

MAT 7003 : Mathematical Foundations

(for Software Engineering)

J Paul Gibson, A207

paul.gibson@it-sudparis.eu

<http://www-public.it-sudparis.eu/~gibson/Teaching/MAT7003/>

Computability

<http://www-public.it-sudparis.eu/~gibson/Teaching/MAT7003/L6-Computability.pdf>

Computable Functions and Computing Machines (Computers)

Alan Turing: 1937 he published a theory of computable functions in: *On Computable Numbers, with an Application to the Entscheidungsproblem*

He reformulated **Kurt Gödel's** 1931 results on the limits of proof and computation, replacing Gödel's universal arithmetic-based formal language with *Turing machines*

He went on to prove that there was no solution to the Entscheidungsproblem by first showing that the halting problem for Turing machines is undecidable: it is not possible to decide, in general, algorithmically whether a given Turing machine will ever halt. (This proof depended on the notion of a Universal machine)

His proof was published subsequent to **Alonzo Church's** equivalent proof in respect to his **lambda calculus** (1936)

In 1999, Time Magazine named **Turing** as one of the 100 Most Important People of the 20th Century for his role in the creation of the modern computer.

Computable Functions and Computing Machines (Computers)

A Turing machine that is able to simulate any other Turing machine is called a *Universal Turing machine* (UTM, or simply a universal machine).

A more mathematically-oriented definition with a similar "universal" nature was introduced by Alonzo Church (and his student Stephen Kleene) at roughly the same time.

Since then, many computational models – including some very simple models - have been shown to be *computationally equivalent* to the Turing machine; such models are said to be *Turing complete*.

Computability is the study of the limits of these machines

Computable Functions and Computing Machines (Computers)

Informally the **Church–Turing thesis** states:

that if an algorithm (a procedure that terminates) exists then there is an equivalent Turing machine, recursively-definable /recursively enumerable function, or applicable λ -function, for that algorithm.

Or

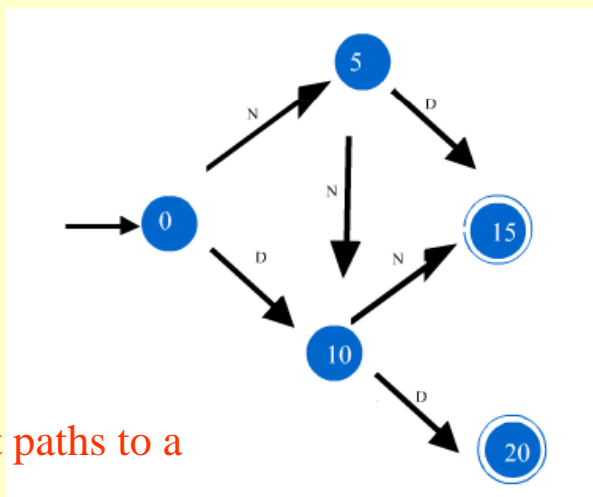
Every effectively calculable function is a computable function, where a function is effectively calculable if its values can be found by some purely mechanical process

Because all the different attempts at formalizing the concept of "effective calculability/computability" have yielded equivalent results, it is now universally accepted that the Church–Turing thesis is correct.

Computable Functions and Computing Machines: **Finite Automata**

Finite automata are computing devices that accept/recognize *regular languages* and are used to model operations of many systems we find in practice.

A classic example is of a vending machine. For example, consider a simple vending machine that accepts only nickels and dimes and requires a payment of 15 cents:



QUESTION: how many different paths to a terminating/accepting state?

Computable Functions and Computing Machines: **Finite Automata**

Definition: deterministic finite automaton (DFA)

Let Q be a finite set and let Σ be a finite set of symbols. Also let δ be a function from $Q \times \Sigma$ to Q , let q_0 be a state in Q and let A be a subset of Q .

We call the elements of Q a **state**, δ the **transition function**, q_0 the **initial state** and A the set of **accepting states**.

Then a **deterministic finite automaton** is a 5-tuple $\langle Q, \Sigma, q_0, \delta, A \rangle$

- The set Q in the above definition is simply a set with a finite number of elements. Its elements can, however, be interpreted as a state that the system (automaton) is in.
- The transition function is also called a **next state function** meaning that the automaton moves into the state (q, a) if it receives the input symbol a while in state q .
- Note that δ is a function. Thus for **each state** q of Q and for **each symbol** a of Σ , $\delta(q, a)$ must be specified.
- If the finite automaton is in an accepting state when the input ceases to come, the sequence of input symbols given to the finite automaton is "accepted"

Computable Functions and Computing Machines: **Finite Automata**

DFAs are often represented by digraphs called (state) **transition diagram**.

- The vertices (denoted by single circles) of a transition diagram represent the states of the DFA
- The arcs labeled with an input symbol correspond to the transitions.
- An arc (p , q) from vertex p to vertex q with label σ represents the transition $\delta(p, \sigma) = q$.
- The accepting states are indicated by double circles.
- Transition functions can also be represented by **transition tables**.

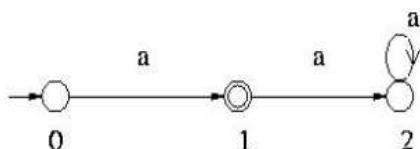
Computable Functions and Computing Machines: **Finite Automata**

Examples of finite automaton

Example 1: $Q = \{ 0, 1, 2 \}$, $\Sigma = \{ a \}$, $A = \{ 1 \}$, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State ($\delta (q, a)$)
0	a	1
1	a	2
2	a	2

A state transition diagram for this DFA is given below.



Computable Functions and Computing Machines: **Finite Automata**

Examples of finite automaton

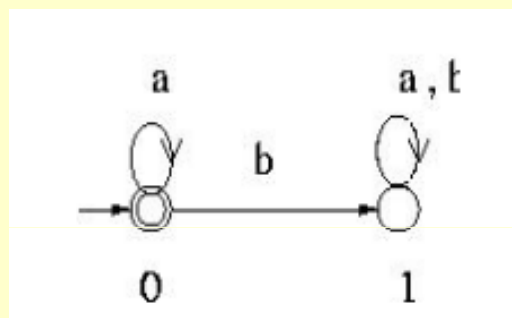
Example 2: $Q = \{ 0, 1, 2 \}$, $\Sigma = \{ a, b \}$, $A = \{ 1 \}$, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State ($\delta (q, a)$)
0	a	1
0	b	2
1	a	2
1	b	2
2	a	2
2	b	2

TO DO: Draw the DiGraph corresponding to this transition table

Computable Functions and Computing Machines: **Finite Automata**

Examples of finite automaton



TO DO: Draw the transition table corresponding to this digraph

Computable Functions and Computing Machines: **Finite Automata**

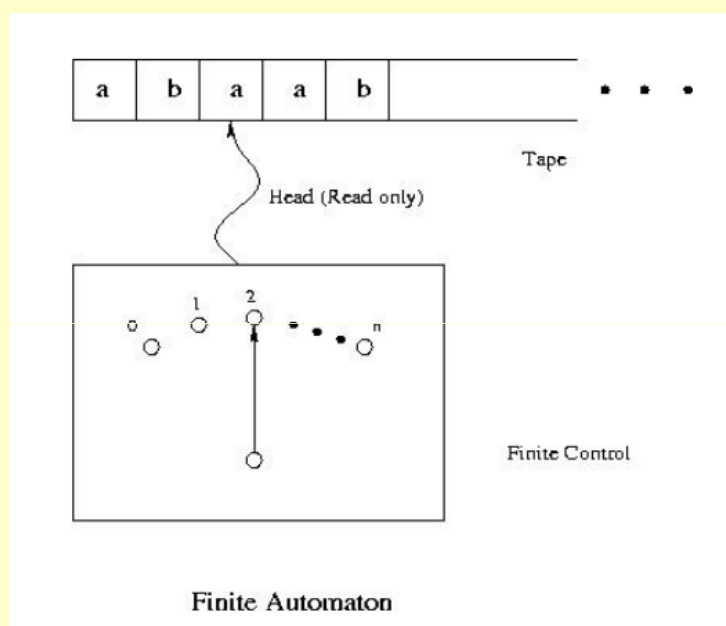
A finite automaton as a machine

A finite automaton can also be thought of as the device consisting of a tape and a control circuit which satisfy the following conditions:

- The tape has the left end and extends to the right without an end.
- The tape is divide into squares in each of which a symbol can be written prior to the start of the operation of the automaton.
- The tape has a read only head.
- The head is always at the leftmost square at the beginning of the operation.
- The head moves to the right one square every time it reads a symbol. It never moves to the left. When it sees no symbol, it stops and the automaton terminates its operation.
- There is a finite control which determines the state of the automaton and also controls the movement of the head.

Computable Functions and Computing Machines: **Finite Automata**

A finite automaton as a machine



Computable Functions and Computing Machines: **Finite Automata**

The Chomsky hierarchy: *linking machines to languages*

Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine.

Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.)

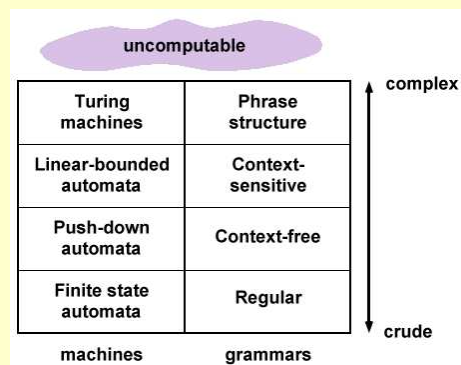
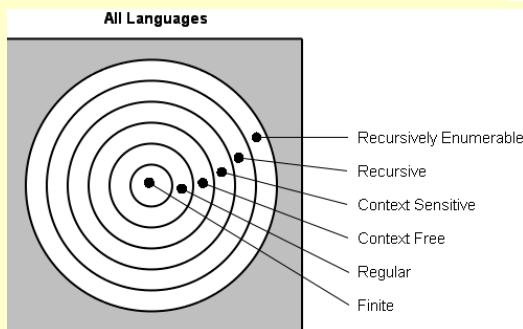
Type-2 grammars (context-free grammars) generate the context-free languages. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context-free languages are the theoretical basis for the syntax of most programming languages.

Type-3 grammars (regular grammars) generate the regular languages. These languages are exactly all languages that can be decided by a finite state automaton. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

Computable Functions and Computing Machines: **Finite Automata**

The differences arise out of the different types of production rules that are allowed:

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$



Computable Functions and Computing Machines: Finite Automata

Finite Automata and Regular Languages - some definitions

First, an **alphabet** is a finite set of symbols. For example $\{0, 1\}$ is an alphabet with two symbols, $\{a, b\}$ is another alphabet with two symbols and English alphabet is also an alphabet. A **string** (also called a word) is a finite sequence of symbols of an alphabet. b, a and $aabab$ are examples of string over alphabet $\{a, b\}$ and $0, 10$ and 001 are examples of string over alphabet $\{0, 1\}$. A **language** is a set of strings over an alphabet. Thus $\{a, ab, baa\}$ is a language (over alphabet $\{a, b\}$) and $\{0, 111\}$ is a language (over alphabet $\{0, 1\}$). The number of symbols in a string is called the **length of the string**. For a string w its length is represented by $|w|$. It can be [defined more formally by recursive definition](#). The **empty string** (also called null string) is the string with length 0. That is, it has no symbols. The empty string is denoted by Λ (capital lambda). Thus $|\Lambda| = 0$.

Let u and v be strings. Then uv denotes the string obtained by **concatenating** u with v , that is, uv is the string obtained by appending the sequence of symbols of v to that of u . For example if $u = aab$ and $v = bbab$, then $uv = aabbbab$. Note that $vu = bbabaab \neq uv$. We are going to use first few symbols of English alphabet such as a and b to denote symbols of an alphabet and those toward the end such as u and v for strings.

A string x is called a **substring** of another string y if there are strings u and v such that $y = uxv$. Note that u and v may be an empty string. So a string is a substring of itself. A string x is a **prefix** of another string y if there is a string v such that $y = xv$. v is called a **suffix** of y .

Computable Functions and Computing Machines: Finite Automata

Finite Automata and Regular Languages - some definitions

Since languages are sets, all the set operations can be applied to languages. Thus the union, intersection and difference of two languages over an alphabet Σ are languages over Σ . The complement of a language L over an alphabet Σ is $\Sigma^* - L$ and it is also a language.

Another operation on languages is concatenation. Let L_1 and L_2 be languages. Then the **concatenation** of L_1 with L_2 is denoted as L_1L_2 and it is defined as $L_1L_2 = \{ uv \mid u \in L_1 \text{ and } v \in L_2 \}$. That is L_1L_2 is the set of strings obtained by concatenating strings of L_1 with those of L_2 . For example $\{ab, b\} \{aaa, abb, aaba\} = \{abaaa, ababb, abaaba, baaa, babb, baaba\}$.

Computable Functions and Computing Machines: **Finite Automata**

Finite Automata and Regular Languages - some definitions

Powers : For a symbol $a \in \Sigma$ and a natural number k , a^k represents the concatenation of k a 's. For a string $u \in \Sigma^*$ and a natural number k , u^k denotes the concatenation of k u 's. Similarly for a language L , L^k means the concatenation of k L 's. Hence L^k is the set of strings that can be obtained by concatenating k strings of L . These powers can be formally defined recursively. For example L^k can be defined recursively as follows.

Recursive definition of L^k :

Basis Clause: $L^0 = \{ \Lambda \}$

Inductive Clause: $L^{(k+1)} = L^k L$.

Since L^k is defined for natural numbers k , the extremal clause is not necessary.

Computable Functions and Computing Machines: **Finite Automata**

Finite Automata and Regular Languages - some definitions

Recursive definition of L^* :

Basis Clause: $\Lambda \in L^*$

Inductive Clause: For any $x \in L^*$ and any $w \in L$, $xw \in L^*$.

Extremal Clause: Nothing is in L^* unless it is obtained from the above two clauses.

L^* is the set of strings obtained by **concatenating zero or more** strings of L as we are going to see in Theorem 1. This $*$ is called **Kleene star**.

For example if $L = \{ aba, bb \}$, then $L^* = \{ \Lambda, aba, bb, ababb, abaaba, bbbb, bbaba, \dots \}$

The $*$ in Σ^* is also the same Kleene star defined above.

Finite Automata and Regular Languages - some definitions

Recursive definition of L^+ :

Basis Clause: $L \subseteq L^+$

Inductive Clause: For any $x \in L^+$ and any $w \in L$, $xw \in L^+$.

Extremal Clause: Nothing is in L^+ unless it is obtained from the above two clauses.

Thus L^+ is the set of strings obtained by **concatenating one or more** strings of L .

For example if $L = \{ aba, bb \}$, then $L^+ = \{ aba, bb, ababb, abaaba, bbbb, bbaba, \dots \}$

Let us also define $\bigcup_{i=0}^{\infty} L^i$ (i.e. $L^0 \cup L \cup L^2 \dots$) as $\bigcup_{i=0}^{\infty} L^i = \{ x \mid x \in L^k \text{ for some natural number } k \}$.

Finite Automata and Regular Languages

The set of regular languages over an alphabet Σ is defined recursively as below. Any language belonging to this set is a **regular language** over Σ .

Definition of Set of Regular Languages :

Basis Clause: \emptyset , $\{\Lambda\}$ and $\{a\}$ for any symbol $a \in \Sigma$ are regular languages.

Inductive Clause: If L_r and L_s are regular languages, then $L_r \cup L_s$, $L_r L_s$ and L_r^* are regular languages.

Extremal Clause: Nothing is a regular language unless it is obtained from the above two clauses.

For example, let $\Sigma = \{a, b\}$. Then since $\{a\}$ and $\{b\}$ are regular languages, $\{a, b\}$ ($= \{a\} \cup \{b\}$) and $\{ab\}$ ($= \{a\}\{b\}$) are regular languages. Also since $\{a\}$ is regular, $\{a\}^*$ is a regular language which is the set of strings consisting of a's such as Λ , a , aa , aaa , $aaaa$ etc. Note also that Σ^* , which is the set of strings consisting of a's and b's, is a regular language because $\{a, b\}$ is regular.

Computable Functions and Computing Machines: **Finite Automata**

Finite Automata, Regular Languages and Regular Expressions

We use the following operations to construct **regular expressions**:

Boolean or - A vertical bar (or +) separates alternatives. For example, gray|grey can match "gray" or "grey".

Grouping - Parentheses are used to define the scope and precedence of the operators. For example, gray|grey and gr(a|e)y are equivalent patterns which both describe the set of "gray" and "grey".

Quantification - A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark ?, the asterisk * (derived from the Kleene star), and the plus sign +.

?The question mark indicates there is *zero or one* of the preceding element. For example, colour?r matches both "color" and "colour".

*The asterisk indicates there are *zero or more* of the preceding element. For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on.

+The plus sign indicates that there is *one or more* of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".

Computable Functions and Computing Machines: **Finite Automata**

Finite Automata, Regular Languages and Regular Expressions

Kleene's Theorem

This important result says that regular expressions have the same expressive power as finite automata: we can translate any r.e. into a finite automaton that accepts its language, and we can take a finite automaton and convert it into an equivalent r.e. (Note: some languages are more conveniently expressed as FAs or r.e.'s, but we always can translate.)

Kleene's Theorem. *Let L be a language over an alphabet A . Then L is regular if and only if it is the language accepted by some finite automaton with alphabet A .*

Finite Automata, Regular Languages and Regular Expressions

QUESTIONS: TO DO

1 Find the shortest string that is not in the language represented by the regular expression $a^*(ab)^*b^*$.

2 For the two regular expressions given below:

- (a) find a string corresponding to r_2 but not to r_1 and
- (b) find a string corresponding to both r_1 and r_2 .

$$r_1 = a^* + b^* \quad r_2 = ab^* + ba^* + b^*a + (a^*b)^*$$

3 Let r_1 and r_2 be arbitrary regular expressions over some alphabet. Find a simple (the shortest and with the smallest nesting of $*$ and $+$) regular expression which is equal to each of the following regular expressions

- (a) $(r_1 + r_2 + r_1r_2 + r_2r_1)^*$
- (b) $(r_1(r_1 + r_2)^*)^+$

Finite Automata, Regular Languages and Regular Expressions

QUESTION:

Can we build a regular expression to check that a string is a palindrome?

ANSWER: NO ... can you think about why this is the case?

Nondeterministic Finite Automata

NFAs are quite similar to DFAs. The only difference is in the transition function.

NFAs do not necessarily go to a unique next state. An NFA may not go to any state from the current state on reading an input symbol or it may select one of several states nondeterministically (e.g. by throwing a die) as its next state.

Nondeterministic Finite Automata

Let Q be a finite set and let Σ be a finite set of symbols. Also let δ be a function from $Q \times \Sigma$ to 2^Q , let q_0 be a state in Q and let A be a subset of Q . We call the elements of Q a **state**, δ the **transition function**, q_0 the **initial state** and A the set of **accepting states**.

Then a **nondeterministic finite automaton** is a 5-tuple $\langle Q, \Sigma, q_0, \delta, A \rangle$

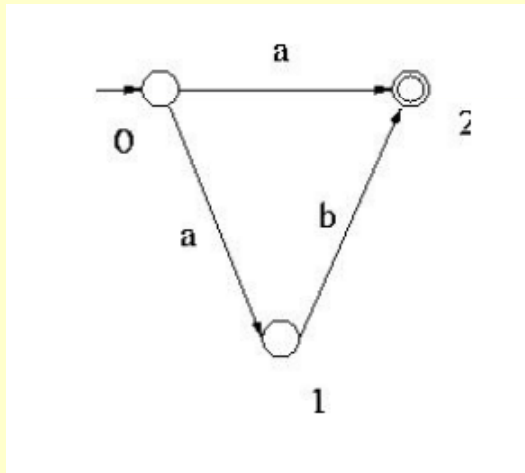
Notes on the definition

1. As in the case of DFA the set Q in the above definition is simply a set with a finite number of elements. Its elements can be interpreted as a state that the system (automaton) is in.
2. The transition function is also called a **next state function**. Unlike DFAs an NFA moves into one of the states given by $\delta(q, a)$ if it receives the input symbol a while in state q . Which one of the states in $\delta(q, a)$ to select is determined nondeterministically.
3. Note that δ is a function. Thus for **each state** q of Q and for **each symbol** a of Σ $\delta(q, a)$ must be specified. But it can be the empty set, in which case the NFA aborts its operation.
4. As in the case of DFA the accepting states are used to distinguish sequences of inputs given to the finite automaton. If the finite automaton is in an accepting state when the input ends i.e. ceases to come, the sequence of input symbols given to the finite automaton is "accepted". Otherwise it is not accepted.
5. Note that any DFA is also a NFA.

Computable Functions and Computing Machines: **Finite Automata**

Nondeterministic Finite Automata: Example

$Q = \{ 0, 1, 2 \}$, $\Sigma = \{ a, b \}$, $A = \{ 2 \}$, the initial state is 0



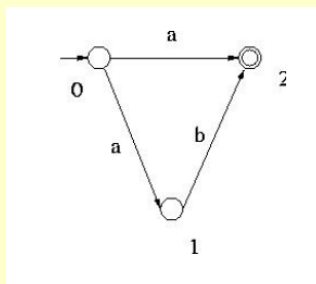
State (q)	Input (a)	Next State ($\delta (q, a)$)
0	a	{ 1 , 2 }
0	b	\emptyset
1	a	\emptyset
1	b	{ 2 }
2	a	\emptyset
2	b	\emptyset

Computable Functions and Computing Machines: **Finite Automata**

Nondeterministic Finite Automata: Theorem

For each NFA M , there is an *equivalent* deterministic FSA M' such that $L(M) = L(M')$.

NOTE: L is the language accepted by the machine/automata



QUESTION: Can you convert this NFA into an *equivalent* DFA?

Computable Functions and Computing Machines: **Finite Automata**

Nondeterministic Finite Automata: Theorem

NOT EVERY LANGUAGE IS REGULAR

Since we now have several tools for generating regular languages, it is tempting to think that all easily-described languages are regular. This is not true. Here is a property of all regular languages. If you can prove that some language you are interested in does NOT have this property, then it is not regular.

THEOREM (PUMPING LEMMA): Suppose $L \subseteq \Sigma^*$ is a regular language. Then there is some $n \in \mathbb{N}$ (n depends on L) such that if $|x| \geq n$ and $x \in L$, then there are strings $u, v, w \in \Sigma^*$ such that $x = uvw$, $v \neq \epsilon$, $|uv| \leq n$, and $uv^k w \in L$ for all $k \in \mathbb{N}$.

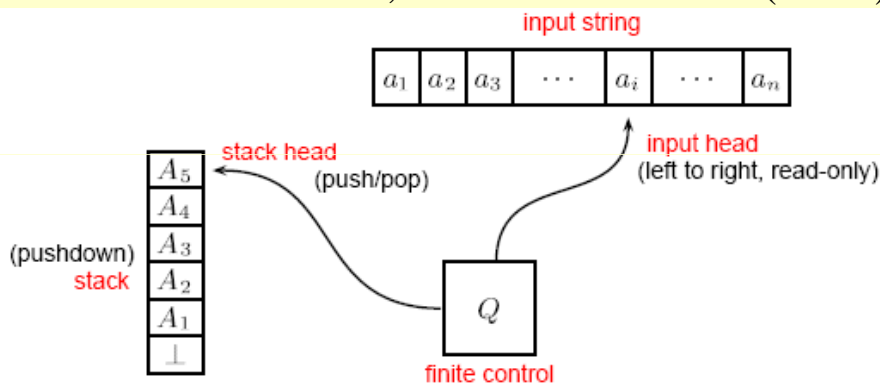
We have seen this (intuitively) with palindromes, but the pumping lemma allows us to prove it formally

NOTE: The language of palindromes is not **regular** (type 3) but it is **context-free** (type 2).

Computable Functions and Computing Machines: **Finite Automata**

Nondeterministic PushDown Automata

A (non-deterministic) pushdown automaton is like an NFA, except it has a *stack* (pushdown store) for recording a potentially unbounded amount of information, in a last-in-first-out (LIFO) fashion.



Theorem. A language is **context-free** (Chomsky type 2) iff some NPDA accepts it.

The workings of an NPDA

In each step, the NPDA pops the top symbol off the stack; based on (1) this symbol, (2) the input symbol currently reading, and (3) its current state, it can:

1. push a sequence of symbols (possibly) onto the stack
2. move its read head one cell to the right, and
3. enter a new state

according to the transition rule of the machine.

We allow ϵ -transition: an NPDA can pop and push without reading the next input symbol or moving its read head.

Note: an NPDA can only access the top of stack symbol in each step.

Example: primes

An example of a language that is not context-free is:

$$L = \{ a^p : p \text{ is a prime number} \}.$$

The language can easily be shown to be neither regular nor context free by applying the respective pumping lemmas for each of the language classes to L.

A **linear bounded automaton** is a restricted form of a nondeterministic **Turing machine**.

Linear bounded automata are acceptors for the class of **context-sensitive** languages (type 1 in Chomsky hierarchy).

$$L = \{ a^p : p \text{ is a prime number} \}.$$

L can be shown to be a **context-sensitive** language by constructing a linear bounded automaton which accepts L.

We will see what this looks like after we consider **Turing machines**

Turing Machines: consider a typical/standard definition

A Turing machine is a kind of *state machine*. At any time the machine is in any one of a finite number of states. Instructions for a Turing machine consist in specified conditions under which the machine will transition between one state and another.

A Turing machine has an infinite one-dimensional *tape* divided into cells. Traditionally we think of the tape as being horizontal with the cells arranged in a left-right orientation. The tape has one end, at the left say, and stretches infinitely far to the right. Each cell is able to contain one symbol, either '0' or '1'.

The machine has a *read-write head*, which at any time scanning a single cell on the tape. This read-write head can move left and right along the tape to scan successive cells.

The action of a Turing machine is determined completely by (1) the current state of the machine (2) the symbol in the cell currently being scanned by the head and (3) a table of transition rules, which serve as the “program” for the machine.

Turing Machines: consider a typical/standard definition

Each transition rule is a 4-tuple:

$\langle State_0, Symbol, State_{next}, Action \rangle$

which can be read as saying "if the machine is in state $State_0$ and the current cell contains $Symbol$ then move into state $State_{next}$ taking $Action$ ". The actions available to a Turing machine are either to write a symbol on the tape in the current cell (which we will denote with the symbol in question), or to move the head one cell to the left or right, which we will denote by the symbols « and » respectively.

If the machine reaches a situation in which there is not exactly one transition rule specified, i.e., none or more than one, then the machine halts.

Turing Machines: consider a typical/standard definition

In modern terms:

- the tape serves as the memory of the machine
- the read-write head is the memory bus through which data is accessed (and updated) by the machine.

Note:

- the machine's tape is infinite in length, corresponding to an assumption that the memory of the machine is infinite.
- that a function will be Turing-computable if there exists a set of instructions that will result in the machine computing the function regardless of the amount of time it takes. One can think of this as assuming the availability of infinite time to complete the computation.

If a function is not Turing-computable it is because Turing machines lack the computational machinery to carry it out, not because of a lack of spatio-temporal resources.

Turing Machines: consider a typical/standard definition

Describing Turing Machines

Every Turing machine has the same machinery.

What makes one Turing machine perform one task and another a different task is the table of transition rules that make up the machine's program, and a specified *initial state* for the machine.

We will assume throughout that a machine starts in the lowest numbered of its states.

We can describe a Turing machine, therefore, by specifying only the 4-tuples that make up its program.

Note: there are « equivalent » alternative definitions, eg using 5-tuples where the machine can write on the tape and move the head in a single step

Turing Machines: consider a typical/standard definition

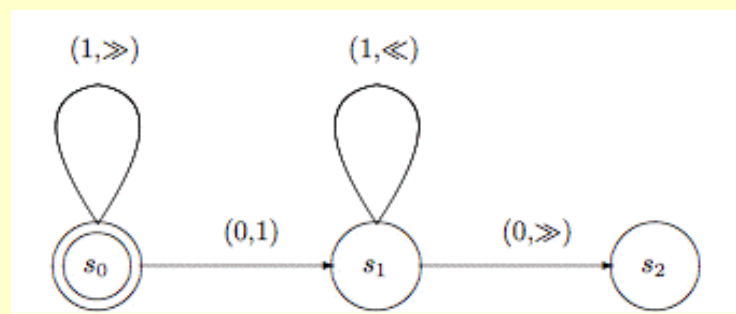
Example

$\langle s_0, 1, s_0, \gg \rangle$

$\langle s_0, 0, s_1, 1 \rangle$

$\langle s_1, 1, s_1, \ll \rangle$

$\langle s_1, 0, s_2, \gg \rangle$



In order to speak about a Turing machine that does *something useful*, we will have to provide an interpretation of the symbols recorded on the tape. For example, if we want to design a machine which will perform some mathematical function, addition say, then we will need to describe how to interpret the ones and zeros appearing on the tape as numbers.

Turing Machines: another typical/standard definition

Example: A Turing machine for checking if input string is a palindrome

```
- Palindrome Detector for strings of 0's and 1's
- A is Start State; Z is Accept State
(A, 1, , B, R) - detect 1 - store as state B
(A, 0, , C, R) - detect 0 - store as state C
(A, , , Z, R) - empty string - done! (even number)
(B, 0, 0, B, R) - go right
(B, 1, 1, B, R)
(B, , , D, L) - end of string detected - go back
(C, 0, 0, C, R) - go right
(C, 1, 1, C, R)
(C, , , E, L) - end of string detected - go back
(D, 1, , F, L) - found 1, cancel it, & go back
(D, , , Z, L) - or found blank - done! (odd number)
(E, 0, , F, L) - found 0, cancel it & go back
(E, , , Z, L) - or found blank - done! (odd number)
(F, 0, 0, F, L) - go back
(F, 1, 1, F, L)
(F, , , A, R) - beginning of string detected
```

TO DO:
check you
understand
how this
works

Turing Machines: programming problem

TO DO: design a TM to calculate the remainder function %

$X \% Y = Z \iff \exists N: \text{NAT}. X = Y * N + Z \ \&\& \ Z < Y,$
where X and Y are positive integers and Z is a non-negative integer

Eg $18 \% 5 = 3$ since $18 = 5 * 3 + 3 \ \&\& \ 3 < 5$
 $20 \% 4 = 0$ since $20 = 5 * 4 + 0$

Your input string will be written in unary notation with the tape head pointing at the 1st character in the input string:

$11111111 \% 111$ X represents the function call $8 \% 3$

The result (in this case 11) should be output to the immediate right of the tape head: $11X$

Computable Functions and Computing Machines: finite automata

A linear bounded automaton: a restriction on TMs

An **LBA** differs from a Turing machine in that while the tape is initially considered to have unbounded length, only a finite contiguous portion of the tape, whose length is a linear function of the length of the initial input, can be accessed by the read/write head.

This limitation makes an LBA a more accurate model of computers that actually exist than a Turing machine in some respects.

A linear bounded automaton recognizes context-sensitive languages

A Turing machine recognizes all formal languages (unrestricted grammars) ... these are also known as **recursively enumerable** languages.

Recursively enumerable sets are also known as semi-recursive/partially decidable sets because it is always possible to confirm in finite time – using a Turing Machine - that a given element is a member of the set, but not necessarily that it isn't.

Computable Functions and Computing Machines: finite automata

Recursively enumerable vs recursive

A recursively enumerable language is a formal language for which there exists a Turing machine (or other computable function) that will halt and accept when presented with any string in the language as input but may either halt and reject or loop forever when presented with a string not in the language.

Contrast this to **recursive languages**, which require that the Turing machine halts in all cases.

Recursive languages are between type 0 and type 1 in the original Chomsky hierarchy

Computable Functions and Computing Machines: finite automata

Recursively enumerable vs recursive

Given the closure properties for these two classes of languages, we can infer that for any language L exactly one of the following three possibilities apply:

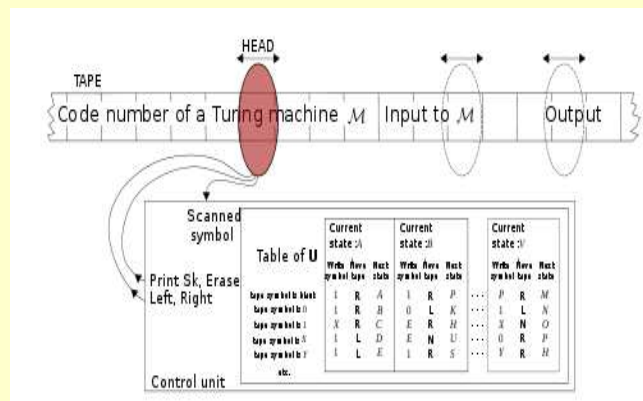
1. both L and \bar{L} are recursive,
2. neither L nor \bar{L} is recursively enumerable, or
3. one of L and \bar{L} is recursively enumerable but not recursive, while the other is not recursively enumerable.

	\bar{L} recursive	\bar{L} recursively enumerable (but not recursive)	\bar{L} not recursively enumerable
L recursive	✓	X	X
L recursively enumerable (but not recursive)	X	X	✓
L not recursively enumerable	X	✓	✓

Computable Functions and Computing Machines: finite automata

Universal Turing Machine

Every Turing machine computes a certain fixed **partial computable** function from the input strings over its alphabet. In that sense it behaves like a computer with a fixed program. However, we can encode the action table of any Turing machine in a string. Thus we can construct a Turing machine that expects on its tape a string describing an action table followed by a string describing the input tape, and computes the tape that the encoded Turing machine would have computed. Turing described such a construction in complete detail in his 1936 paper



ASIDE: There are some *very small* universal Turing Machines c.f. Woods and Neary ☺

QUESTION: can you guess how small (number of states and transitions)

Computable Functions and Computing Machines: finite automata

Universal Machines

In order to show that any other machine is universal (ie it is equivalent in power to a Universal Turing Machine, we usually follow a proof by construction/proof by simulation:

To show an **A** (some class of machines) is as powerful as a **B** (some class of machines) we need to show that for any **B**, there is some equivalent **A**.

Proof-by-construction:

Given any $b \in \mathbf{B}$, construct an $a \in \mathbf{A}$ that recognizes the same language as b .

Proof-by-simulation:

Show that there is some **A** that can simulate any **B**.

Note: the existence of a Universal machine in **B** guarantees the existence of a universal machine in **A**. Also, we can show that a machine is no more powerful than a Turing machine by showing that a TM can simulate its behaviour.

Computable Functions and Computing Machines: finite automata

Universal Machines: Register Machines

It has been proven that Register Machines are universal – recognise recursively enumerable languages. These machines are interesting because they correspond to a very low-level *programming language*. There are many different sub-classes - from primitive to computer-like, eg:

- *Counter machine -- the most primitive and reduced model. Lacks indirect addressing.*
- *Pointer machine -- a blend of counter machine and RAM models.*
- *Random access machine (RAM) -- a counter machine with indirect addressing and, usually, an augmented instruction set.*
- *Random access stored program machine model (RASP) -- a RAM with instructions in its registers analogous to the Universal Turing machine; thus it is an example of the von Neumann architecture. But unlike a computer the model is idealized with effectively-infinite registers. The instruction set is much reduced in the number of instructions.*

The most primitive machine models are useful if one wishes to show (through simulation) that another machine is universal. The most powerful are useful in showing that something (set/function) is semi-computable/decidable

Universal Machines: A Simple Register Machine

A simple computational model introduced by Shepherdson and Sturgis (1963) is the *register machine model*. There is an infinite number of registers indexed by the integers. Each register stores an integer. We write $c_t(n)$ for the **contents** of register $n \in \mathbb{Z}$ at time $t \in \mathbb{N}$. We omit t when convenient and unambiguous. The following are instructions:

- ZERO n : This is the assignment $c(n) \leftarrow 0$.
- SUCC n : This is the assignment $c(n) \leftarrow c(n) + 1$. We also allow ADD n, m or SUB n, m as instructions: this means $c(n) \leftarrow c(n) \pm c(m)$.
- SET n, m : This is just $c(n) \leftarrow c(m)$.
- JUMP n, L : If $c(n) = 0$ then go to instruction L ; else proceed normally to the next instruction. L is a positive integer.
- ACCEPT: The computation halts and accepts. If an output number is desired, this can be $c(0)$. The computation may also come to a halt because there is no next instruction – this is regarded as rejection.
- CHOOSE L, L' : this allows the choice mode of computation.

A **register machine** is just a finite sequence of instructions. The instructions are numbered consecutively, starting from 1 (this is a convention for the labels in JUMP instructions). This model is obviously geared towards computation over natural numbers. We may introduce conventions about input and output, etc. Note that a register machine can only refer to a finite number of registers – and yet its computation need not be bounded because the registers can store arbitrarily large numbers.

NOTE:
You don't need to know how this works – you just need to see the simplicity of the definition.

Universal Machines: Register Machine Extensions

§2. The RAM model

An extension of the register machines is to allow indirect addressing: we can use the contents of a register to name another register that will be used as the operand of our instructions. That is, instead of the argument n in the above instructions, we can also have an argument of the form “@ n ”. This means that we are using the register $c(n)$ instead of the register n . The resulting model is called the **random-access memory model** (RAM model). This can be viewed as an abstract simple assembly language.

§3. Vector Machines

The RAM model here may be called **integer RAM model**. An alternative is to work with bit vectors. The motivation for vector machines is to introduce parallelism into the register model. This was first discussed by Pratt, Stockmeyer and Rabin (1974).

Universal Machines: lambda-calculus

The Lambda Calculus was developed by Alonzo Church in the 1930s and published in 1941 as 'The Calculi Of Lambda Conversion'. It became important, along with Turing machines, in the development of computation theory, and is the theoretical basis of all functional programming languages, such as Lisp, Haskell and ML.

A function in lambda calculus is written in the form $\lambda x.E$, where x is the function's parameter and E is a lambda expression constituting the function body. A lambda expression is either a variable (like the x in the above expression), a function in the form above, or an application $E_1 E_2$.

In the expression $\lambda x.E$, any occurrence of x in E is 'bound', while any other variable is 'free' (unless bound by another lambda expression, like the y in $\lambda x.\lambda y.xy$). A 'pure' lambda expression has no free variables.

Universal Machines: lambda-calculus

Three things can be done with lambda expressions:

α conversion

Alpha conversion renames a bound variable — $\lambda x.x$ can be alpha converted to $\lambda y.y$.

β reduction

Beta reduction allows applications to be reduced — $(\lambda x.E_1)E_2$ can be beta reduced to E_1 with all occurrences of x replaced with E_2 . If there are name clashes (for example in $(\lambda x.\lambda y.xy)y$), alpha conversion may be required first.

η conversion

Eta conversion allows us to say that f and $\lambda x.fx$ are equivalent.

The calculus can be called the *smallest universal programming language*. The calculus consists of a single transformation rule (variable substitution) and a single function denition scheme.

Universal Machines: lambda-calculus

The lambda calculus can be used to implement:

- Natural numbers and arithmetic
- Boolean logic
- Conditional execution
- Recursion

The lambda calculus is equivalent to a Turing machine

The halting problem

Given a description of a program, decide whether the program finishes running or will run forever.

This is equivalent to the problem of deciding, given a program and an input, whether the program will eventually halt when run with that input, or will run forever.

Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

The halting problem is undecidable over Turing machines.

The halting problem is famous because it was one of the first problems proven algorithmically undecidable. (First for the lambda calculus and then for Turing machines)

The halting problem: sketch of proof

The proof shows there is no total computable function that decides whether an arbitrary program i halts on arbitrary input x ; that is, the following function h is not computable:

$$h(i, x) = \begin{cases} 1 & \text{if program } i \text{ halts on input } x, \\ 0 & \text{otherwise.} \end{cases}$$

Here *program* i refers to the i th program in an enumeration of all the programs of a fixed Turing-complete model of computation.

The proof proceeds by directly establishing that every total computable function with two arguments differs from the required function h .

To this end, given any total computable binary function f , the following partial function g is also computable by some program e :

$$g(i) = \begin{cases} 0 & \text{if } f(i, i) = 0, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The halting problem: sketch of proof

The following **pseudocode** illustrates a straightforward way to compute g :

```
procedure compute_g(i):  
  if f(i,i) == 0 then  
    return 0  
  else  
    loop forever
```

Because g is partial computable, there must be a program e that computes g , by the assumption that the model of computation is Turing-complete. This program is one of all the programs on which the halting function h is defined. The next step of the proof shows that $h(e,e)$ will not have the same value as $f(e,e)$. It follows from the definition of g that exactly one of the following two cases must hold:

- $g(e) = f(e,e) = 0$. In this case $h(e,e) = 1$, because program e halts on input e .
- $g(e)$ is undefined and $f(e,e) \neq 0$. In this case $h(e,e) = 0$, because program e does not halt on input e .

In either case, f cannot be the same function as h . Because f was an arbitrary total computable function with two arguments, all such functions must differ from h . This proof is typically referred to as a diagonalization proof.

Computable Functions and Computing Machines: finite automata

The halting problem: solvable on finite machines (in theory)

The halting problem is, *in theory* if not in practice, decidable for linear bounded automata (**LBAs**), or deterministic machines with finite memory.

A machine with finite memory has a finite number of states, and thus any deterministic program on it must eventually either halt or repeat a previous state:

"...any finite-state machine, if left completely to itself, will fall eventually into a perfectly periodic repetitive pattern. The duration of this repeating pattern cannot exceed the number of internal states of the machine..."(Minsky 1967, p. 24)

Minsky warns us, however, that machines such as computers with e.g. a million small parts, each with two states, will have on the order of $2^{1,000,000}$ possible states!!!!