by a *Finite State Automaton*. The following language $SimplPal = \{wcw^r | w \in \Sigma^*\}$, where w^r is the reverse of w, is also not a regular language. Both of these languages require a machine that can remember something. In the case of *SimpPal* it must remember w. Then after seeing c it then checks for w^r.

We can extend a Nondeterministic Finite State Automaton by adding a *stack memory*. This is called a *Pushdown Automaton*.

Pushdown Automaton (2)

A *Pushdown Automaton* (PDA) is a 7-tuple $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ where

- Q is a finite set of *states*;
- Σ and Γ are finite *input* and *stack alphabets*;
- $q_0 \in Q$ is the *initial state*;
- $Z_0 \in \Gamma$ is the *initial stack symbol*;
- $A \subseteq Q$ is the set of *accepting states*;
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow 2^{(Q \times \Gamma^*)}$ is the *transition function*.

The PDA is nondeterministic because the transition function δ can map the same element of the range to different elements of the range.

A *configuration* of the PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a triple (q, x, α) where $q \in Q, x \in \Sigma^*$ and $\alpha \in \Gamma^*$.

Pushdown Automaton (3)

$(p, x, \alpha) \vdash_M (q, y, \beta)$ represents the PDA M moving from configuration (p, x, α) to configuration (q, y, β). This can occur in two ways:

- an input symbol is read; or
- a Λ-transition occurs.

i.e. $x = \sigma y, \sigma \in \Sigma \cup \{\epsilon\}$.

If $\alpha = X\gamma, X \in \Gamma, y \in \Gamma^*$ then $\beta = \xi\gamma$ where $(q, \xi) \in \delta(p, \sigma, X)$.

$(p, x, \alpha) \vdash_M^n (q, y, \beta)$ denotes moving from configuration (p, x, α) to configuration (q, y, β) in n steps.

$(p, x, \alpha) \vdash_M^* (q, y, \beta)$ denotes moving from configuration (p, x, α) to configuration (q, y, β) in zero or more steps.

Pushdown Automaton (4)

If $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ and $x \in \Sigma^*$, then x is *accepted by* M if $\exists \alpha \in \Gamma^*, q \in A$ such that

$$(q_0, x, Z_0) \vdash_M^* (q, \epsilon, \alpha)$$

A language $L \subseteq \Sigma^*$ is said to be accepted by M if L is precisely the set of strings accepted by M.

Theorem

A language is context-free iff it can be recognized by a pushdown automaton.

Pushdown Automaton (5)

Example PDAs

AnBn
$Q = \{q_0, q_1, q_2, q_3\}$
$A = \{q_3\}$
The transition table for *AnBn* is:

State	Input	Top of Stack	Move(s)
q_0	ϵ	Z_0	(q_3, Z_0)
q_0	a	Z_0	(q_1, aZ_0)
q_1	a	a	(q_1, aa)
q_1	b	a	(q_2, ϵ)
q_2	b	a	(q_2, ϵ)
q_2	ϵ	Z_0	(q_3, Z_0)

The sequence of moves that accepts *aabb* is:
$(q_0, aabb, Z_0) \vdash (q_1, abb, aZ_0) \vdash (q_1, bb, aaZ_0) \vdash (q_2, b, aZ_0) \vdash$
$(q_2, \epsilon, Z_0) \vdash (q_3, \epsilon, Z_0)$

Pushdown Automaton (6)

SimplPal
$Q = \{q_0, q_1, q_2\}$ $A = \{q_2\}$

The transition table for *SimplPal* is:

State	Input	Top of Stack	Move(s)
q_0	a	Z_0	(q_0, aZ_0)
q_0	b	Z_0	(q_0, bZ_0)
q_0	a	a	(q_0, aa)
q_0	b	a	(q_0, ba)
q_0	a	b	(q_0, ab)
q_0	b	b	(q_0, bb)
q_0	c	Z_0	(q_1, Z_0)
q_0	c	a	(q_1, a)
q_0	c	b	(q_1, b)
q_1	a	a	(q_1, ϵ)
q_1	b	b	(q_1, ϵ)
q_1	ϵ	Z_0	(q_2, Z_0)

$(q_0, abcba, Z_0) \vdash$
$(q_0, bcba, aZ_0) \vdash$
$(q_0, cba, baZ_0) \vdash$
$(q_1, ba, baZ_0) \vdash$
$(q_1, a, aZ_0) \vdash$
$(q_1, \epsilon, Z_0) \vdash$
(q_2, ϵ, Z_0)

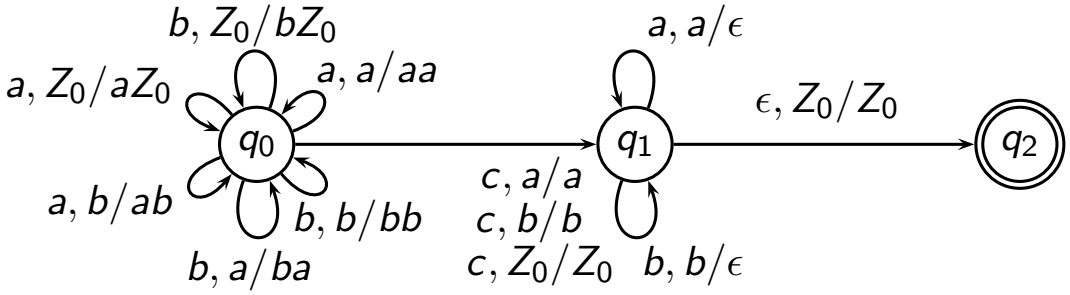
Pushdown Automaton (7)

We can rewrite the same PDA as:

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{a, b, c\} \\
\Gamma &= \{a, b\} \\
A &= \{q_2\} \\
\delta &= ((q_0, a, Z_0), (q_0, aZ_0)), ((q_0, b, Z_0), (q_0, bZ_0)), \\
&\quad ((q_0, a, a), (q_0, aa)), ((q_0, b, a), (q_0, ba)), \\
&\quad ((q_0, a, b), (q_0, ab)), ((q_0, b, b), (q_0, bb)) \\
&\quad ((q_0, c, Z_0), (q_1, Z_0)), ((q_0, c, a), (q_1, a)) \\
&\quad ((q_0, c, b), (q_1, b)), ((q_1, a, a), (q_1, \epsilon)) \\
&\quad ((q_1, b, b), (q_1, \epsilon)), ((q_1, \epsilon, Z_0), (q_2, Z_0))
\end{aligned}
$$

Pushdown Automaton (8)

Or draw it as a transition diagram;



Pushdown Automaton (9)

Consider the language $EvenPal = \{ww^r | w \in \{a, b\}^*\}$

The PDA that accepts *EvenPal* is:

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{a, b\} \\
\Gamma &= \{a, b\} \\
A &= \{q_2\} \\
\delta &= ((q_0, a, Z_0), (q_0, aZ_0)), ((q_0, a, a), (q_0, aa)), \\
&\quad ((q_0, a, b), (q_0, ab)), ((q_0, b, Z_0), (q_0, bZ_0)), \\
&\quad ((q_0, b, a), (q_0, ba)), ((q_0, b, b), (q_0, bb)), \\
&\quad ((q_0, \epsilon, \epsilon), (q_1, \epsilon)), \\
&\quad ((q_1, a, a), (q_1, \epsilon)), ((q_1, b, b), (q_1, \epsilon)), \\
&\quad ((q_1, \epsilon, Z_0), (q_2, \epsilon))
\end{aligned}
$$

Deterministic PDA

A PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is deterministic if we can always decide which transition will be used next, i.e.

- $\forall q \in Q, \sigma \in \Sigma \cup \{\epsilon\}, X \in \Gamma$, the set $\delta(q, \sigma, X)$ has at most one element; **and**
- $\forall q \in Q, \sigma \in \Sigma, X \in \Gamma$, the sets $\delta(q, \sigma, X)$ and $\delta(q, \epsilon, X)$ cannot both be nonempty.

The *SimplPal* language is deterministic. The *EvenPal* language is nondeterministic. **Why?**

A language L is deterministic context-free if there is some deterministic PDA that recognizes L.

Theorem

Not every non-deterministic PDA can be converted to an equivalent deterministic PDA.

This has serious implications for the efficient parsing of context-free languages.

Pumping Lemma for Context-Free Languages

Theorem

If L is a context-free language there is an integer n such that $\forall u \in L$ with $|u| \geq n$, $u = vwxyz$ such that

1. $|wy| > 0$
2. $|wxy| \leq n$
3. $\forall m \geq 0, vw^m xy^m z \in L$

Example: The language $AnBnCn = \{a^n b^n c^n | n \geq 0\}$ is not a context-free language.

Pumping Lemma for Context-Free Languages (2)

Proof.

Let's assume $AnBnCn$ is context-free. Then

$$a^n b^n c^n = vw^m xy^m z, \forall m \geq 0.$$

Conditions 1 and 2 imply that wxy has at least one symbol and no more that 2 distinct symbols.
Let σ_1 be one of the symbols occurring in wy and σ_2 be the symbol that does not occur in wy.
Then the string $vw^0 xy^0 z$, which deletes w and y from u, will have less than n occurrences of σ_1 but exactly n occurrences of σ_2.
But $u = vw^0 xy^0 z \in AnBnCn$ must have an equal number each symbol but we have shown that number of occurrences of σ_2 is greater than the number of occurrences of σ_1. This contradiction invalidates the assumption that $AnBnCn$ is context-free. □

CA320 - Computability & Complexity

Context-Sensitive Languages

Dr. David Sinclair

Context-Sensitive Grammars

An *unrestricted grammar* is a 4-tuple $G = (V, \Sigma, S, P)$, where V and Σ are disjoint sets of variables and terminals respectively. $S \in V$ is called the *start symbol* and P is a set of production rules of the form $\alpha \rightarrow \beta$.

A *Context-Sensitive Grammar* (CSG) is an *unrestricted grammar* in which every production is of the form $\alpha \rightarrow \beta$ and $|\beta| \geq |\alpha|$, i.e. no production rule is length-decreasing.

An Example CSG

An example CSG is:

$$
\begin{aligned}
S &\rightarrow aBCT|aBC \\
T &\rightarrow ABCT|ABC \\
BA &\rightarrow AB \\
CA &\rightarrow AC \\
CB &\rightarrow BC \\
aA &\rightarrow aa \\
aB &\rightarrow ab \\
bB &\rightarrow bb \\
bC &\rightarrow bc \\
cC &\rightarrow cc
\end{aligned}
$$

How a variable is derived depends on the context!

An Example CSG (2)

Here are some derivations from this CSG.

$$
\begin{aligned}
S &\Rightarrow aBC \\
&\Rightarrow abC \\
&\Rightarrow abc
\end{aligned}
$$

$$
\begin{aligned}
S &\Rightarrow aBCT \\
&\Rightarrow aBCABC \\
&\Rightarrow aBACBC \\
&\Rightarrow aABCBC \\
&\Rightarrow aABBCC \\
&\Rightarrow aaBBCC \\
&\Rightarrow aabBCC \\
&\Rightarrow aabbCC \\
&\Rightarrow aabbcC \\
&\Rightarrow aabbcc
\end{aligned}
$$

$L(G) = \{a^n b^n c^n | n \geq 1\}$

We have already shown that *AnBnCn* is not a context-free language using the CFG pumping lemma. But it is a context-sensitive language.

CSLs are not a generalisation of CFGs as CSLs cannot have any Λ-productions.

Linear Bounded Automata

A *linear bounded automaton* (LBA) is a finite state machine with a finite length data store called a *tape*. The tape consists of a sequence of cells, where each cell can store a symbol from the machine's alphabet. Symbols can be written or read from any position on this tape and therefore the LBA has a *read-write head* that can be moved left or right one cell.

The tape is used both to store the input and any ongoing calculations. There are 2 special symbols, [and], that are used to mark the finite bounds of the tape. The read-write head cannot move beyond either of these symbols and it cannot overwrite these symbols.

Linear Bounded Automata (2)

At each step the LBA read the symbol under the read-write head, replaces the by another symbol (could be the same symbol) and then perform one of four possible actions $\mathcal{A} \in \{Y, N, L, R\}$, where:

- Y denotes "Yes", accept the input string
- N denotes "No", reject the input string
- L denotes "Left", move the read-write head one cell to the left
- R denotes "Right", move the read-write head one cell to the right

Linear Bounded Automata (2)

Formally, a linear bounded automaton is a 5-tuple $M = \{Q, \Sigma, \Gamma, q_0, \delta\}$ where:

- Q is a finite set of *states*;
- Σ is a finite alphabet (*input symbols*);
- Γ is a finite alphabet (*store symbols*);
- $q_0 \in Q$ is the *initial state*; and
- $\delta : Q \times (\Gamma \cup \{[,]\}) \rightarrow Q \times (\Gamma \cup \{[,]\} \times \mathcal{A}$, is the *transition function*.

If $((q, \sigma), (q', \psi, \mathcal{A})) \in \delta$ then when in state q with σ at the current read-write head position, M will replace σ by ψ and perform action $\mathcal{A}$ and enter state q'.

M accepts $w \in \Sigma^*$ iff it starts with configuration $(q_0, [w])$ and the action Y is taken.

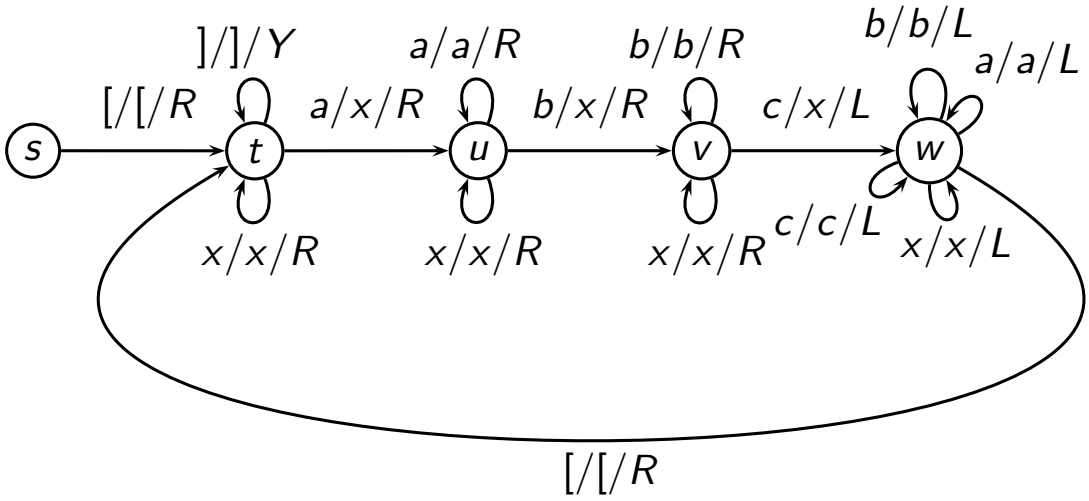
Linear Bounded Automaton Example

An LBA to accept $AnBnCn = \{a^n b^n c^n | n \geq 0\}$ is:

$$
\begin{array}{lcl}
Q & = & \{s, t, u, v, w\} \\
\Sigma & = & \{a, b, c\} \\
\Gamma & = & \{a, b, c, x\} \\
q_0 & = & \mathrm{s} \\
\delta & = & \{((s, [), (t, [, R)), \\
 & & ((t,]), (t,], Y)), ((t, x), (t, x, R)), ((t, a), (u, x, r)), \\
 & & ((u, a), (u, a, R)), ((u, x), (u, x, R)), ((u, b), (v, x, R)) \\
 & & ((v, b), (v, b, R)), ((v, x), (v, x, R)), ((v, c), (w, x, L)) \\
 & & ((w, c), (w, c, L)), ((w, b), (w, b, L)), ((w, a), (w, a, L)) \\
 & & ((w, x), (w, x, L)), ((w, [), (t, [, R)) \\
 & & \}
\end{array}
$$

Linear Bounded Automaton Example (2)

Or as a transition diagram;



where $\sigma/\psi/\mathcal{A}$ denotes reading symbol σ, writing symbol ψ and performing action $\mathcal{A}$.

Linear Bounded Automaton Example (3)

Intuitively the previous LBA behaves as follows.

- The LBA performs multiple passes over the input string.
- On each pass (from left to right) starting at the start symbol `[` in state t, the LBA coverts the first `a` into an `x`, and then the first `b` into an `x` and finally the first `c` into an `x`.
- After converting a `c` into an `x` (after matching it with an `a` and `b`) the LBA moves right to left until it reaches the start symbol `[` and goes into state t.
- If the LBA in state t only encounters `x` symbols from the start symbol `[` to the end symbol `]`, then the LBA performs a Y action.
- the LBA gets stuck in either state u, v or w if there is not a "matching" `a`, `b` or `c` symbol respectively.

Can you design a better version of this LBA that actually rejects string that are not in the language *AnBnCn*?

CA320 - Computability & Complexity

Decidability

Dr. David Sinclair

Diagonalisation

How do we compare the sizes of two infinite set?

For example comparing the set of natural numbers $\{1, 2, 3, \ldots\}$ with the set of even natural numbers $\{2, 4, 6, \ldots\}$ is one of these bigger that the other?

To answer this we need the concept of a *correspondence*. A *correspondence* is a function $f : A \rightarrow B$ that is:

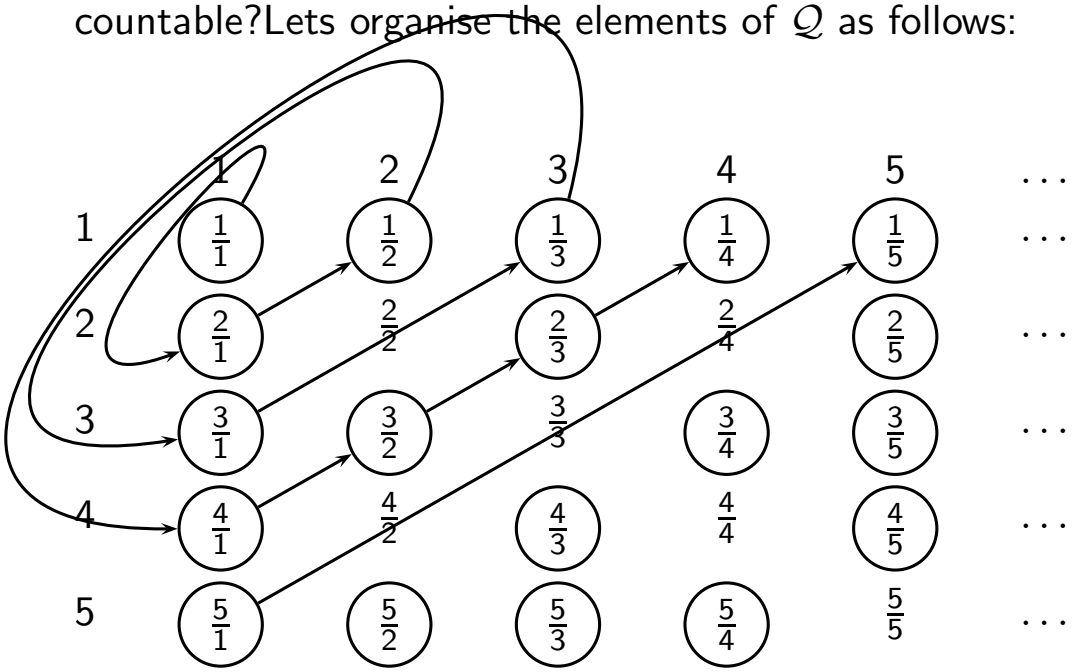
- **one-to-one**, that is it never maps two different elements of A to the same element of B; and
- **onto**, that is $\forall c \in B,\ \exists a \in A$ such that $f(a) = b$.

There is a *correspondence*, $f(n) = 2n$, between $A = \{1, 2, 3, \ldots\}$ and $B = \{2, 4, 6, \ldots\}$, so both sets have the same size.

Diagonalisation (2)

A set is countable if it is either finite or has the same size as $\mathcal{N}$.

Is the set $\mathcal{Q} = \{\frac{m}{n} | m, n \in \mathcal{N}\}$, the set of positive rational numbers, countable?Lets organise the elements of $\mathcal{Q}$ as follows:



Diagonalisation (3)

Each "bottom-left to upper right" diagonal is finite in size and hence we can list all the distinct elements of $\mathcal{Q}$. Since there is a correspondence between $\mathcal{Q}$ and $\mathcal{N}$, $\mathcal{Q}$ is countable.

Is the set $\mathcal{R}$, the set of real numbers, countable?

Theorem

The set $\mathcal{R}$ is uncountable.

Proof.

We will assume a correspondence f exists and show this generates a contradiction, hence no correspondence really exists and $\mathcal{Q}$ is uncountable. We will do this by constructing $x \in \mathcal{R}$ and showing that it cannot be paired with anything in $\mathcal{N}$.

Diagonalisation (4)

Proof (contd.)

Suppose a correspondence f exists. Here is a hypothetical example

n	$f(n)$
1	0.<u>1</u>4159...
2	0.5<u>5</u>555...
3	0.12<u>3</u>45...
4	0.500<u>0</u>0...
⋮	⋮

Construct x as follows. Let the i-th fractional digit of x not be equal to the i-th fractional digit of $f(i)$.
e.g. $x = 0.4641\ldots$
Hence $\forall i \in \mathcal{N},\ x \neq f(i)$.

Since we have created an x that cannot be paired with any element of $\mathcal{N}$ our assumption that a correspondence existed is false and $\mathcal{R}$ is uncountable. □

This is an example of the diagonalisation techniques developed by Georg Cantor in 1873.

Undecidability and the Halting Problem

Theorem

The language $A_{TM} = \{\langle M, w\rangle | M$ is a Turing machine and M accepts $w\}$ is undecidable.

Proof.

Let's assume a Turing machine H is a decider for A_{TM}.

$$H(\langle M, w\rangle) = \begin{cases} accept, & \text{if } M \text{ accepts } w \\ reject, & \text{if } M \text{ halts and does not accept } w \end{cases}$$

Let D be a Turing machine that uses H as follows.

D(M) = **1.** Run H on input $\langle M, M\rangle$.
2. Output the opposite of what H produces.

Therefore,

$$D(M) = \begin{cases} accept, & \text{if } M \text{ halts and does not accept } M \\ reject, & \text{if } M \text{ accepts } M \end{cases}$$

Undecidability and the Halting Problem (2)

Proof (contd.)

If we run D on its own description we get:

$$D(D) = \begin{cases} \textit{accept}, & \text{if } D \text{ halts and does not accept } D \\ \textit{reject}, & \text{if } D \text{ accepts } D \end{cases}$$

This contradiction implies that the initial assumption that A_{TM} is decidable is false. □

This is just another version of the diagonalisation argument. Consider the table $H(M_i, M_j)$ with some fictional values.

	M_1	M_2	M_3	...	D	...
M_1	**accept**	reject	accept	...	accept	...
M_2	accept	**reject**	accept	...	accept	...
M_3	accept	reject	**accept**	...	reject	...
⋮						
D	reject	accept	reject	...	?	...
⋮						

Undecidability and the Halting Problem (3)

Theorem

The language $HALT_{TM} = \{\langle M, w\rangle | M$ is a Turing machine and M halts on input $w\}$ is undecidable.

Proof.

Assume that R is a Turing machine that decides $HALT_{TM}$. Using R we can construct a TM S that decides A_{TM}. S operates as follows on the input $\langle M, w\rangle$ where M is a Turing machine and w is a string.

1. Run R on $\langle M, w\rangle$.
2. If R rejects $\langle M, w\rangle$, then reject.
3. If R accepts, then simulate M on w until it halts.
4. If M accepts, then return accept else return reject.

But since A_{TM} is undecidable then the initial assumption that there exists a Turing machine that decides $HALT_{TM}$ must be false. □

Reducability

Rather than using a diagonalisation proof to show the undecidability of a language L_1 we can map/reduce the language to a language L_2 whose decidability is already known.

The language L_1 is *mapping reducible* to language L_2, written $L_1 \leq L_2$, if there is a computable function $f : \Sigma^* \rightarrow \Sigma^*$ where for every w,

$$w \in L_1 \Leftrightarrow f(w) \in L_2$$

If $L_1 \leq L_2$ and L_2 is decidable, then L_1 is decidable.

If $L_1 \leq L_2$ and L_2 is undecidable, then L_1 is undecidable.

Other Undecidable Problems

Some examples of:

- Given a Turing machine M, is there any string at all on which M halts?
- Given a Turing machine M, does M halt on every input?
- Given two Turing machines M_1 and M_2, do they halt on the same input strings?
- Given a Turing machine M, is the language M accepts regular? Is it context-free? Is it Turing- decidable?
- Does a particular line (transition) in a program (machine) get executed?
- Does a program contain a computer virus?

CA320 - Computability & Complexity

Introduction to Haskell

Dr.David Sinclair

These are just notes! Programming requires practice.

- These notes are just to give you an overview of the Haskell programming language. The real work will be in the lab sessions.
- Attending and actively participating in the lab sessions is **vital**.
- Programming is part knowledge, part design and part skill. The *design* element required experience (which comes through **practice**). The *skill* element requires **practice**.
- Programming in Haskell, or any other functional language, requires you to think differently than programming in an imperative language.

Styles of Programming Languages

There are two basic styles of programming languages.

Imperative In *imperative programming* you describe how to do something, usually as a series of sequential operations (that modify the state of the program) and conditional jumps (based on the state of the program). Imperative programming is all about *state*, and these state changes can be persistent and as a result have *side-effects*.

Declarative In *declarative programming* you describe "what to do", not "how to do". Declarative languages fall into 2 broad categories, *logic programming* in which you describe the properties of the solution and then the programming language searches for the solution; and *functional languages* in which you describe the solution as the evaluation of mathematical functions that avoids state and side-effects.

Functional Languages and Referential Transparency

- A functional language program is constructed using *expressions*. Each *expression*, which can be a function call, has a *meaning* which evaluates to a *value*.
- There is no sequencing.
- There is no concept of state in the sense of variables and the values assigned to them.
- As a result of this there are no side-effects! An expression, or a function, simply evaluates to a value. If you replace an expression, or a function, by another expression, of function, that evaluates to the same values given the same arguments, it will have no effect on the rest of the program. This is called *referential transparency*.

Advantages of functional languages

Why use a language that does not have variables, assignments and sequencing?

- Concepts such as variables and sequencing only really make sense if you focus on the hardware of a computer. By focusing at this level we are really modeling a particular style of a *solution*. Functional programming is more abstract and focuses on modeling the *problem*.
- Functional programs are easier to reason about.
- Functional programming promotes good programming practices. Even if you never use function programming again, your programming in imperative languages will improve.

Disadvantages of functional languages

Functional languages are not perfect!

They are particularly difficult to use:

- when you require input and output (because these require side-effects); and
- when the program needs to run continuously and/or requires interaction with the user.

Because functional programs are at a level of abstraction above the hardware, it can be hard to reason about the time and space complexity of functional programs.

Haskell

The Haskell programming language was designed in 1998 by Peyton-Jones *et al* and is named after Haskell B. Curry who pioneered the λ-calculus, the mathematical theory on which functional programming is based.

Haskell is:

- a *purely functional language* with no side-effects and referential transparency
- a *lazy* programming language, in that it does not evaluate an expression unless it really has to; and
- a *statically typed* so that at compile time we know the type of each thing and has a *type inference* system that works out the type of a thing if it hasn't been explicitly specified.

Types

Everything in Haskell has a type!

There are *basic types*. Some examples are:

Keyword	type	value
`Int`	integer	12
`Float`	floating-point number	-1.23
`Char`	character	`'A'`
`Bool`	boolean	`True`

There are *compound types*. Some examples are:

Description	type	example
tuple	`(Int, Float)`	`(2,12.75)`
list	`[Int]`	[1,3,5,7,9]
string	`String`	``"Hello there"''
function	`Int -> Int`	`square`

Functions and Function Types

Here is a quick example.

```
rectangleArea x y = x * y
```

A function has a *name* and a set of parameters, each separated by a space. How the function is evaluated is defined by what follows the = symbol.

This only tells part of the story of the `rectangleArea` function.

- A function always evaluates to a value which has a type.
- Each of the parameters of a function have a type.

```
rectangleArea ::  Float -> Float -> Float
rectangleArea x y = x * y
```

The parameters and return type are separated by -> in the function declaration. The return type is the last item and the parameter types are the previous items.

if ... then ... else

Haskell has an `if ... then ... else` expression.

```
doubleSmallNumber x = if x > 100
                      then x
                      else x*2
```

Because it is an expression we can use `if ... then ... else` inside another expression.

```
incDoubleSmallNumber x = (if x > 100 then x else x*2) + 1
```

Lists

Lists are a very useful data structure. They are a *homogeneous* data structure where each element of the list has the same type.

```
let numbers = [1,3,5,7,9,11,13,15]
```

creates a list called `numbers`.

```
let joined = [1,3,5,7,9] ++ [2,4,6,8,10]
```

The ++ operator *concatenates* two lists, in this case generating the list `[1,3,5,7,9,2,4,6,8,10]` called `joined`.

Strings are lists of characters so,
`"Hello " ++ "World"`

evaluates to `"Hello World"`.

Lists (2)

The *cons* operator, `:`, adds an element to the start of a list.

```
let mylist = 1:[5,4,3,2]
```

evaluates to a list called `mylist` containing `[1,5,4,3,2]`.

```
1:2:3:[]
```

evaluates to the list `[1,2,3]`.

`[]` is the empty list.
So `3:[]` evaluates to `[3]`.
`2:[3]` evaluates to `[2,3]`.
`1:[2,3]` evaluates to `[1,2,3]`.
In fact, `[1,2,3]` is syntactic sugar for `1:2:3:[]`.

Lists (3)

To access a specific element of a list we use the `!!` operator. The `!!` operator, like the ++ operator, is an infix operator. Before the `!!` operator we have the list we are accessing and after the `!!` operator we have the index of the element we are accessing (indices start at 0).

```
[[1,2,3],[2,4,6,8],[1,3,5,7,9]] !!  2
```

results in the list `[1,3,5,7,9]` since this list is at index 2 of the lists of lists.

How would we get the value 7 from `[[1,2,3],[2,4,6,8],[1,3,5,7,9]]`?

Lists can be compared lexicographically using the <, <=, > and >= operators. The first elements are compared and if they are equal the second elements are compared, and so on.

Some useful List functions

`head [2,4,6,8]` evaluates to 2.

`tail [2,4,6,8]` evaluates to `[4,6,8]`.

`last [2,4,6,8]` evaluates to 8.

`init [2,4,6,8]` evaluates to `[2,4,6]`.

`length [2,4,6,8]` returns the size of the list, which in this case evaluates to 4.

`null [2,4,6,8]` tests whether or not a list is empty, which in this case evaluates to `False`.

`reverse [2,4,6,8]` reverses a list, which in this case evaluates to `[8,6,4,2]`.

`take 2 [2,4,6,8]` extracts the specified number of elements from the start of a list, which in this case evaluates to `[2,4]`.

Some useful List functions (2)

`drop 3 [2,4,6,8]` removes the specified number of elements from the start of a list, which in this case evaluates to `[8]`.

`maximun [2,8,4,9,6]` returns the largest element of a list, which in this case evaluates to 9.

`miniumum [2,8,4,9,6]` returns the smallest element of a list, which in this case evaluates to 2.

`sum [2,8,4,9,6]` returns the sum of all the elements of a list, which in this case evaluates to 29.

`product [2,8,4,9,6]` returns the sum of all the elements of a list, which in this case evaluates to 384.

`elem 4 [2,8,4,6]` tests if the specified element is contained in a list, which in this case evaluates to `True`. This can also be written as an infix operator, i.e. `4 ‘elem‘ [2,8,4,6]`.

Lists and Ranges

Rather than having to write down all the elements of a list, we can use *ranges* if there is a regular interval between the elements.

`[1,2..10]` evaluates to the list `[1,2,3,4,5,6,7,8,9,10]`.

`[2,4..20]` evaluates to the list `[2,4,6,8,10,12,14,16,18,20]`.

Ranges also work with characters.

`['a'..'z']` evaluates to the string `"abcdefghijklmnopqrstuvwxyz"`.

Ranges can be used to generate infinite lists, e.g.,

```
[1,2..]
[12,24..]
```

List Comprehension

When we defined sets we used a technique called *set comprehension*, e.g. $\{2x|x \in \mathcal{N}, x \leq 10\}$.
The expression before the pipe operator, (|), is called the output function. In this case the output function is a function of x. The expressions after the pipe are the predicates that the variables of the output function must satisfy.

We can do something very similar for lists and this is called *list comprehension*.

```
[ 2*x | x <-[1..10]]
```

```
[ x*y | x <- [5,10,15], y <- [4..6], x*y < 50]
```
evaluates to `[20,25,30,40]`

What does this function do?
```
mysteryFn ::  String -> String
mysteryFn x = [y | y <- x, y `elem` ['a'..'z']]
```


Tuples

Lists are homogeneous collections of data. If we want to collect together data of *different types* we need to use *tuples*.

A *tuple* has its own type and that type depends on its size, the types of its components and the order in which they occur.

The tuples `(5,"hello")` and `("world", 1)` have different types since the first tuple has the type `(Int, String)` whereas the second tuple has the type `(String, Int)`.

A *pair* is the smallest tuple and pairs have 2 functions, `fst` and `snd` for extracting data from them.

`fst (5,"hello")` evaluates to 5 and `snd(5,"hello")` evaluates to `"hello"`.

Extracting data from larger tuples, such as triple, 4-tuples, 5-tuples, etc., requires pattern matching.

Tuples (2)

For example, we could use a tuple to represent a triangle. Each element of the tuple is the length of one of the sides. Let's limit ourselves to triangles with integer side lengths.

We could generate a list of triangles whose perimeter was at most 24 as follows.

```
[(a,b,c) | a <-[1..24], b<-[1..24], c<-[1..24], a+b+c
<= 24]
```

This generates a lot of triangles most of which are duplicates since the triangles `(10,12,2)`, `(12,10,2)`, `(2,12,10)` and many more are effectively the same triangle. We can remove the "duplicates" by letting `a` be the longest side, `b` the second longest and `c` the shortest side.

```
[(a,b,c) | a <-[1..24], b<-[1..a], c<-[1..b], a+b+c
<= 24]
```


Tuples (3)

Still a lot of triangles, but how many of these are right-angle triangles?

```
length [(a,b,c) | a <-[1..24], b<-[1..24],
c<-[1..24], a+b+c <= 24, a^2 == b^2 + c^2]
```

Typeclasses

You can find out the type of a item including functions by using the `:t` command.

`:t head` evaluates to `head::[a]->a`

which says that `head` is a function that takes an array of things of type `a` and evaluates to a thing of type `a`.

`:t (==)` evaluates to `(==)::(Eq a)=>a->a->Bool`

What does the `(Eq a)` before the => signify?

`(Eq a)` is an example of a *class constraint*. In this case, the == function takes two items of the same type and this type must belong to the `Eq` class. The `Eq` class includes all standard Haskell types except IO and functions.

Typeclasses (2)

Some other type classes are:

Ord This is the *Ordered* class. Functions using `Ord` are `>=`, `<=`, `>`, `<`, `compare`, `max` and `min`. Most Haskell standard types belong to `Ord` except function types and some abstract types.

Show Types belonging to this class can be displayed as strings. Again most Haskell standard types belong to `Show` except function types.

Read Contains all the type that can be taken in as a string and converted into a type. Again all standard types are included in `Read`, but sometimes some additional information is required if the intended type is unclear. `read "2"` returns an ambiguous type error, so `read "2"::Int` is required.

Num This is the *Numeric* class and includes all types that act as numbers.

Integral This class includes only whole numbers.

Pattern Matching and Recursion

Pattern marching and *recursion* are very common programming constructs in functional programming. *Recursion* is where a a function definition calls itself when performing an evaluation. To prevent an infinite loop, a recursive function must have a *base case* and the parameters of each invocation of the recursive function should move closer to the *base case*.

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- The 2nd line is the *base case*.
- Lines 2 and 3 demonstrate pattern matching. If the parameter to the `factorial` function is `0` the 2nd line is evaluated. If the parameter is not `0` the 3rd line is evaluated. The parameter is matched to `n` and the `factorial` function is evaluated with `n-1`.

Pattern Matching and Recursion (2)

We can write our own version of the standard list`length` function.

```
listLength :: (Integral b) => [a] -> b
listLength [] = 0
listLength (_:xs) = 1 + listLength xs
```

- The *base case* is the empty list.
- If the list we are measuring is not the empty list we try the next pattern `_:xs`.
  - The list is matched against `_:xs`, which is a list with something at its head (start) followed by a tail.
  - The tail of the list is matched with `xs`.
  - The head of the list is matched with `_`. We use `_` when we don't care what the value we are matching against is.
  - It is **very important** to make sure that the patterns you specify match against all possible patterns.

as Patterns

To refer to the original complete item and not just the elements of the pattern, we can use *as patterns*. An *as pattern* consists of the name of the complete item followed by an `@` and then the pattern.

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++
                     " is " ++ [x]
```

We need to be careful with the amount of whitespace we use at the start of a line in Haskell as Haskell is whitespace sensitive.

- The first non-whitespace after a `do`, `let` or `where` defines the start of a *layout block* and its column number is rememberd.
- If a new line has the same amount of whitespace as the start of the layout block it is a new line.
- If a new line has more whitespace, it is a line continuation.
- If a new line has less whitespace, it is a new line.

Guards

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "Have a steak!"
    | weight / height ^ 2 <= 25.0 = "Supposedly normal!"
    | weight / height ^ 2 <= 30.0 = "How about a walk?"
    | otherwise                   = "Skip that snack"
```

- *Guards* follow a function name and parameters. They are boolean expressions that follow a pipe (`|`) symbol.
- It is good practice to ensure the pipes are aligned.
- When a guard evaluates to `True`, the function body is evaluated.
- When a guard evaluates to `False`, the next guard is checked.
- `otherwise` is defined as `True` and acts as a "catch-all".

Where

The `where` keyword allows you to share *name bindings* within a function.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "Have a steak!"
    | bmi <= normal = "Supposedly normal!"
    | bmi <= comfortable = "How about a walk?"
    | otherwise = "Skip that snack"
    where bmi = weight / height ^ 2
                (skinny, normal, comfortable)
                      = (18.5, 25.0, 30.0)
```

- Note that you can use pattern matching inside `where` blocks.

Where (2)

Here is another example of the `where` keyword and pattern matching.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

`where` bindings can also be nested.

- It is common when defining a function to define some helper function in its where clause and then to give those functions helper functions as well, each with its own where clause.

Let

`let` bindings are far more local than `where` bindings. `let` bindings have the form:

`let` *bindings* `in` *expressions*

For example:

```
cylinderArea :: (RealFloat a) => a -> a -> a
cylinderArea r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r^2
    in  sideArea + 2 * topArea
```

`let` bindings are expressions in themselves, so we can write expressions such as:

```
4 * (let a = 9 in a + 1) + 2
```


Case

case expressions are like case statements from imperative languages but with pattern matching added.

```
headOfList :: [a] -> a
headOfList xs = case xs of []    -> error "Empty list!"
                           (x:_) -> x
```

But couldn't we have written `headOfList` as:

```
headOfList :: [a] -> a
headOfList [] = error "Empty list!"
headOfList (x:_) = x
```

Yes, they are interchangeable because pattern matching in function definitions is just syntactic sugar for case expressions.

More Recursion

Let's implement our own versions of a few standard list functions

```
takeList :: (Num i, Ord i) => i -> [a] -> [a]
takeList n _
        | n <= 0   = []
takeList _ []      = []
takeList n (x:xs) = x : takeList (n-1) xs
```

Note that there are 2 base cases, `n <= 0` and the empty list.

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

While this works it is not the most efficient solution as concatenating strings takes time.

More Recursion (2)

A common trick with recursive functions is to use an *accumulator*.

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

reverseList2 :: [a] -> [a]
reverseList2 x = reverseListAcc x []
```

Quicksort

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```


Higher Order Functions

Why do Haskell function type signatures have structures like `a->a->b->c` and not `a,a,b->c`?

Because Haskell functions actually only take **one** parameter!

Consider the `max` function.

```
max :: (Ord a) => a -> a -> a
```

When `max 4 5` is evaluated it creates a new function that takes one parameter and evaluates to 4 if the new parameter is less than 4 or the new parameter if it is greater than 4.

So `max 4 5` and `(max 4) 5` are equivalent and a <space> between 2 items is a *function application*.

`max` is an example of a *curried function*.

Higher Order Functions (2)

Why use *curried functions*?

Because when you supply a curried function with less parameters than required you get back a *partially applied function* which is like a new function that used as a parameter in another function.

```
comparehi = compare 100
comparelo = compare 20

checkThreshlods :: (Num a) => a -> (a->Ordering) ->
                                  (a-> Ordering) -> String
checkThresholds x hi lo
    | high = "too high"
    | low = "too low"
    | otherwise = "OK"
   where (high, low) = (hi x == LT, lo x == GT)

compareThresholds 80 comparehi comparelo
```


Higher Order Functions (3)

`map` takes a function and applies it to every element of a list.

Some examples:

- `map (>3) [1,8,2,4]` evaluates to `[False,True,False,True]`
- `map fst [(3,2),(1,4),(2,5)]` evaluates to `[3,1,2]`
- `map (map (^2)) [(3,2),(1,4),(2,5)]` evaluates to `[(9,4),(1,16),(4,25)]`

`filter` takes a predicate and a list and evaluates to a list of the elements of the original list that satisfy the predicate.

- `filter (>3) [1,8,2,4,5,1,6]` evaluates to `[8,4,5,6]`
- `let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]` evaluates to `[[1,2,3],[3,4,5],[2,2]`.

Higher Order Functions (4)

Find the sum of all odd squares that are smaller than 10,000.

- `sum` is a function that sums the elements of a list.
- `takeWhile` is a function that removes elements from that start of a list while a predicate is true.

```
sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Things to note:

- An infinite data structure.
- Functions as parameters.
- Partially applied functions.

Lambda Functions

These are anonymous functions that are mainly used as a parameter to a higher-order function.

```
chain :: (Integral a) => a -> [a]   -- Collatz chains
chain 1 = [1]
chain n
    | even n =  n:(chain (n `div` 2))
    | odd n  =  n:(chain (n*3 + 1))

numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15)
                               (map chain [1..100]))
```

Lambda functions are expressions and hence can be passed to functions.

Input/Output

Input/Output (I/O) can be tricky in a purely functional environment since there are no side-effects and no changes of state. For the contents of a screen to change there has to be some state change.

Haskell provides a mechanism for I/O that isolates the state changes from the rest of the Haskell program, maintaining the benefits of a purely functional programming language.

```
main = do
         putStrLn "What's your name?"
         name <- getLine
         putStrLn ("Hi " ++ name")
```

- I/O is performed by *IO actions*.

Input/Output (2)

- `putStrLn`'s type is `String->IO()`, i.e. it takes a `String` and evaluates to an *IO action* with a result type of `( )`, `empty tuple`.
- `getLine`'s type is `IO String`, i.e. it evaluates to an *IO action* with a result type of `String`.
- The `do` keyword creates a block of *IO actions* that are "executed" in sequence and act as one composite *IO action*.
- The syntax `name <- getLine` binds the result of `getLine` to `name`.

Input/Output (3)

A common construct is to print out a list of IO actions.

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    sequence $ map print rs
```

- `$` is the function application operator. `x $ y` applies function `x` to the results of `y`

`mapM` maps the function over the list and then sequences it. `mapM_` does the same, only it throws away the result.

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    mapM_ print rs
```


Input/Output (4)

Other useful I/O functions are:

- `putStr` write a string to the terminal but doesn't terminate it with a newline.
- `getChar` which reads a character from the terminal.
- `getContents` which reads everything from the terminal until an end-of-file character.
  - The key aspect of `getContents` is that it is a *lazy* function. It doesn't read everything from the terminal and store it somewhere. Instead it return but not actually read anything from the terminal **until you really need it**!

File Input/Output

File I/O is done by *IO Handles*.

```
import System.IO

main = do
    handle <- openFile "file.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

- `openFile` opens a file in either `ReadMode`, `WriteMode`, `AppendMode` or `ReadWriteMode`. `openFile` returns an `IO Handle` which can be bound to an identifier.
- `hGetContents` is line `getContents` except it takes a handle.
- `hClose` takes a file handle and closes the file.
- `hPutStr`, `hPutStrLn`, `hGetLine`, `hGetChar` are like their counterparts, but they take an additional parameter, an `IO Handle`.

CA320 - Computability & Complexity

Introduction

Dr. David Sinclair

Overview

In this module we are going to answer 2 important questions:

- Can all problems be solved by a computer?
- What problems be efficiently solved by a computer?

Computability The study of computable functions. In other words, the study of problems that are computable and hence have an algorithm.

Complexity The classification of computable problems by their inherent difficulty.

Overview (2)

- Introduction
  - Sets, functions and languages
- Introduction of Haskell
- Regular languages and finite automata
  - Regular grammars, regular expressions and finite state automata
- Context-free languages and pushdown automata
  - Context-free grammars, derivations, ambiguity and pushdown automata
- Context-sensitive languages and linear bounded automata
  - Context-sensitive grammars and linear bounded automata
- Models of Computation
  - Turing machines, partial recursive functions, lambda calculus, Church-Turing thesis
- Computability
  - Halting problem and reducibility
- Complexity
  - Asymptotic notation, time complexity and space complexity

Texts

Recommended:

- *Introduction to the Theory of Computation*, Sipser, PWS, ISBN 053494728X, 1996
- *Haskell: The Craft of Functional Programming*, Thompson, Addison-Wesley, ISBN 0-201-34275-8, 1999

Supplementary:

- *Elements of the Theory of Computation*, Lewis and Papadimitriou, Prentice Hall, ISBM 0-13-272741-2, 1998
- *Introduction to Languages and the Theory of Computation*, Martin, McGraw-Hill, ISBN 0-07-115468-X, 1997
- *Programming in Haskell*, Hutton, Cambridge University Press, ISBN 9780521692694, 2007

Contact Details

Lecturer: Dr. David Sinclair
Office: L253
Phone: 5510
Email: david.sinclair@dcu.ie
WWW: http://www.computing.dcu.ie/~davids
Course web page:
http://www.computing.dcu.ie/~davids/CA320/CA320.html

How do I successfully complete this module?

The module mark is a straight weighted average of:

- 25% continuous assessment
  - 2 assignments
    - first assignment (10%)
    - second assignment (15%)
- 75% end-of-semester examination
  - 5 questions. Do 4 questions.
  - 2 sections, at least 1 from section A
    - Section A: Haskell
    - Section B: Computability & Complexity

What if I don't successfully complete this module?

If you fail the module in the end-of-semester exams then you will need to repeat some elements of the assessment.

- If you just failed the exam you can resit the exam in the Autumn.
- If you just failed the continuous assessment then you must complete a resit assignment.
  - Contact me after the results are published for the resit assignment.
- If you failed both the exam and the continuous assessment, you must repeat both.

If you fail the module after the resit examination and continuous assessment you must repeat all aspects of the module in the following year.

Some Mathematical Revision: Logic

Connective	Symbol	Use	English equivalent
conjunction	$\wedge$	$p \wedge q$	p and q
disjunction	$\vee$	$p \vee q$	p or q
negation	$\neg$	$\neg p$	not p
conditional	$\rightarrow$	$p \rightarrow q$	if p then q
			p only if q
biconditional	$\leftrightarrow$	$p \leftrightarrow q$	p if and only if q

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Logic (2)

A *compound proposition* is a proposition constructed from an arbitrary combination of the 5 basic operations.

$$(p \vee q) \wedge \neg(p \rightarrow q)$$

p	q	$p \vee q$	$p \rightarrow q$	$\neg(p \rightarrow q)$	$(p \vee q) \wedge \neg(p \rightarrow q)$
T	T	T	T	F	F
T	F	T	F	T	T
F	T	T	T	F	F
F	F	F	T	F	F

A *tautology* is a compound proposition that is **true** for every possible truth value combination of its constituent propositions.

A *contradiction* is a compound proposition that is **false** for every possible truth value combination of its constituent propositions.

Logic (3)

Two propositions P and Q are *logically equivalent*, $P \Leftrightarrow Q$, if they have the same truth value for every possible combination of base propositions. Hence, in any expression where P is used we can substitute Q and the entire expression remains unchanged.

A proposition P *logically implies* a proposition Q, $P \Rightarrow Q$, if in every case P is true then Q is also true.

Beware of the subtle difference between $P \rightarrow Q$ and $P \Rightarrow Q$!
$P \rightarrow Q$ is a proposition, just like P and Q, whereas $P \Rightarrow Q$, is a *meta-statement*. It is an assertion about the relationship between the propositions P and Q. If $P \Rightarrow Q$, then $P \rightarrow Q$ is a *tautology*.

Logic (4)

There is a large set of logical identities that we can use when manipulating compound propositions. Some of the more useful ones are:

commutative laws	$p \vee q \Leftrightarrow q \vee p$ $p \wedge q \Leftrightarrow q \wedge p$
associative laws	$p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$ $p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$
distributive laws	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$
De Morgan's laws	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$ $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$
	$p \rightarrow q \Leftrightarrow \neg p \vee q$ $p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$
contrapositive	$p \rightarrow q \Leftrightarrow \neg q \rightarrow \neg p$

Logic (5)

Quantifiers

$\forall x(P(x))$ states that the proposition P, which depends on the truth value of x, is true for all values of x.
$\exists x(P(x))$states that the proposition P, which depends on the truth value of x, is true for some value of x.

Quantifiers can combined in the same expression but great care is needed. The following 2 expressions which are very similar mean 2 different things (in fact the second is not true if x and y are from the domain of natural numbers, $\mathcal{N}$).

$$\forall x(\exists y(x < y))$$
$$\exists y(\forall x(x < y))$$

The logical identifiers for quantifiers are:

$$\forall x(P(x)) \Leftrightarrow \neg(\exists x(\neg P(x)))$$
$$\exists x(P(x)) \Leftrightarrow \neg(\forall x(\neg P(x)))$$

Sets

Sets are unordered collections of distinct elements. A set can be described by listing its elements.
$A = \{1, 2, 4, 8\}$
In the case of large (and infinite) sets we can use ellipses (...)
$B = \{2, 4, 6, 8, \ldots, 100\}$
$C = \{0, 5, 10, 15, 20, 25, \ldots\}$
However, a much nicer way of specifying a set is to use a property that all the elements satisfy.
$C = \{x | x = 5i, i \in \mathcal{N}\}$

For any set A, $x \in A$ means that x is an element of A. $x \notin A$ means that x is not an element of A. $A \subseteq B$ means that every element of A is also an element of B. $A \nsubseteq B$ means that at least one element of A is not and element of B.
The *empty set*, the set with no elements, is denoted $\emptyset$.

Sets (2)

Given 2 sets A and B, we can define their *union*, $A \cup B$, their *intersection*, $A \cap B$ and *difference*, $A - B$, as:
$A \cup B = \{x | x \in A \vee x \in B\}$
$A \cap B = \{x | x \in A \wedge x \in B\}$
$A - B = \{x | x \in A \wedge x \notin B\}$

If $A \cap B = \emptyset$ then A and B are *disjoint*.
A collection of sets are *pairwise disjoint* if distinct pairs of sets A and B from the collection are disjoint.

Sets (3)

The *complement* of a set A, denoted A' is everything not in A. If U is the universal set then

$$A' = \{x | x \in U \wedge x \notin A\}$$

The De Morgan laws for sets are:

$$(A \cup B)' = A' \cap B'$$
$$(A \cap B)' = A' \cup B'$$

The *Cartesian product* $A \times B$ of two sets A and B is:

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

The elements of $A \times B$ are called *ordered pairs* because $(a, b) = (c, d)$ only if $a = c$ and $b = d$.

Relations and Functions

A *relation* R on two sets A and B is a set of ordered pairs, $A \times B$, where A is the *domain* of R and B is the *codomain* of R.
If $x \in A$ and $y \in B$, then xRy is true if $(x, y) \in R$.

A *function* is a special kind of relationship in which an element of the domain to related to just one element of the codomain.

A function $f : A \to B$ relates an element $x \in A$ to an element $y \in B$ where $y = f(x)$. If $f(x)$ is defined for all $x \in A$ then the function is said to be *total*. If $f(x)$ is not defined for some $x \in A$ then the function is said to be *partial*. The set of values $f(x)$ is called the *range*, which is a subset of the codomain.

Relations and Functions (2)

A function $f : A \to B$ is said to be *one-to-one* if it never assigns the same value to two different elements of its domain. A function $f : A \to B$ is said to be *onto* if it its range is the same as its codomain. A function that is *one-to-one* and *onto* is called a *bijection*.

We will be interested in a special kind of relation, called an *equivalence relation*. A relation R on a set A is an *equivalence relation* if it satisfies the following conditions:

1. R is *reflexive*, i.e. $\forall x \in A, xRx$
2. R is *symmetric*, i.e. $\forall x, y \in A$, if xRy, then yRx
3. R is *transitive*, i.e. $\forall x, y, z \in A$, if xRy and yRz, then xRz

For an equivalence relation R on a set A, and an element $x \in A$, the *equivalent class containing* x is

$$[x]_R = \{y \in A | yRx\}$$

Proofs

During this module we will use a few different proof techniques. All proofs use reasoning based on logic to derive some statement from initial facts, assumptions, hypotheses or statements that have been previously proven.

Proof by Contrapostive
Remember that $p \to q \equiv \neg q \to \neg p$.

Example: $\forall i, j$ and n, if $ij = n$ then $i \leq \sqrt{n}$ or $j \leq \sqrt{n}$.

Rather than trying to prove that $ij = n \to (i \leq \sqrt{n} \vee j \leq \sqrt{n})$ we will prove $\neg(i \leq \sqrt{n} \vee j \leq \sqrt{n}) \to ij \neq n$.

$\neg(i \leq \sqrt{n} \vee j \leq \sqrt{n}) \equiv \neg(i \leq \sqrt{n}) \wedge \neg(j \leq \sqrt{n})$ by De Morgans law
$\equiv (i > \sqrt{n}) \wedge (j > \sqrt{n})$
Therefore $ij > \sqrt{n}\sqrt{n}$
$\equiv ij > n \equiv ij \neq n$

Proofs (2)

Proof by Contradiction

A variant of *proof by contrapositive* is *proof by contradiction*. Every proposition p is equivalent to the proposition $true \rightarrow p$. Its contrapositive is $\neg p \rightarrow false$. *Proof by contradiction* works by assuming p is false and deriving the statement *false* (i.e. deriving its contradiction).

Example: $\forall m, n \in \mathcal{N},\ m/n \neq \sqrt{2}$.

Assume the contradiction of the proposition, i.e. $\exists m, n \in \mathcal{N}$ such that $m/n = \sqrt{2}$.
Dividing out the common factors of m and n yields $p/q = \sqrt{2}$ where p and q have no common factors.
$p/q = \sqrt{2} \equiv p = q\sqrt{2} \equiv p^2 = 2q^2$.
Since p^2 is even, then p must be even (aside: can you prove that the product of two even numbers must be even?), so $p = 2r, r \in \mathcal{N}$

Proofs (3)

Since $p = 2r$ then $p^2 = 4r^2$ and
$p^2 = 2q^2 \equiv 4r^2 = 2q^2 \equiv 2r^2 = q^2$.

Hence q^2 and q are even.If p and q are even they must have a common factor, 2, but based on the contradiction of the proposition we derived that p and q had no common factors. The contradiction of the proposition has given rise to a *false* statement, and hence the proposition $\forall m, n \in \mathcal{N},\ m/n \neq \sqrt{2}$ is *true*.

Proof by Cases

If we want to prove P and $P_1, P_2, \ldots, P_n$ are propositions of which at least one must be *true* then we can prove P by proving that $P_i \rightarrow P, \forall i$.

$$
\begin{aligned}
(P_1 \rightarrow P) \wedge (P_2 \rightarrow P) \wedge \ldots (P_n \rightarrow P) &\Leftrightarrow (P_1 \vee P_2 \vee \ldots \vee P_n) \rightarrow P \\
&\Leftrightarrow true \rightarrow P \\
&\Leftrightarrow P
\end{aligned}
$$

Proofs (4)

Proof by Structural Induction

When dealing with recursive definitions *proof by structural induction* is a common approach. Given an object O which comprises base elements, $a_1, a_2, \ldots, a_n$, and operations $o_1, o_2, \ldots, o_m$ such that

- $a_i \in O, 1 \leq i \leq n$
- $\forall x_1, x_2, \ldots, x_p \in O, op_j(x_1, x_2, \ldots, x_p) \in O, 1 \leq j \leq m$

the principle of *structural induction* states that to prove $P(x)$ is *true* for every x, it is sufficient to prove

- $P(a_i)$ is *true*, $1 \leq i \leq n$ (*basis statement*)
- $\forall x_1, x_2, \ldots, x_p \in O$, if $P(x_j)$ is *true*, $1 \leq k \leq p$ then $P(op_j(x_1, x_2, \ldots, x_p)$ is *true*)(*induction step*)

Proofs (5)

Example:

The language *Expr* is defined as:

- $a \in Expr$
- $\forall x, y \in Expr, x + y, x * y \in Expr$

Prove every element of *Expr* has odd length.

Using *structural induction* this translates into

- $|a|$ is odd
- $\forall x, y \in Expr$ if $|x|$ and $|y|$ are odd, then $|x + y|$ and $|x * y|$ are odd

A number n is odd if $\exists j \in \mathcal{Z}$ such that $n = 2j + 1$

Base step: $|a|$ is odd since $|a| = 1$.

Induction step: Assuming $|x| = 2g + 1$ and $|y| = 2h + 1$ (*induction hypothesis*, $|x|$ and $|y|$ are odd) then $|x + y|$ and $|x * y|$ are $(2g + 1) + 1 + (2h + 1) = 2(g + h + 1) + 1 = 2k + 1$ are odd, where $k = g + h + 1$.

Languages

This module addresses the 2 central questions in this module, namely,

1. What problems are computable?
2. Which problems have efficient algorithmic solution?

To do this we will need:

1. A way to describe problems without specifying how they are solved.
2. An abstract model of a "computer" that is independent of any technological implementation.

The most fundamental type of problem is a *decision problem*, a problem to which the answer is either **yes** or **no**.
Other types of problems, such as calculating a function or manipulating a data structure, have a *decision problem* at their core. So how do we represent a *decision problem* without specifying how the problem is solved?

Languages (2)

For any given *decision problem*,Π, we can partition the set of inputs into 2 sets:

1. Input values that result in the answer **yes**, Y_Π.
2. Input values that result in the answer **no**, N_Π.

Let's focus on the set of input values that result in the answer **yes**, Y_Π. We can think of each element of Y_Π as a word in a language, L_Π, and the complete set Y_Π as the complete language, L_Π.

A machine that then determines if a word belongs to the language L_Π is then said to *accept* the language and hence solve the related decision problem, Π.

CA320 - Computability & Complexity

Models of Computation - Turing Machines

Dr. David Sinclair

The Search for a General Model of Computation

The language $SimplPal = \{xcx^r | x \in \{a, b\}\}$ cannot be accepted by a finite automaton but it can be accepted by a push-down automaton (PDA).

A PDA cannot accept either $AnBnCn = \{a^n b^n c^n | n \in \mathcal{N}\}$ or $L = \{xcx | x \in \{a, b\}\}$.

- A stack is not sufficient.
- A finite automaton with *2 stacks* could accept $AnBnCn$.
- A finite automaton with a *queue* could accept L.

Either adding a queue or 2 stacks to a finite automaton significantly enhances the computational power of the device and either could be the bases for a general model of computation.

Alan Turing

Alan Turing (1912-1954) proposed an abstract model of computation in 1936. Though his model was a purely theoretical one, its principles and features predated many of the features of modern computers.

Turing's model was developed by considering how "human computers" at the time worked with paper and pencil.

- The data written on the paper are symbols from a fixed finite alphabet.
- A human computer's next action could only depend on the symbol currently being examined and their "state of mind" at that instant. This could result is a new "state of mind".
- Only a finite number of "states of mind" are possible.

These ideas predated the finite automaton and PDA models.

Turing Machine

A *Turing machine* is similar to a *linear bounded automaton* (LBA) except that the tape is "semi-infinite".

- The tape is composed of sequence of cells, where each cell holds one symbol (if no symbol is written to a cell it contains a *blank* symbol).
- The tape has a start cell (the left-most cell). This cell contains the "[" symbol which cannot be overwritten.
- The tape extends infinitely to the right.
- The *read-write head* is centered over one cell at a time and can move one cell to the left or right.

Turing Machine (2)

At each step the Turing machine reads the symbol under the read-write head, replaces the by another symbol (could be the same symbol) and then performs one of three possible actions $\mathcal{A} \in \{L, R, S\}$, where:

- L denotes "Left", move the read-write head one cell to the left.
- R denotes "Right", move the read-write head one cell to the right.
- S denotes "Stationary", read-write head does not move.

Because the tape is "semi-infinite" this introduces a new issue to be considered. The Turing machine may not halt in either an *accepting state* or a *rejecting state* but continue to process the tape!

Turing Machine (3)

A Turing machine (TM) is a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$ where:

- Q is a finite set of states. Two additional states h_a and h_r, the accepting and rejecting states, are not elements of Q.
- Σ is the finite input alphabet.
- Γ is the finite tape alphabet. $\Gamma \supseteq \Sigma$. The *blank* symbol, Δ is not an element of Γ.
- $q_0 \in Q$ is the initial state.
- $\delta : Q \times (\Gamma \cup \{[, \Delta\}) \rightarrow (Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{[, \Delta\}) \times \{R, L, S\}$ is the *transition function*.

If $((q, \sigma), (q', \psi, D)) \in \delta$ then when in state q with σ at the current read-write head position, M will replace σ by ψ, move in direction D and enter state q'. If q' is either h_a or h_r the transition causes T to halt. Executing a L move while the current cell contains "[" will cause T to enter h_r.

Turing Machine (4)

A *configuration* of a Turing machine T is defined as:

$$xqy$$

where q is the current state of T, x are the symbols to the left of read-write head on the tape, the string y is either null or starts under the read-write head and everything after xy on the tape is blank.

A *move* of T is when T changes from one configuration to another.

$$xqy \vdash zrw$$

Turing Machine as a Language Acceptor

A Turing machine T accepts w if

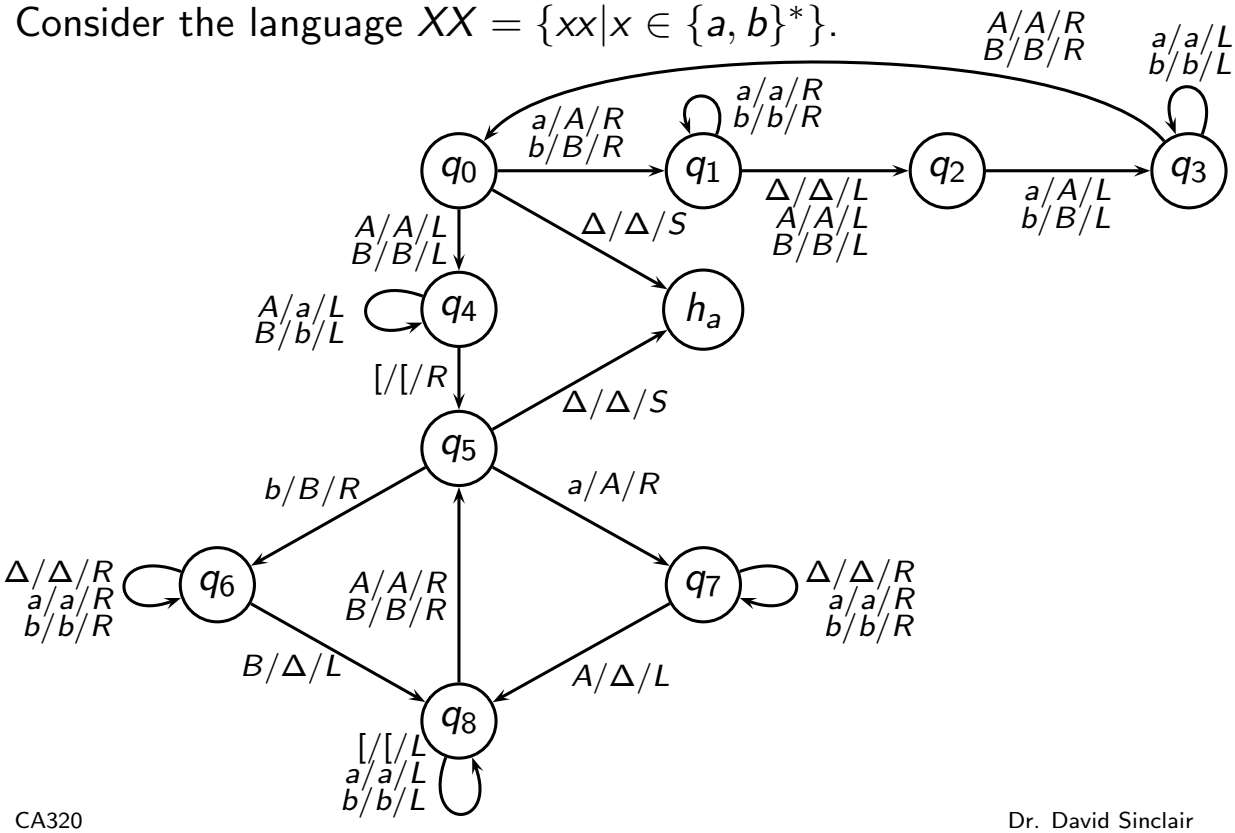
$$[q_0 w \vdash^* x h_a y$$

Let $L = L(T)$ be the language accepted by T. If $x \in L$, T halts in the accepting state h_a. If $x \notin L$, then there are 2 possibilities:

1. T rejects x.
2. T does not halt.

Example 1: TM as a Language Acceptor

Consider the language $XX = \{xx | x \in \{a, b\}^*\}$.



Example 1: TM as a Language Acceptor (2)

We assume that if a transition is not given then the Turing machine moves to the rejecting state h_r.

We can trace the operation of this Turing machine for a few examples.

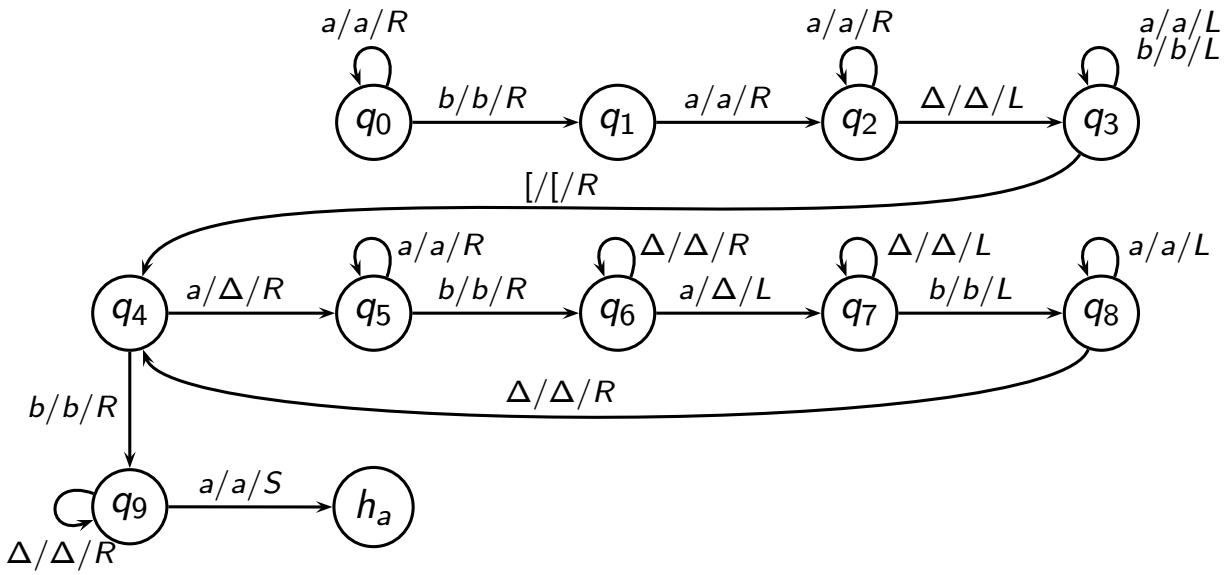
$[q_0aba \vdash [Aq_1ba \vdash^* [Abaq_1\Delta \vdash [Abq_2a \vdash [Aq_3bA$
$\vdash [q_3AbA \vdash [Aq_0bA \vdash [ABq_1A \vdash [Aq_2BA$
$\vdash [Ah_rBA$

$[q_0ab \vdash [Aq_1b \vdash [Abq_1\Delta \vdash [Aq_2b\Delta \vdash [q_3AB$
$\vdash [Aq_0B \vdash [q_4AB \vdash q_4[aB \vdash [q_5aB$
$\vdash [Aq_7B \vdash [Ah_rB$

$[q_0aa \vdash [Aq_1a \vdash [Aaq_1\Delta \vdash [Aq_2a\Delta \vdash [q_3AA$
$\vdash [Aq_0A \vdash [q_4AA \vdash q_4[aA \vdash [q_5aA$
$\vdash [Aq_7A \vdash [q_8A\Delta \vdash [Aq_5\Delta \vdash [Ah_a\Delta$

Example 2: TM as a Language Acceptor

Consider the language $L_{ex2} = \{a^i b a^j | 0 \leq i < j\}$.



Example 2: TM as a Language Acceptor (2)

Consider the input *abaa*

$$
\begin{array}{llll}
[q_0abaq & \vdash [aq_0baa & \vdash [abq_1aa & \vdash [abaq_2a & \vdash [abaaq_2\Delta \\
 & \vdash [abaq_3a & \vdash^* q_3[abaa & \vdash [q_4abaa & \vdash [q_5\Delta baa \\
 & \vdash [\Delta bq_6aa & \vdash [\Delta q_7b\Delta a & \vdash [q_8\Delta b\Delta a & \vdash [\Delta q_4b\Delta a \\
 & \vdash [\Delta bq_9\Delta a & \vdash [\Delta b\Delta q_9a & \vdash [\Delta b\Delta h_aa &
\end{array}
$$

What happens when the input is *aba*? Try this as an exercise.

- Will it halt?
- If it doesn't is that a problem if $aba \notin L_{ex2}$?

TM that Computes Partial Functions

Let $T = (Q, \Sigma, \Gamma, q_0, \delta)$ be a Turing machine, k a natural number and f a partial function on $(\Sigma^*)^k$ with values in Γ^*. We say that T computes f if for every $(x_1, x_2, \ldots, x_k)$ in the domain of f

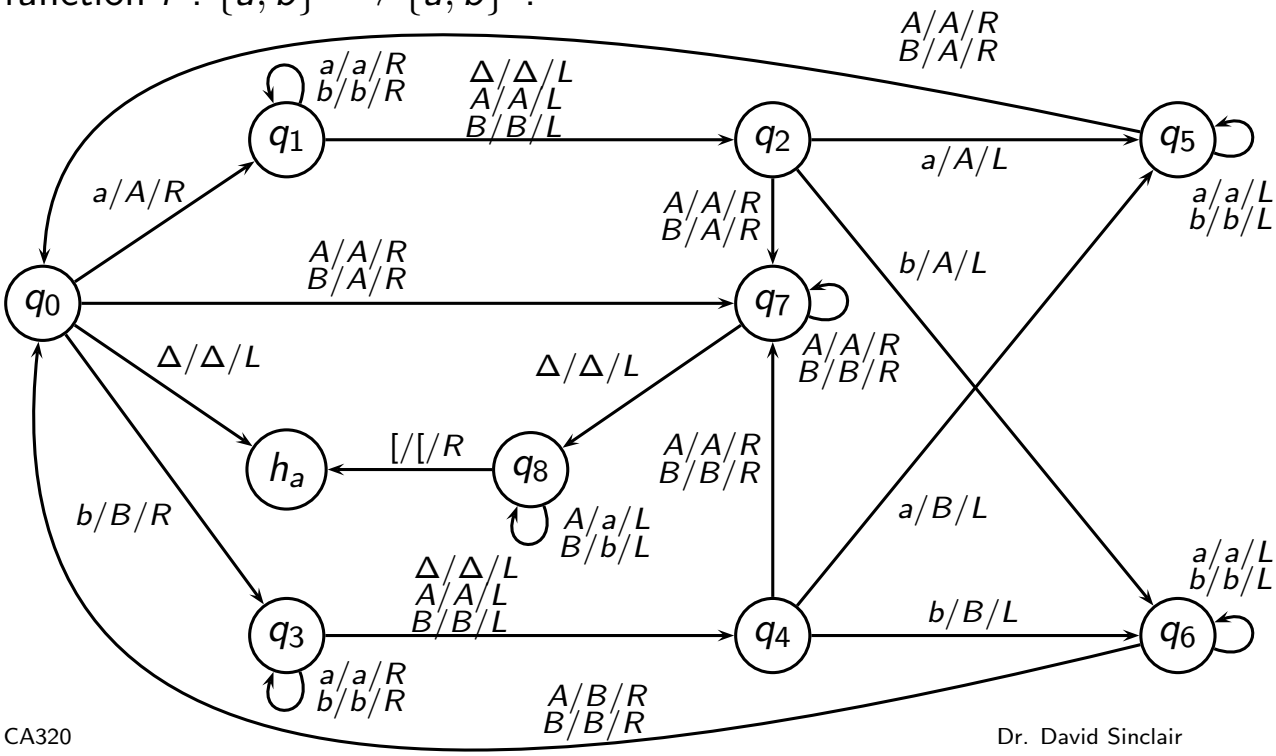
$$[q_0 x_1 \Delta x_2 \Delta \ldots \Delta x_k \vdash^* [h_a f x_1, x_2, \ldots, x_k)$$

and no other input that is a k-tuple of strings is accepted by T.

A partial function $f : (\Sigma^*)^k \to \Gamma^*$ is Turing-computable, or simply computable, if there is a Turing machine that computes f.

Example: TM that Computes Partial Functions

The following diagram is the transition diagram for the reverse function $r : \{a, b\}^* \to \{a, b\}^*$.



Example: TM that Computes Partial Functions (2)

Consider the string *baba*.

$$
\begin{array}{llll}
[q_0baba & \vdash [Bq_3aba & \vdash^* [Babaq_3\Delta & \vdash [Babq_4a & \vdash [Baq_5bB \\
& \vdash^* [q_5BabB & \vdash [Aq_0abB & \vdash [AAq_1bB & \vdash [AAbq_1B \\
& \vdash [AAq_2bB & \vdash [Aq_6AAB & \vdash [ABq_0AB & \vdash [ABAq_7B \\
& \vdash [ABABq_7\Delta & \vdash [ABAq_8B & \vdash^* q_8[abab & \vdash [h_aabab
\end{array}
$$

Functions on Natural Numbers

Represent numbers in unary notation using only the symbol 1 (zero is represented by the empty string).

$f : \mathcal{N} \rightarrow \mathcal{N}$ is computed by M if M computes $f' : \{1\}^* \rightarrow \{1\}^*$ where $f'(1^n) = 1^{f(n)}$, $\forall n \in \mathcal{N}$.

Example: $f(n) = n + 1$, $\forall n \in \mathcal{N}$.

State	Symbol	$\delta(State, Symbol)$
q_0	1	$(q_0, 1, R)$
q_0	Δ	$(q_1, 1, L)$
q_1	1	$(q_1, 1, L)$
q_1	[	$(h_a, [, R)$

$$[q_011 \vdash [1q_01 \vdash [11q_0\Delta \vdash [1q_111 \vdash [q_1111 \vdash [h_a111$$

Functions on Natural Numbers (2)

The following Turing machine computes the addition of 2 unary numbers, m and n, where the initial configuration is $[q_0 m \Delta n$.

State	Symbol	$\delta(State, Symbol, Action)$
q_0	1	$(q_0, 1, R)$
q_0	Δ	(q_1, Δ, R)
q_1	1	$(q_1, 1, R)$
q_1	Δ	(q_2, Δ, L)
q_2	1	(q_3, Δ, L)
q_2	Δ	(q_3, Δ, L)
q_3	1	$(q_3, 1, L)$
q_3	Δ	$(q_3, 1, L)$
q_3	[	$(h_a, [, R)$

Recursive Languages

A language $L \subseteq \Sigma^*$ is *recursive* (also called *Turing-decidable*) iff the characteristic function $\chi_L : \Sigma^* \rightarrow 0, 1$ is Turing-computable, where $\forall w \in \Sigma^*$,

$$\chi_L(w) = \begin{cases} 1, & \text{if } w \in L \\ 0, & \text{otherwise} \end{cases}$$

For example, let $\Sigma = \{a\}$ and $L = \{w \in \Sigma^* | \ |w| \text{ is even}\}$.

The Turing machine that calculates the characteristic function χ_L scans w symbols by symbol from left to right and uses 2 states to determine if the number of symbols it has scanned so far is odd or even. When a blank is reached, it uses its states to remember if the number of symbols was odd or even, moves from right to left, overwriting the contents of the tape and once the start of tape symbol has been found writes a '1' or '0' to the tape.

Recursively Enumerable Languages

A Turing machine M accepts a string w if M halts on the input w.

M accepts a language L iff M halts on w iff $w \in L$.

A language is *recursively enumerable* (also called *Turing-acceptable* or *semi-decidable*) if there is some Turing machine that accepts it.

For example, let $\Sigma = \{a, b\}$ and $L = \{w \in \Sigma^* | \ w$ contains at least one $a\}$.

State	Symbol	$\delta(State, Symbol)$
q_0	a	(h_a, a, S)
q_0	b	(q_0, b, R)

Every recursive language is recursively enumerable.

Combining Turing Machines

Two Turing machine computations M_1 and M_2 can be combined into a larger machine:

- M_1 prepares string as input to M_2.
- M_1 passes control to M_2 with the read-write head at the start of the input.
- M_1 retrieves control when M_2 has completed its computation.

Some basic Turing machines are:

- *Symbol-writing machine* M_a for each symbol $a \in \Sigma$.
- *Head-moving machines* R and L that move the read-write head right and left respectively.

Combining Turing Machines (2)

These basic machines can be combined to perform more complex operations:

- If the current symbol $a = b$, then do M_1 else do M_2.
  IF $(a = b,\ M_1,\ M_2)$
- Move read-write head right until a blank is found.
  R_Δ
- Move read-write head left until a blank is found.
  L_Δ
- Copy machine, transform $w\Delta$ to $w\Delta w\Delta$.
  C = IF $(a = \Delta, R_\Delta, M_\Delta R_\Delta R_\Delta M_a L_\Delta L_\Delta M_a RC)$
- Shift machine, transform $\Delta w\Delta$ to $w\Delta$.
  S = IF$(a = \Delta,\ L,\ M_\Delta L M_a RRS)$

Extensions

The following extensions **do not** increase the power of Turing machines.

- 2-way infinite tape
- Multiple tapes
- Multiple read-write heads on one tape
- 2-dimensional tape
- Nondeterminism

Computable Functions

In order to characterise those functions from $\mathcal{N}$ to $\mathcal{N}$ that can be computed we need to define some *basic* functions.

- $\text{zero}_k(n_1, \ldots, n_k) = 0$
- $\text{succ}(n) = n + 1,\ \forall n \in \mathcal{N}$
- $p_i(n_1, \ldots, n_k) = n_i\ \forall 1 \leq i \leq n$

and a way of *composing* existing functions to define new functions:

$$f(x) = h(g_1(x), \ldots, g_m(x))$$

Primitive Recursion

We can construct a *primitive recursive* function f from existing functions h and g:

$$\begin{aligned} f(x, 0) &= h(x) \\ f(x, \text{succ}(y)) &= g(x, y, f(x, y)) \end{aligned}$$

For example,

$$\begin{aligned} \text{plus}(m, 0) &= m \\ \text{plus}(m, \text{succ}(n)) &= \text{succ}(\text{plus}(m, n)) \end{aligned}$$

$$\begin{aligned} \text{mult}(m, 0) &= \text{zero}(m) \\ \text{mult}(m, \text{succ}(n)) &= \text{plus}(m, \text{mult}(m, n)) \end{aligned}$$

All primitive recursive functions are *total* recursive functions.

Primitive Recursion (2)

Not all computable functions from $\mathcal{N}$ to $\mathcal{N}$ are primitive recursive. The Ackerman function is an example of a computable function that is not primitive recursive.

$$\begin{aligned} \text{Ack}(0, n) &= \text{succ}(n) \\ \text{Ack}(\text{succ}(m), 0) &= \text{Ack}(m, 1) \\ \text{Ack}(\text{succ}(m), \text{succ}(n)) &= \text{Ack}(m, \text{Ack}(\text{succ}(m), n)) \end{aligned}$$

μ-Recursion

For a predicate P, the *unbounded minimisation of* P is the function f defined as follows.

$$f(x) = \min\{y | P(x, y) \text{ is true}\}$$

i.e. the least value of y that satisfies $P(x, y)$.
This is denoted as:

$$f(x) = \mu y[P(x, y)]$$

functions defined in this way are called μ-*recursive* functions.
For example:

$$\text{minus}(x, y) = \mu z[y + z = x]$$

All partial recursive functions are μ-recursive functions.

Theorem

A function is μ-recursive iff it is computable.

Gödelisation

We have seen how we can convert numbers to strings by using unary notation.

We can convert strings to unique numbers by assigning each character in Σ a unique number. The i^{th} character is assigned the value of the i^{th} prime number.

Denoting the i^{th} prime number as p_i, then the string $S = s_1, s_2, \ldots, s_k$ is represented as:

$$g(s_1, s_2, \ldots, s_k) = p_1^{i_1} p_2^{i_2} \ldots p_k^{i_k}$$

So CAT is $2^3 3^1 5^{20}$.

We can use μ-recursive functions to compute functions from strings to strings.

Universal Turing Machine

All the Turing machines we have looked at so far have been specific to a given algorithm. Each new algorithm requires a new Turing machine. Turing in his 1936 paper also describe the *Universal Turing machine* that could execute any algorithm provided it received a description of the algorithm and the data the algorithm is to operate on. The description of the algorithm is in terms of the Turing machine that implements the algorithm.

A Universal Turing machine T_u receives its input as a string (on the tape). The form of the string is $e(T)e(z)$ where T is an arbitrary Turing machine, z is its input and e is an encoding function whose values are strings in $\{0,1\}^*$. The computation performed by T_u on $e(T)e(z)$ will:

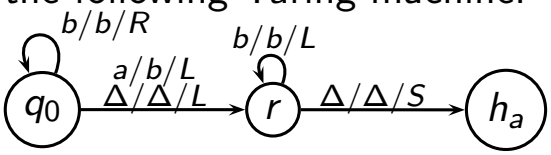
1. accept $e(T)e(z)$ iff T accepts z; and
2. if T accepts z and produces y, then T_u produces $e(y)$.

Universal Turing Machine (2)

Each state, symbol and read-write head direction is assigned a non-zero number:

- $n(h_a) = 1$, $n(h_r) = 2$, $n(q_0) = 3$ and all other states are assigned distinct numbers
- Each symbol a_i, including Δ, is assigned a number $n(a_i)$
- $n(R) = 1$, $n(L) = 2$ and $n(S) = 3$

Consider the following Turing machine:



Universal Turing Machine (3)

We can encode this as follows:

$$n(q_0) = 3, n(r) = 4, n(\Delta) = 1, n(a) = 2 \text{ and } n(b) = 3$$

and then the move $m = \delta(q_0, \Delta) = (r, \Delta, L)$ would be encoded as

$$e(m) = 1^3 0 1 0 1 4 0 1 0 1 2 0 = 1110101111010110$$

and the complete Turing machine T would be

$$\begin{array}{lll} e(T) & = & 1110111011101110 \quad 1110101111010110 \\ & & 11110111011111011110 \quad 111101010101110 \end{array}$$

Church-Turing Thesis

The Church-Turing thesis states that the Turing machine is a general model of computing, that is, that **any** algorithmic procedure that can be carried out, be it by a human computer, a team of human computers or electronic computer, can be carried out by a Turing machine. It was first formulated by Alonzo Church but has never been mathematically proven (mainly because of the difficulty in precisely defining what is meant by an *algorithmic procedure*).

However there is substantial evidence for the thesis, so much evidence that the thesis is generally deemed to be true.

Church-Turing Thesis (2)

Some of the evidence for the Church-Turing thesis is:

1. The nature of the model makes it seem likely that all the crucial step to a computation can be carried out by a Turing machine.
2. All the various enhancements to the Turing machine model, though they may make the model more efficient, have not increased the computational power of the Turing machine.
3. Other theoretical models have been proposed but all have been shown to be equivalent to the Turing machine.
4. Since the conception of the Turing machine model no one has suggested ant type of computation that should be considered as an algorithmic procedure that cannot be implemented by a Turing machine.

CA320 - Computability & Complexity

Regular Languages

Dr. David Sinclair

Regular Languages

The set of *regular languages* $\mathcal{R}$ over an alphabet Σ is defined as:

- The language $\emptyset$ is an element of $\mathcal{R}$.
- $\forall a \in \Sigma$, the language $\{a\}$ is in $\mathcal{R}$.
- $\forall L_1, L_2 \in \mathcal{R}$, then the following languages are in $\mathcal{R}$.
  - $L_1 \cup L_2$
  - $L_1 L_2$
  - L_1^*

Regular Expressions

Every *regular language* can be represented by a *regular expressions*, and every *regular expression* represents a *regular language*.

- $\emptyset$ and ϵ are regular expressions.
- $\forall a \in \Sigma, a$ is a regular expression.
- If R_1, R_2 are regular expressions, them the following are regular expressions in order of precedence with the first having the highest precedence.

(R_1)	parentheses
R_1^*	closure
$R_1 R_2$	concatenation
$R_1 + R_2$	alternation

Given regular expression R, $L(R)$ stands for the language represented by R.

Regular Expressions (2)

Some Examples:

Regular Language	Corresponding Regular Expression
$\emptyset$	$\emptyset$
$\{a\}$	a
$\{a, b\}^*$	$(a + b)^*$
$\{aab\}^* a, ab$	$(aab)^*(a + ab)$
$(\{aa, bb\} \cup \{ab, ba\}\{aa, bb\}^*\{ab, ba\})^*$	$(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$

Some properties of regular expressions.

- $R^* = R^* R^* = (R^*)^* = R + R^*$
- $R_1(R_2 R_1)^* = (R_1 R_2)^* R_1$
- $(R_1^* R_2)^* = \epsilon + (R_1 + R_2)^* R_2$
- $(R_1 R_2^*)^* = \epsilon + R_1(R_1 + R_2)^*$

Some More Examples

Let $\Sigma = \{a, b\}$.

Regular Expression	Language	Comments
$(a + b)(a + b)$	$\{aa, ab, ba, bb\}$	$(a+b)(a+b) = aa+ab+ba+bb$
$(a + b)^*$	all strings of a's and b's *including* ϵ	$(a + b)^* = (a^*b^*)^*$
$a + a^*b$	$\{a, b, ab, aab, aaab, \ldots\}$	note order of precedence
$b^*ab^*(ab^*ab^*)^*$	string with an odd number of a's	Other valid expressions are $b^*a(b + ab^*a)^*$ or $(b + ab^*a)^*ab^*$

Deterministic Finite Automata

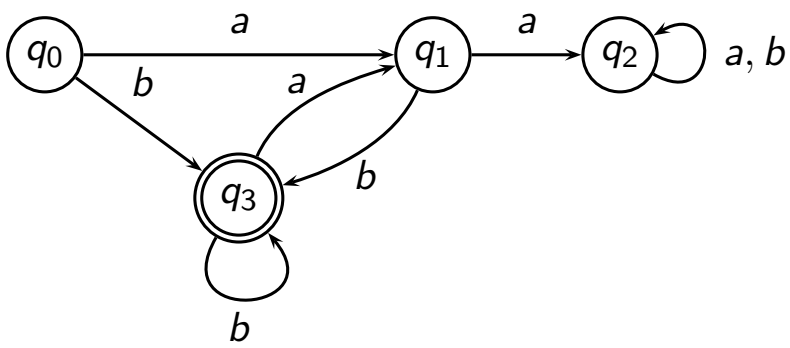
A *deterministic finite automata* (DFA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$ where

- Q is a **finite** set of *states*;
- Σ is a **finite** *input alphabet*;
- $q_0 \in Q$ is the *initial state*;
- $A \subseteq Q$ is the set of *accepting states*;
- $\delta : Q \times \Sigma \to Q$ is the *transition function*.

For any element $q \in Q$ and any symbol $\sigma \in \Sigma$, $\delta(q, \sigma)$ is the state the DFA moves to if it receives input σ while in state q.

Deterministic Finite Automata (2)

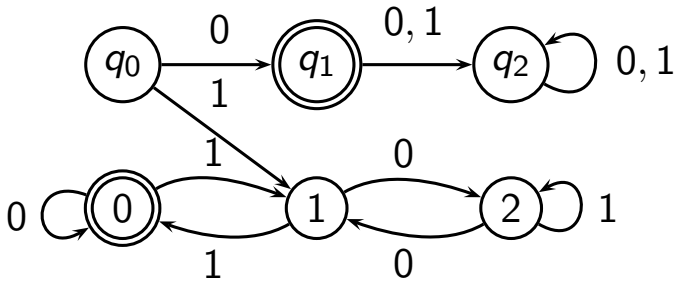
Here is the DFA that accepts strings ending in b but does not contain aa.



What is the corresponding regular expression?

Deterministic Finite Automata (3)

Here is a DFA that accepts binary number that are divisible by 3. Adding a 0 onto a binary number x is doubling it. Adding a 1 onto a binary number x is doubling it and adding 1. The same happens to remainder, though if the remainder is greater than 3 we need to do an additional *mod* 3 operations.
The states labelled 0,1 and 2 correspond to states in which the x *mod* 3 is equal to 0,1 and 2 respectively.



What sort of binary strings will it not accept?

Deterministic Finite Automata (4)

The *extended transition function* $\delta^* : Q \times \Sigma^* \to Q$ is defined as:

- for every $q \in Q, \delta^*(q, \epsilon) = q$
- for every $q \in Q$, every $y \in \Sigma^*$ and every $\sigma \in \Sigma$
$$\delta^*(q, y\sigma) = \delta(\delta^*(q, y), \sigma)$$

Let $M = (Q, \Sigma, q_0, A, \delta)$ be a finite automata and let $x \in \Sigma^*$. The string x is *accepted by* M if

$$\delta^*(q_0, x) \in A$$

and is *rejected by* M otherwise.
The *language* accepted by M is the set

$$L(M) = \{x \in \Sigma | x \text{ is accepted by } M\}$$

If L is a language over Σ, L is accepted by M if and only if $L = L(M)$.

Nondeterministic Finite Automata

A *nondeterministic finite automaton* (NFA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$ where

- Q is a **finite** set of *states*;
- Σ is a **finite** *input alphabet*;
- $q_0 \in Q$ is the *initial state*;
- $A \subseteq Q$ is the set of *accepting states*;
- $\delta : Q \times (\Sigma \cup \{\Lambda\}) \to 2^Q$ is the *transition function*.

For any element $q \in Q$ and any symbol $\sigma \in \Sigma \cup \{\Lambda\}$, ($\Lambda$ is the null symbol), $\delta(q, \sigma)$ is the *set of states* the NFA moves to if it receives input σ while in state q.

Nondeterministic Finite Automata (2)

Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA and $S \subseteq Q$ be a set of states. The Λ-*closure of* S is the set $\Lambda(S)$ and is defined as

- $S \subseteq \Lambda(S)$
- $\forall q \in \Lambda(S), \delta(q, \Lambda) \subseteq \Lambda(S)$

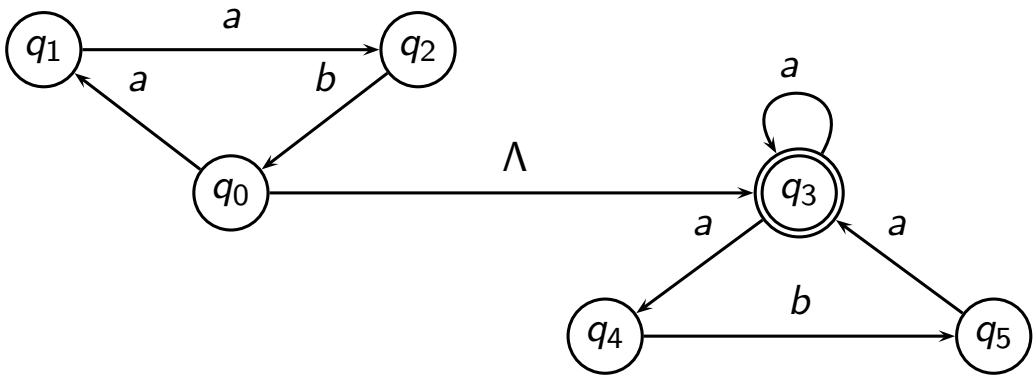
The *extended transition function* for an NFA, $\delta^* : Q \times \Sigma^* \rightarrow 2^Q$ is defined as

- $\forall q \in Q, \delta^*(q, \Lambda) = \Lambda(\{q\})$
- $\forall q \in Q, y \in \Sigma^*, \sigma \in \Sigma,$
$$\delta^*(q, y\sigma) = \Lambda(\bigcup\{\delta(p, \sigma) | p \in \delta^*(q, y)\})$$

A string $x \in \Sigma^*$ is *accepted by* M if $\delta^*(q_0, x) \cap A \neq \emptyset$.
The language $L(M)$ accepted by M is the set of all strings accepted by M.

Nondeterministic Finite Automata (3)

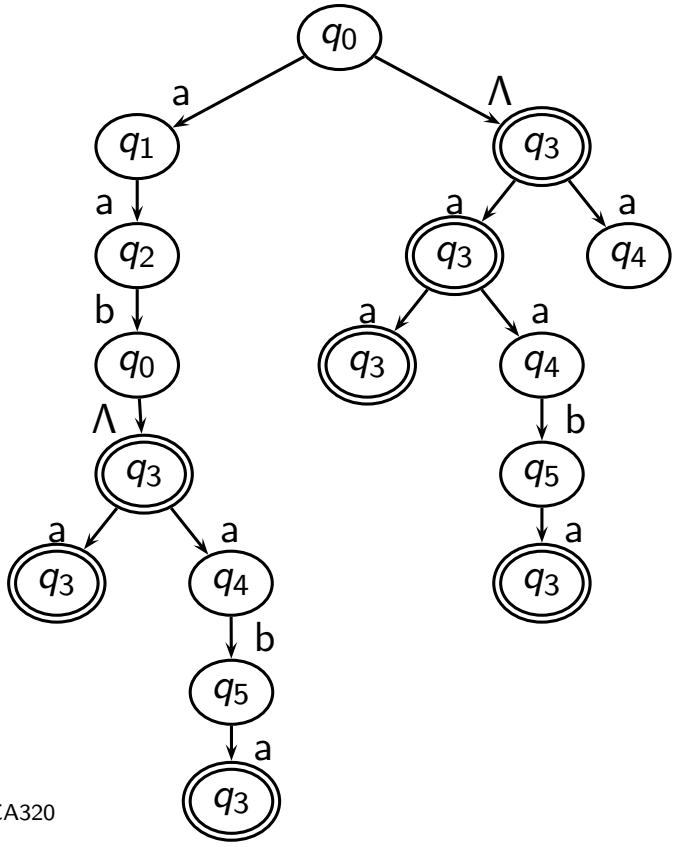
The concept of acceptance for an NFA is quite different than that corresponding concept for a DFA.
Consider the language $\{aab\}^*\{a, aba\}^*$. An NFA that accepts this language is:



Consider how this NFA would process the string *aababa*.

Nondeterministic Finite Automata (4)

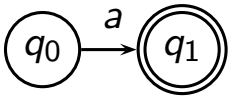
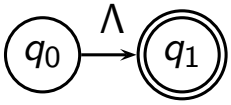
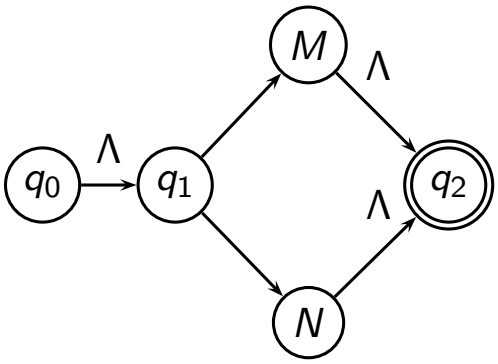
We can represent this by a *computation tree*.



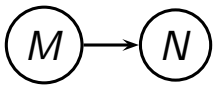
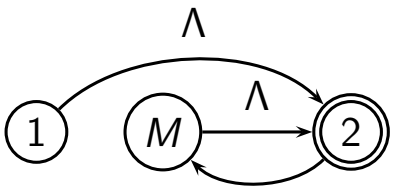
input: $\epsilon aababa$

Regular Expressions to NFA

It is straightforward to convert a regular expression into a nondeterministic finite automaton.

Regular Expression	NFA
a	
ϵ	
$M + N$	

Regular Expressions to NFA [2]

Regular Expression	NFA
MN	
M^*	

Converting an NFA to a DFA

While NFAs are nice theoretical devices we do not know how to build such a device. DFAs, on the other hand, are devices we can build.

Fortunately we can convert an NFA into a DFA in two steps:

- First remove the Λ transitions.
- Secondly redefine the states so that there is only one possible next state from the current state given any input.

Theorem

$\forall L \subseteq \Sigma^*$ *accepted by an NFA* $M = (Q, \Sigma, q_0, A, \delta)$ *there is an NFA* M_1 *with no* Λ*-transitions that also accepts* L.

Converting an NFA to a DFA (2)

Proof.

$$\text{Let } M_1 = (Q, \Sigma, q_0, A_1, \delta_1)$$

where

$$A_1 = \begin{cases} A \cup \{q_0\} & \text{if } \Lambda \in L \\ A & \text{otherwise} \end{cases}$$

$$\delta_1(q, \sigma) = \delta^*(q, \sigma)$$

We need to prove that $\delta_1^*(q, x) = \delta^*(q, x)$ for $|x| \geq 1$ and this is done by structural induction on x.
If $x = a \in \Sigma$ then by definition $\delta_1(q, x) = \delta^*(q, x)$ and because M_1 has no Λ-transitions $\delta_1(q, x) = \delta_1^*(q, x)$, hence $\delta_1^*(q, x) = \delta^*(q, x)$.

Converting an NFA to a DFA (3)

Proof contd.

Let $x = y\sigma, y \in \Sigma^*, \sigma \in \Sigma$

$$\begin{aligned} \delta_1^*(q, y\sigma) &= \bigcup\{\delta_1(p, \sigma) | p \in \delta_1^*(q, y)\} \\ &= \bigcup\{\delta_1(p, \sigma) | p \in \delta^*(q, y)\} && \text{by induction hypothesis} \\ &= \bigcup\{\delta^*(p, \sigma) | p \in \delta^*(q, y)\} && \text{by definition of } \delta_1 \\ &= \delta^*(q, y\sigma) \end{aligned}$$

Hence $L(M_1) = L(M) = L$.

□

Theorem

$\forall L \in \Sigma^*$ accepted by an NFA $M = (Q, \Sigma, q_0, A, \delta)$ there is a DFA $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ that also accepts L.

Converting an NFA to a DFA (4)

Proof.

The previous theorem means we can remove all Λ-transitions. We can remove the last source of nondeterminism, the multiple next states, by redefining the states as the set of states that can be reached given a specific input symbol.

$Q_1 = 2^Q$
$q_1 = \{q_0\}$
$A_1 = \{q \in Q_1 | q \cap A \neq \emptyset\}$
$\delta_1(q, \sigma) = \bigcup\{\delta(p, \sigma) | p \in q\}$

To prove M and M_1 accept the same languages we need to show $\delta_1^*(q_1, x) = \delta^*(q_0, x), \forall x \in \Sigma^*$. This is done by structural induction on x.

Converting an NFA to a DFA (5)

Proof contd.

If $x = \epsilon$, then

$$\begin{aligned}
\delta_1^*(q_1, x) &= \delta_1^*(q_1, \Lambda) \\
&= q_1 && \text{by definition of } \delta_1^* \\
&= \{q_0\} && \text{by definition of } q_1 \\
&= \delta^*(q_0, \Lambda) && \text{by definition of } \delta^* \\
&= \delta^*(q_0, x)
\end{aligned}$$

If $x = y\sigma$, then

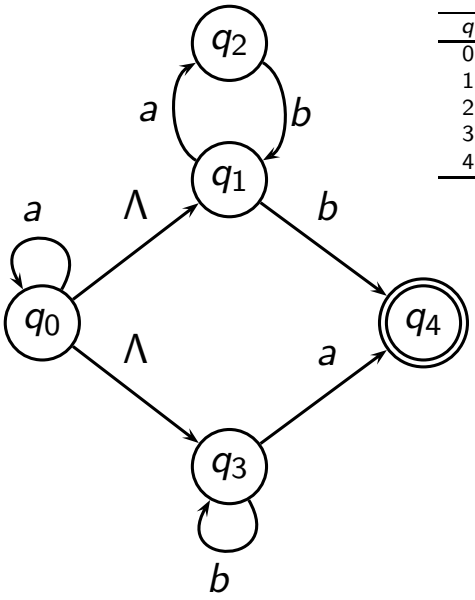
$$\begin{aligned}
\delta_1^*(q_1, y\sigma) &= \delta_1(\delta_1^*(q_1, y), \sigma) && \text{by the definition of } \delta_1^* \\
&= \delta_1(\delta^*(q_0, y), \sigma) && \text{by the induction hypothesis} \\
&= \bigcup\{\delta(p, \sigma) | p \in \delta^*(q_0, y)\} && \text{by definition of } \delta_1 \\
&= \delta^*(q_0, y\sigma) && \text{by the definition of } \delta^*
\end{aligned}$$

Hence $L(M_1) = L(M) = L$.

□

Example NFA to DFA

Consider the following NFA.

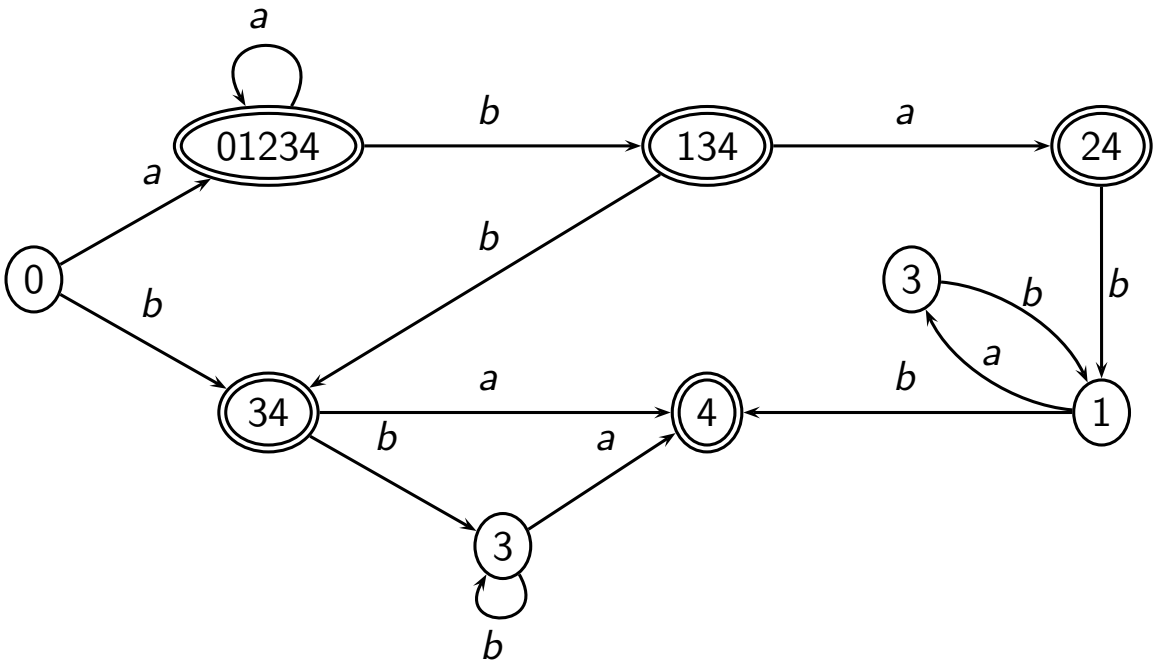


Transition function (in tabular form)

q	$\delta(q,a)$	$\delta(q,b)$	$\delta(q,\Lambda)$	$\delta^*(q,a)$	$\delta^*(q,b)$
0	$\{0\}$	$\emptyset$	$\{1,3\}$	$\{0,1,2,3,4\}$	$\{3,4\}$
1	$\{2\}$	$\{4\}$	$\emptyset$	$\{2\}$	$\{4\}$
2	$\emptyset$	$\{1\}$	$\emptyset$	$\emptyset$	$\{1\}$
3	$\{4\}$	$\{3\}$	$\emptyset$	$\{4\}$	$\{3\}$
4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Example NFA to DFA (2)

This results in the following DFA.



This technique is called *subset construction*.

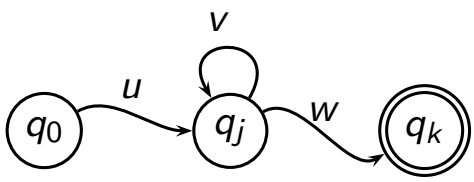
Pumping Lemma

A finite automaton, whether it is deterministic or nondeterministic, has a finite number of states, n. So, can it deal with an input x whose length is longer than n, i.e. $|x| > n$?
The *Pumping Lemma for Regular Languages* describes the conditions that x must adhere to if the automaton is to accept it.

Lemma (Pumping Lemma for Regular Languages)

Let $L \in \Sigma^*$ and $M = (Q, \Sigma, q_0, A, \delta)$ such that $L = L(M)$. If M has n states then for every $x \in L$ satisfying $|x| \geq n$, there are three strings u, v and w such that $x = uvw$ and:

- $|uv| \leq n$
- $|v| > 0$
- $\forall i \geq 0, uv^i w \in L$



Limitations of Regular Languages

Consider the language $AnBn = \{a^n b^n | n \geq 0\}$.

Let's assume there is a finite automaton that accepts $AnBn$.

Let $x = a^n b^n$. Since $x \in AnBn$ and $|x| \geq n$, the Pumping Lemma conditions must apply.

Condition 1	$\|uv\| \leq n$ and since the first n symbols of x are a's then all the symbols of u and v are a.
Condition 2	$v = a^k$ for some $k > 0$.
Condition 3	$uv^i w \in AnBn$ but $uv^i w = a^{n+k} b^n \notin AnBn$. Hence the contradiction implies the initial assumption that there exists a FA that accepts $AnBn$ must be false.

- DFAs cannot count.
- Not all languages are regular languages. In fact $AnBn$ is not a regular language.