Turing Machine and Persistence in Computer Science

Merlin Forbes

Lina Mcdougal

First Edition, 2012

ISBN 978-81-323-0995-6

Published by:
Academic Studio
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1 - Introduction to Turing Machine

Chapter 2 - Universal Turing Machine

Chapter 3 - Turing Machine Equivalents and Turing Machine Examples

Chapter 4 - Post–Turing Machine and Langton's Ant

Chapter 5 - Non–Deterministic Turing Machine and Turmite

Chapter 6 - System Image and System Prevalence

Chapter 7 - Database Management System

Chapter 8 - Serialization

Chapter 9 - Carbonado (Java) and Persistent Data Structure

Chapter 10 - Java Persistence API and Snapshot (Computer Storage)

Chapter 11 - QuickDB ORM

Chapter 12 - XML

Chapter 13 - Base64

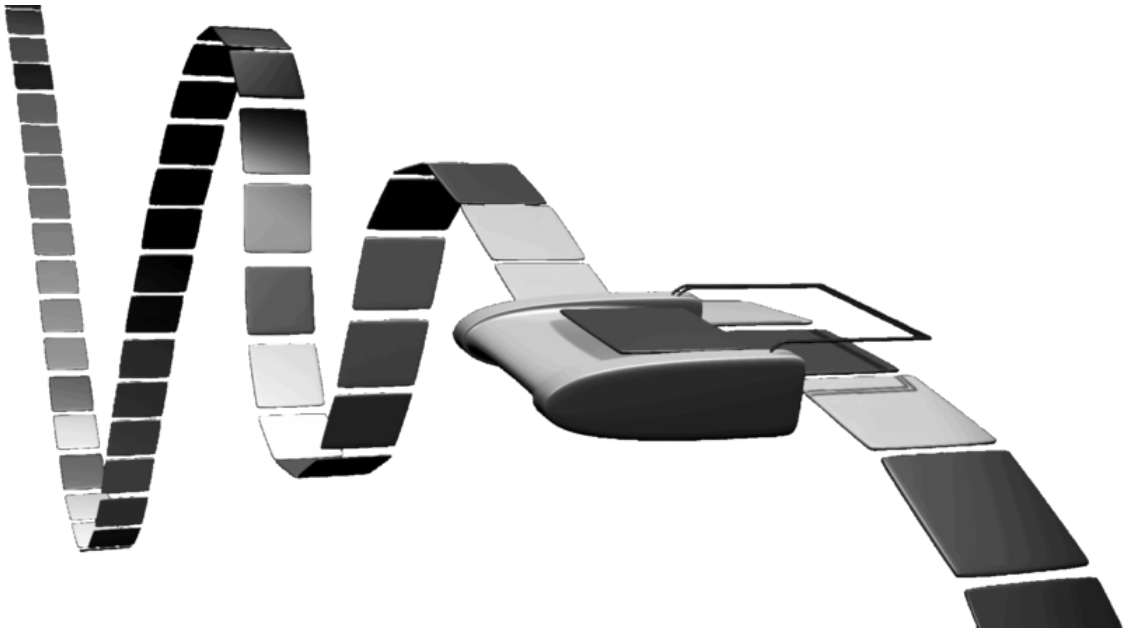
Chapter 14 - Comma–Separated Values

Chapter 15 - JSON

Chapter 16 - JsonML

Chapter 1

Introduction to Turing Machine



An artistic representation of a Turing machine (Rules table not represented)

A **Turing machine** is a theoretical device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.

The "Turing" machine was described by Alan Turing in 1937, who called it an "*a*(utomatic)-machine". Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine. They help computer scientists understand the limits of mechanical computation.

Turing gave a succinct definition of the experiment in his 1948 essay, "Intelligent Machinery". Referring to his 1936 publication, Turing wrote that the Turing machine, here called a Logical Computing Machine, consisted of:

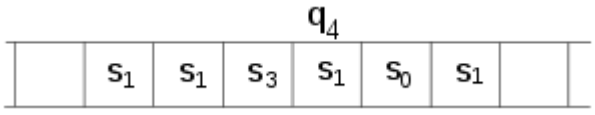
...an infinite memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings. (Turing 1948, p. 61)

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, whose work on lambda calculus intertwined with Turing's in a formal theory of computation known as the Church–Turing thesis. The thesis states that Turing machines indeed capture the informal notion of effective method in logic and mathematics, and provide a precise definition of an algorithm or 'mechanical procedure'.

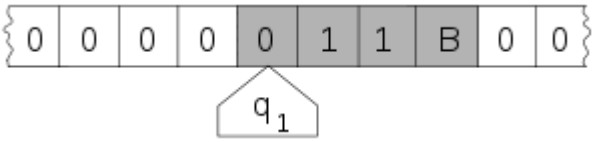
Studying their abstract properties yields many insights into computer science and complexity theory.

Informal description

The Turing machine mathematically models a machine that mechanically operates on a tape on which symbols are written which it can read and write one at a time using a tape head. Operation is fully determined by a finite set of elementary instructions such as "in state 42, if the symbol seen is 0, write a 1; if the symbol seen is 1, shift to the right, and change into state 17; in state 17, if the symbol seen is 0, write a 1 and change to state 6;" etc. In the original article ("On computable numbers, with an application to the Entscheidungsproblem"), Turing imagines not a mechanism, but a person whom he calls the "computer", who executes these deterministic mechanical rules slavishly (or as Turing puts it, "in a desultory manner").



The head is always over a particular square of the tape; only a finite stretch of squares is given. The instruction to be performed (q_4) is shown over the scanned square. (Drawing after Kleene (1952) p.375.)



Here, the internal state (q_1) is shown inside the head, and the illustration describes the tape as being infinite and pre-filled with "0", the symbol serving as blank. The system's full state (its *configuration*) consists of the internal state, the contents of the shaded squares including the blank scanned by the head ("11B"), and the position of the head. (Drawing after Minsky (1967) p. 121).

More precisely, a Turing machine consists of:

1. **TAPE** which is divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special *blank* symbol (here written as 'B') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written to before are assumed to be filled with the blank symbol. In some models the tape has a left end marked with a special symbol; the tape extends or is indefinitely extensible to the right.
2. A **HEAD** that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time. In some models the head moves and the tape is stationary.
3. A finite **TABLE** ("action table", or *transition function*) of instructions (usually quintuples [5-tuples] : $q_i a_j \rightarrow q_{i1} a_{j1} d_k$, but sometimes 4-tuples) that, given the *state*(q_i) the machine is currently in *and* the *symbol*(a_j) it is reading on the tape (symbol currently under HEAD) tells the machine to do the following in sequence (for the 5-tuple models):
    - Either erase or write a symbol (instead of a_j written a_{j1}), *and then*
    - Move the head (which is described by d_k and can have values: 'L' for one step left *or* 'R' for one step right *or* 'N' for staying in the same place), *and then*
    - Assume the same or a *new state* as prescribed (go to state q_{i1}).

    In the 4-tuple models, erase or write a symbol (a_{j1}) and move the head left or right (d_k) are specified as separate instructions. Specifically, the TABLE tells the machine to (ia) erase or write a symbol *or* (ib) move the head left or right, *and then* (ii) assume the same or a new state as prescribed, but not both actions (ia) and (ib) in the same instruction. In some models, if there is no entry in the table for the current combination of symbol and state then the machine will halt; other models require all entries to be filled.

4. A **STATE REGISTER** that stores the state of the Turing table, one of finitely many. There is one special *start state* with which the state register is initialized. These states, writes Turing, replace the "state of mind" a person performing computations would ordinarily be in.

Note that every part of the machine—its state and symbol-collections—and its actions—printing, erasing and tape motion—is *finite*, *discrete* and *distinguishable*; it is the potentially unlimited amount of tape that gives it an unbounded amount of storage space.

Formal definition

Hopcroft and Ullman (1979, p. 148) formally define a (one-tape) Turing machine as a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where

- Q is a finite, non-empty set of *states*
- Γ is a finite, non-empty set of the *tape alphabet/symbols*
- $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation)
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *final* or *accepting states*.
- $\delta : Q \setminus F \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a partial function called the *transition function*, where L is left shift, R is right shift. (A relatively uncommon variant allows "no shift", say N, as a third element of the latter set.)

Anything that operates according to these specifications is a Turing machine.

The 7-tuple for the 3-state busy beaver looks like this :

Q = { **A**, **B**, **C**, **HALT** }
Γ = { 0, 1 }
b = 0 = "blank"
Σ = { 1 }
δ = see state-table below
q_0 = **A** = initial state
F = the one element set of final states {**HALT**}

Initially all tape cells are marked with 0.

State table for 3 state, 2 symbol busy beaver

Tape symbol	**Current state A**			**Current state B**			**Current state C**		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	1	R	**B**	1	L	**A**	1	L	**B**
1	1	L	**C**	1	R	**B**	1	R	**HALT**

Additional details required to visualize or implement Turing machines

In the words of van Emde Boas (1990), p. 6: "The set-theoretical object [his formal seven-tuple description similar to the above] provides only partial information on how the machine will behave and what its computations will look like."

For instance,

- There will need to be some decision on what the symbols actually look like, and a failproof way of reading and writing symbols indefinitely.
- The shift left and shift right operations may shift the tape head across the tape, but when actually building a Turing machine it is more practical to make the tape slide back and forth under the head instead.
- The tape can be finite, and automatically extended with blanks as needed (which is closest to the mathematical definition), but it is more common to think of it as stretching infinitely at both ends and being pre-filled with blanks except on the explicitly given finite fragment the tape head is on. (This is, of course, not implementable in practice.) The tape *cannot* be fixed in length, since that would not correspond to the given definition and would seriously limit the range of computations the machine can perform to those of a linear bounded automaton.

Alternative definitions

Definitions in literature sometimes differ slightly, to make arguments or proofs easier or clearer, but this is always done in such a way that the resulting machine has the same computational power. For example, changing the set $\{L,R\}$ to $\{L,R,N\}$, where N ("None" or "No-operation") would allow the machine to stay on the same tape cell instead of moving left or right, does not increase the machine's computational power.

The most common convention represents each "Turing instruction" in a "Turing table" by one of nine 5-tuples, per the convention of Turing/Davis (Turing (1936) in *Undecidable*, p. 126-127 and Davis (2000) p. 152):

> (definition 1): $\mathbf{(q_i, S_j, S_k/E/N, L/R/N, q_m)}$
> **(** current state $\mathbf{q_i}$ **,** symbol scanned $\mathbf{S_j}$ **,** print symbol $\mathbf{S_k}$/erase **E**/none **N ,** move_tape_one_square left **L**/right **R**/none **N ,** new state $\mathbf{q_m}$ **)**

Other authors (Minsky (1967) p. 119, Hopcroft and Ullman (1979) p. 158, Stone (1972) p. 9) adopt a different convention, with new state $\mathbf{q_m}$ listed immediately after the scanned symbol S_j:

> (definition 2): $\mathbf{(q_i, S_j, q_m, S_k/E/N, L/R/N)}$
> **(** current state $\mathbf{q_i}$ **,** symbol scanned $\mathbf{S_j}$ **,** new state $\mathbf{q_m}$ **,** print symbol $\mathbf{S_k}$/erase **E**/none **N ,** move_tape_one_square left **L**/right **R**/none **N)**

Example: state table for the 3-state 2-symbol busy beaver reduced to 5-tuples

Current state	Scanned symbol	Print symbol	Move tape	Final (i.e. next) state	5-tuples
A	0	1	R	**B**	(**A**, 0, 1, R, **B**)
A	1	1	L	**C**	(**A**, 1, 1, L, **C**)
B	0	1	L	**A**	(**B**, 0, 1, L, **A**)
B	1	1	R	**B**	(**B**, 1, 1, R, **B**)
C	0	1	L	**B**	(**C**, 0, 1, L, **B**)
C	1	1	N	**H**	(**C**, 1, 1, N, **H**)

In the following table, Turing's original model allowed only the first three lines that he called N1, N2, N3 (cf Turing in *Undecidable*, p. 126). He allowed for erasure of the "scanned square" by naming a 0th symbol S_0 = "erase" or "blank", etc. However, he did not allow for non-printing, so every instruction-line includes "print symbol S_k" or "erase" (cf footnote 12 in Post (1947), *Undecidable* p. 300). The abbreviations are Turing's (*Undecidable* p. 119). Subsequent to Turing's original paper in 1936–1937, machine-models have allowed all nine possible types of five-tuples:

	Current m-configuration (Turing state)	Tape symbol	Print-operation	Tape-motion	Final m-configuration (Turing state)	5-tuple	5-tuple comments	4-tuple
N1	q_i	S_j	Print(S_k)	Left L	q_m	(q_i, S_j, S_k, L, q_m)	"blank" = S_0, 1=S_1, etc.	
N2	q_i	S_j	Print(S_k)	Right R	q_m	(q_i, S_j, S_k, R, q_m)	"blank" = S_0, 1=S_1, etc.	
N3	q_i	S_j	Print(S_k)	None N	q_m	(q_i, S_j, S_k, N, q_m)	"blank" = S_0, 1=S_1, etc.	(q_i, S_j, S_k, q_m)
4	q_i	S_j	None N	Left L	q_m	(q_i, S_j, N, L, q_m)		(q_i, S_j, L, q_m)
5	q_i	S_j	None N	Right R	q_m	(q_i, S_j,		(q_i, S_j,

						N, R, q_m)		R, q_m)
6	q_i	S_j	None N	None N	q_m	(q_i, S_j, N, N, q_m)	Direct "jump"	(q_i, S_j, N, q_m)
7	q_i	S_j	Erase	Left L	q_m	(q_i, S_j, E, L, q_m)		
8	q_i	S_j	Erase	Right R	q_m	(q_i, S_j, E, R, q_m)		
9	q_i	S_j	Erase	None N	q_m	(q_i, S_j, E, N, q_m)		(q_i, S_j, E, q_m)

Any Turing table (list of instructions) can be constructed from the above nine 5-tuples. For technical reasons, the three non-printing or "N" instructions (4, 5, 6) can usually be dispensed with.

Less frequently the use of 4-tuples are encountered: these represent a further atomization of the Turing instructions (cf Post (1947), Boolos & Jeffrey (1974, 1999), Davis-Sigal-Weyuker (1994)).

The "state"

The word "state" used in context of Turing machines can be a source of confusion, as it can mean two things. Most commentators after Turing have used "state" to mean the name/designator of the current instruction to be performed—i.e. the contents of the state register. But Turing (1936) made a strong distinction between a record of what he called the machine's "m-configuration", (its internal state) and the machine's (or person's) "state of progress" through the computation - the current state of the total system. What Turing called "the state formula" includes both the current instruction and *all* the symbols on the tape:

Thus the state of progress of the computation at any stage is completely determined by the note of instructions and the symbols on the tape. That is, the **state of the system** may be described by a single expression (sequence of symbols) consisting of the symbols on the tape followed by Δ (which we suppose not to appear elsewhere) and then by the note of instructions. This expression is called the 'state formula'.
—*Undecidable*, p.139–140, emphasis added

Earlier in his paper Turing carried this even further: he gives an example where he places a symbol of the current "m-configuration"—the instruction's label—beneath the scanned square, together with all the symbols on the tape (*Undecidable*, p. 121); this he calls "the

complete configuration" (*Undecidable*, p. 118). To print the "complete configuration" on one line he places the state-label/m-configuration to the *left* of the scanned symbol.

A variant of this is seen in Kleene (1952) where Kleene shows how to write the Gödel number of a machine's "situation": he places the "m-configuration" symbol q_4 over the scanned square in roughly the center of the 6 non-blank squares on the tape and puts it to the *right* of the scanned square. But Kleene refers to "q_4" itself as "the machine state" (Kleene, p. 374-375). Hopcroft and Ullman call this composite the "instantaneous description" and follow the Turing convention of putting the "current state" (instruction-label, m-configuration) to the *left* of the scanned symbol (p. 149).

Example: total state of 3-state 2-symbol busy beaver after 3 "moves" (taken from example "run" in the figure below):

1**A**1

This means: after three moves the tape has ... 000110000 ... on it, the head is scanning the right-most 1, and the state is **A**. Blanks (in this case represented by "0"s) can be part of the total state as shown here: **B**01 ; the tape has a single 1 on it, but the head is scanning the 0 ("blank") to its left and the state is **B**.

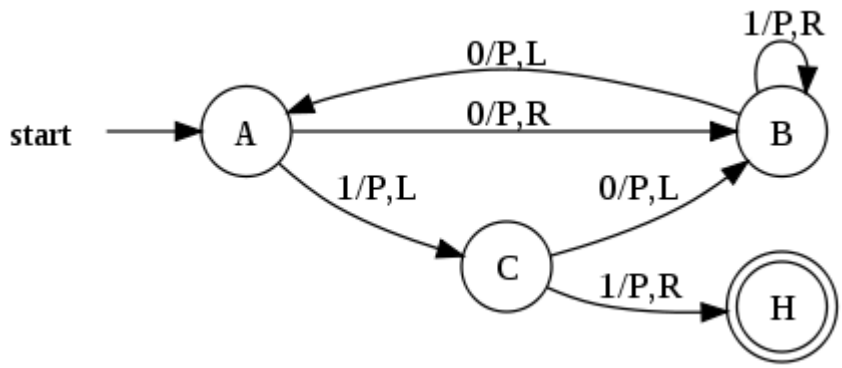
"State" in the context of Turing machines should be clarified as to which is being described: (*i*) the current instruction, or (*ii*) the list of symbols on the tape together with the current instruction, or (*iii*) the list of symbols on the tape together with the current instruction placed to the left of the scanned symbol or to the right of the scanned symbol.

Turing's biographer Andrew Hodges (1983: 107) has noted and discussed this confusion.

Turing machine "state" diagrams

The table for the 3-state busy beaver ("P" = print/write a "1")

Tape symbol	**Current state A**			**Current state B**			**Current state C**		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	P	R	**B**	P	L	**A**	P	L	**B**
1	P	L	**C**	P	R	**B**	P	R	**HALT**

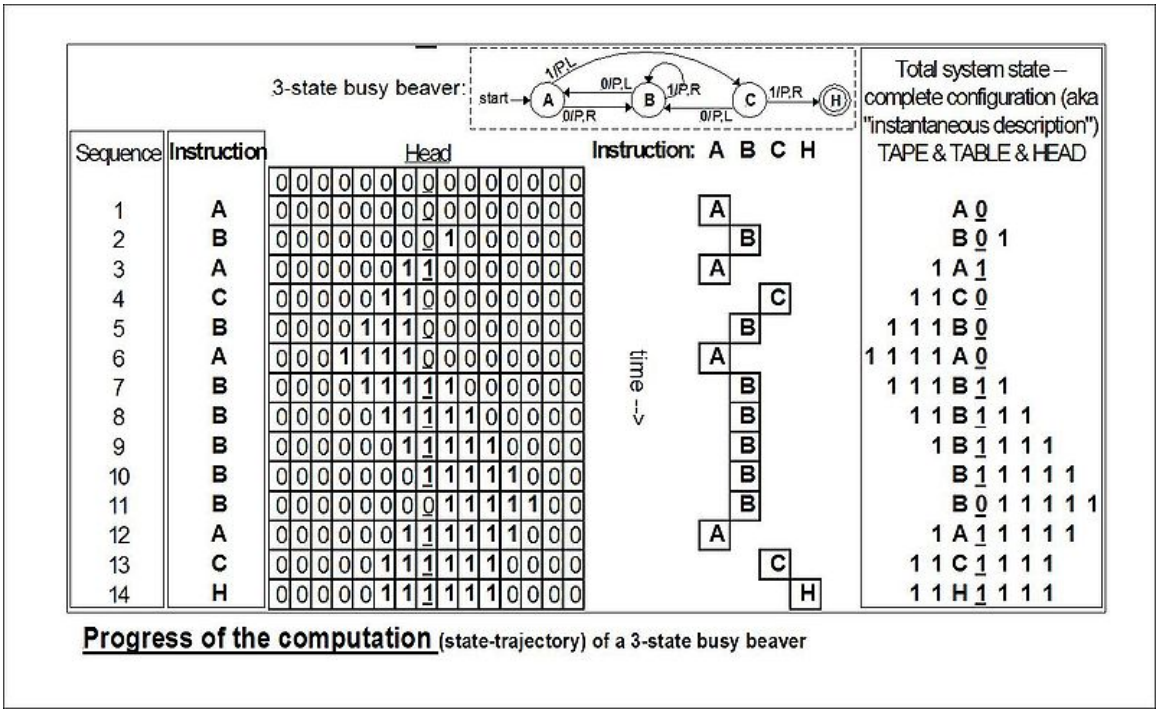


The "3-state busy beaver" Turing Machine in a finite state representation. Each circle represents a "state" of the TABLE—an "m-configuration" or "instruction". "Direction" of a state *transition* is shown by an arrow. The label (e.g.. **0/P,R**) near the outgoing state (at the "tail" of the arrow) specifies the scanned symbol that causes a particular transition (e.g. **0**) followed by a slash /, followed by the subsequent "behaviors" of the machine, e.g. "**P P**rint" then move tape "**R R**ight". No general accepted format exists. The convention shown is after McClusky (1965), Booth (1965), Hill and Peterson (1974).

To the right: the above TABLE as expressed as a "state transition" diagram.

Usually large TABLES are better left as tables (Booth, p. 74). They are more readily simulated by computer in tabular form (Booth, p. 74). However, certain concepts—e.g. machines with "reset" states and machines with repeating patterns (cf Hill and Peterson p. 244ff)—can be more readily seen when viewed as a drawing.

Whether a drawing represents an improvement on its TABLE must be decided by the reader for the particular context.



Progress of the computation (state-trajectory) of a 3-state busy beaver

The evolution of the busy-beaver's computation starts at the top and proceeds to the bottom.

The reader should again be cautioned that such diagrams represent a snapshot of their TABLE frozen in time, *not* the course ("trajectory") of a computation *through* time and/or space. While every time the busy beaver machine "runs" it will always follow the same state-trajectory, this is not true for the "copy" machine that can be provided with variable input "parameters".

The diagram "Progress of the computation" shows the 3-state busy beaver's "state" (instruction) progress through its computation from start to finish. On the far right is the Turing "complete configuration" (Kleene "situation", Hopcroft-Ullman "instantaneous description") at each step. If the machine were to be stopped and cleared to blank both the "state register" and entire tape, these "configurations" could be used to rekindle a computation anywhere in its progress (cf Turing (1936) *Undecidable* pp. 139–140).

Models equivalent to the Turing machine model

Many machines that might be thought to have more computational capability than a simple universal Turing machine can be shown to have no more power (Hopcroft and Ullman p. 159, cf Minsky (1967)). They might compute faster, perhaps, or use less memory, or their instruction set might be smaller, but they cannot compute more powerfully (i.e. more mathematical functions). (Recall that the Church–Turing thesis *hypothesizes* this to be true for any kind of machine: that anything that can be "computed" can be computed by some Turing machine.)

A Turing machine is equivalent to a pushdown automaton that has been made more flexible and concise by relaxing the last-in-first-out requirement of its stack. (Interestingly, this seemingly minor relaxation enables the Turing machine to perform such a wide variety of computations that it can serve as a clearer model for the computational capabilities of all modern computer software.)

At the other extreme, some very simple models turn out to be Turing-equivalent, i.e. to have the same computational power as the Turing machine model.

Common equivalent models are the multi-tape Turing machine, multi-track Turing machine, machines with input and output, and the *non-deterministic* Turing machine (NDTM) as opposed to the *deterministic* Turing machine (DTM) for which the action table has at most one entry for each combination of symbol and state.

Read-only, right-moving Turing Machines are equivalent to NDFA's (as well as DFA's by conversion using the NDFA to DFA conversion algorithm).

For practical and didactical intentions the equivalent register machine can be used as a usual assembly programming language.

Choice c-machines, Oracle o-machines

Early in his paper (1936) Turing makes a distinction between an "automatic machine"—its "motion ... completely determined by the configuration" and a "choice machine":

...whose motion is only partially determined by the configuration ... When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems.
—*Undecidable*, p. 118

Turing (1936) does not elaborate further except in a footnote in which he describes how to use an a-machine to "find all the provable formulae of the [Hilbert] calculus" rather than use a choice machine. He "suppose[s] that the choices are always between two possibilities 0 and 1. Each proof will then be determined by a sequence of choices i_1, i_2, ..., i_n ($i_1 = 0$ or 1, $i_2 = 0$ or 1, ..., $i_n = 0$ or 1), and hence the number $2^n + i_1 2^{n-1} + i_2 2^{n-2} + ... +i_n$ completely determines the proof. The automatic machine carries out successively proof 1, proof 2, proof 3, ..." (Footnote ‡, *Undecidable*, p. 138)

This is indeed the technique by which a deterministic (i.e. a-) Turing machine can be used to mimic the action of a nondeterministic Turing machine; Turing solved the matter in a footnote and appears to dismiss it from further consideration.

An oracle machine or o-machine is a Turing a-machine that pauses its computation at state "**o**" while, to complete its calculation, it "awaits the decision" of "the oracle"—an

unspecified entity "apart from saying that it cannot be a machine" (Turing (1939), Undecidable p. 166–168). The concept is now actively used by mathematicians.

Comparison with real machines

It is often said that Turing machines, unlike simpler automata, are as powerful as real machines, and are able to execute any operation that a real program can. What is missed in this statement is that, because a real machine can only be in finitely many *configurations*, in fact this "real machine" is nothing but a linear bounded automaton. On the other hand, Turing machines are equivalent to machines that have an unlimited amount of storage space for their computations. In fact, Turing machines are not intended to model computers, but rather they are intended to model computation itself; historically, computers, which compute only on their (fixed) internal storage, were developed only later.

There are a number of ways to explain why Turing machines are useful models of real computers:

1. Anything a real computer can compute, a Turing machine can also compute. For example: "A Turing machine can simulate any type of subroutine found in programming languages, including recursive procedures and any of the known parameter-passing mechanisms" (Hopcroft and Ullman p. 157). A large enough FSA can also model any real computer, disregarding IO. Thus, a statement about the limitations of Turing machines will also apply to real computers.
2. The difference lies only with the ability of a Turing machine to manipulate an unbounded amount of data. However, given a finite amount of time, a Turing machine (like a real machine) can only manipulate a finite amount of data.
3. Like a Turing machine, a real machine can have its storage space enlarged as needed, by acquiring more disks or other storage media. If the supply of these runs short, the Turing machine may become less useful as a model. But the fact is that neither Turing machines nor real machines need astronomical amounts of storage space in order to perform useful computation. The processing time required is usually much more of a problem.
4. Descriptions of real machine programs using simpler abstract models are often much more complex than descriptions using Turing machines. For example, a Turing machine describing an algorithm may have a few hundred states, while the equivalent deterministic finite automaton on a given real machine has quadrillions. This makes the DFA representation infeasible to analyze.
5. Turing machines describe algorithms independent of how much memory they use. There is a limit to the memory possessed by any current machine, but this limit can rise arbitrarily in time. Turing machines allow us to make statements about algorithms which will (theoretically) hold forever, regardless of advances in *conventional* computing machine architecture.
6. Turing machines simplify the statement of algorithms. Algorithms running on Turing-equivalent abstract machines are usually more general than their counterparts running on real machines, because they have arbitrary-precision data

types available and never have to deal with unexpected conditions (including, but not limited to, running out of memory).

One way in which Turing machines are a poor model for programs is that many real programs, such as operating systems and word processors, are written to receive unbounded input over time, and therefore do not halt. Turing machines do not model such ongoing computation well (but can still model portions of it, such as individual procedures).

Limitations of Turing machines

Computational Complexity Theory

A limitation of Turing Machines is that they do not model the strengths of a particular arrangement well. For instance, modern stored-program computers are actually instances of a more specific form of abstract machine known as the random access stored program machine or RASP machine model. Like the Universal Turing machine the RASP stores its "program" in "memory" external to its finite-state machine's "instructions". Unlike the Universal Turing Machine, the RASP has an infinite number of distinguishable, numbered but unbounded "registers"—memory "cells" that can contain any integer (cf. Elgot and Robinson (1964), Hartmanis (1971), and in particular Cook-Rechow (1973); references at random access machine). The RASP's finite-state machine is equipped with the capability for indirect addressing (e.g. the contents of one register can be used as an address to specify another register); thus the RASP's "program" can address any register in the register-sequence. The upshot of this distinction is that there are computational optimizations that can be performed based on the memory indices, which are not possible in a general Turing Machine; thus when Turing Machines are used as the basis for bounding running times, a 'false lower bound' can be proven on certain algorithms' running times (due to the false simplifying assumption of a Turing Machine). An example of this is binary search, an algorithm that can be shown to perform more quickly when using the RASP model of computation rather than the Turing machine model.

Concurrency

Another limitation of Turing machines is that they do not model concurrency well. For example, there is a bound on the size of integer that can be computed by an always-halting nondeterministic Turing Machine starting on a blank tape. By contrast, there are always-halting concurrent systems with no inputs that can compute an integer of unbounded size. (A process can be created with local storage that is initialized with a count of 0 that concurrently sends itself both a stop and a go message. When it receives a go message, it increments its count by 1 and sends itself a go message. When it receives a stop message, it stops with an unbounded number in its local storage.)

History

They were described in 1936 by Alan Turing.

Historical background: computational machinery

Robin Gandy (1919–1995)—a student of Alan Turing (1912–1954) and his life-long friend—traces the lineage of the notion of "calculating machine" back to Babbage (circa 1834) and actually proposes "Babbage's Thesis":

That the whole of development and operations of analysis are now capable of being executed by machinery.
—(italics in Babbage as cited by Gandy, p. 54)

Gandy's analysis of Babbage's Analytical Engine describes the following five operations (cf p. 52–53):

1. The arithmetic functions +, −, × where − indicates "proper" subtraction $x - y = 0$ if $y \geq x$
2. Any sequence of operations is an operation
3. Iteration of an operation (repeating n times an operation P)
4. Conditional iteration (repeating n times an operation P conditional on the "success" of test T)
5. Conditional transfer (i.e. conditional "goto")

Gandy states that "the functions which can be calculated by (1), (2), and (4) are precisely those which are Turing computable." (p. 53). He cites other proposals for "universal calculating machines" included those of Percy Ludgate (1909), Leonardo Torres y Quevedo (1914), M. d'Ocagne (1922), Louis Couffignal (1933), Vannevar Bush (1936), Howard Aiken (1937). However:

... the emphasis is on programming a fixed iterable sequence of arithmetical operations. The fundamental importance of conditional iteration and conditional transfer for a general theory of calculating machines is not recognized ...
—Gandy p. 55

The Entscheidungsproblem (the "decision problem"): Hilbert's tenth question of 1900

With regards to Hilbert's problems posed by the famous mathematician David Hilbert in 1900, an aspect of problem #10 had been floating about for almost 30 years before it was framed precisely. Hilbert's original expression for #10 is as follows:

10. Determination of the solvability of a Diophantine equation. Given a Diophantine equation with any number of unknown quantities and with rational integral coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

The Entscheidungsproblem [decision problem for first-order logic] is solved when we know a procedure that allows for any given logical expression to decide by finitely many

operations its validity or satisfiability ... The Entscheidungsproblem must be considered the main problem of mathematical logic.

—quoted, with this translation and the original German, in Dershowitz and Gurevich, 2008

By 1922, this notion of "Entscheidungsproblem" had developed a bit, and H. Behmann stated that

... most general form of the Entscheidungsproblem [is] as follows:

> A quite definite generally applicable prescription is required which will allow one to decide in a finite number of steps the truth or falsity of a given purely logical assertion ...

—Gandy p. 57, quoting Behmann

Behmann remarks that ... the general problem is equivalent to the problem of deciding which mathematical propositions are true.

—*ibid.*

If one were able to solve the Entscheidungsproblem then one would have a "procedure for solving many (or even all) mathematical problems".

—*ibid.*, p. 92

By the 1928 international congress of mathematicians Hilbert "made his questions quite precise. First, was mathematics *complete* ... Second, was mathematics *consistent* ... And thirdly, was mathematics *decidable*?" (Hodges p. 91, Hawking p. 1121). The first two questions were answered in 1930 by Kurt Gödel at the very same meeting where Hilbert delivered his retirement speech (much to the chagrin of Hilbert); the third—the Entscheidungsproblem—had to wait until the mid-1930s.

The problem was that an answer first required a precise definition of "*definite general applicable prescription*", which Alonzo Church would come to call "effective calculability", and in 1928 no such definition existed. But over the next 6–7 years Emil Post developed his definition of a worker moving from room to room writing and erasing marks per a list of instructions (Post 1936), as did Princeton professor Church and his two students Stephen Kleene and J. B. Rosser by use of Church's lambda-calculus and Gödel's recursion theory (1934). Church's paper (published 15 April 1936) showed that the Entscheidungsproblem was indeed "undecidable" and beat Turing to the punch by almost a year (Turing's paper submitted 28 May 1936, published January 1937). In the meantime, Emil Post submitted a brief paper in the fall of 1936, so Turing at least had priority over Post. While Church refereed Turing's paper, Turing had time to study Church's paper and add an Appendix where he sketched a proof that Church's lambda-calculus and his machines would compute the same functions.

But what Church had done was something rather different, and in a certain sense weaker. ... the Turing construction was more direct, and provided an argument from first principles, closing the gap in Church's demonstration.

—Hodges p. 112

And Post had only proposed a definition of calculability and criticized Church's "definition", but had proved nothing.

Alan Turing's a- (automatic-)machine

In the spring of 1935 Turing as a young Master's student at King's College Cambridge, UK, took on the challenge; he had been stimulated by the lectures of the logician M. H. A. Newman "and learned from them of Gödel's work and the Entscheidungsproblem ... Newman used the word 'mechanical' ... In his obituary of Turing 1955 Newman writes:

To the question 'what is a "mechanical" process?' Turing returned the characteristic answer 'Something that can be done by a machine' and he embarked on the highly congenial task of analysing the general notion of a computing machine.
—Gandy, p. 74

Gandy states that:

I suppose, but do not know, that Turing, right from the start of his work, had as his goal a proof of the undecidability of the Entscheidungsproblem. He told me that the 'main idea' of the paper came to him when he was lying in Grantchester meadows in the summer of 1935. The 'main idea' might have either been his analysis of computation or his realization that there was a universal machine, and so a diagonal argument to prove unsolvability.
—*ibid.*, p. 76

While Gandy believed that Newman's statement above is "misleading", this opinion is not shared by all. Turing had a life-long interest in machines: "Alan had dreamt of inventing typewriters as a boy; [his mother] Mrs. Turing had a typewriter; and he could well have begun by asking himself what was meant by calling a typewriter 'mechanical'" (Hodges p. 96). While at Princeton pursuing his PhD, Turing built a Boolean-logic multiplier (see below). His PhD thesis, titled "Systems of Logic Based on Ordinals", contains the following definition of "a computable function":

It was stated above that 'a function is effectively calculable if its values can be found by some purely mechanical process'. We may take this statement literally, understanding by a purely mechanical process one which could be carried out by a machine. It is possible to give a mathematical description, in a certain normal form, of the structures of these machines. The development of these ideas leads to the author's definition of a computable function, and to an identification of computability with effective calculability. It is not difficult, though somewhat laborious, to prove that these three definitions [the 3rd is the λ-calculus] are equivalent.
—Turing (1939) in *The Undecidable*, p. 160

When Turing returned to the UK he ultimately became jointly responsible for breaking the German secret codes created by encryption machines called "The Enigma"; he also became involved in the design of the ACE (Automatic Computing Engine), "[Turing's]

ACE proposal was effectively self-contained, and its roots lay not in the EDVAC [the USA's initiative], but in his own universal machine" (Hodges p. 318). Arguments still continue concerning the origin and nature of what has been named by Kleene (1952) Turing's Thesis. But what Turing *did prove* with his computational-machine model appears in his paper *On Computable Numbers, With an Application to the Entscheidungsproblem* (1937):

[that] the Hilbert Entscheidungsproblem can have no solution ... I propose, therefore to show that there can be no general process for determining whether a given formula U of the functional calculus K is provable, i.e. that there can be no machine which, supplied with any one U of these formulae, will eventually say whether U is provable.
—from Turing's paper as reprinted in *The Undecidable*, p. 145

Turing's example (his second proof): If one is to ask for a general procedure to tell us: "Does this machine ever print 0", the question is "undecidable".

1937–1970: The "digital computer", the birth of "computer science"

In 1937, while at Princeton working on his PhD thesis, Turing built a digital (Boolean-logic) multiplier from scratch, making his own electromechanical relays (Hodges p. 138). "Alan's task was to embody the logical design of a Turing machine in a network of relay-operated switches ..." (Hodges p. 138). While Turing might have been just curious and experimenting, quite-earnest work in the same direction was going in Germany (Konrad Zuse (1938)), and in the United States (Howard Aiken) and George Stibitz (1937); the fruits of their labors were used by the Axis and Allied military in World War II (cf Hodges p. 298–299). In the early to mid-1950s Hao Wang and Marvin Minsky reduced the Turing machine to a simpler form (a precursor to the Post-Turing machine of Martin Davis); simultaneously European researchers were reducing the new-fangled electronic computer to a computer-like theoretical object equivalent to what was now being called a "Turing machine". In the late 1950s and early 1960s, the coincidentally-parallel developments of Melzak and Lambek (1961), Minsky (1961), and Shepherdson and Sturgis (1961) carried the European work further and reduced the Turing machine to a more friendly, computer-like abstract model called the counter machine; Elgot and Robinson (1964), Hartmanis (1971), Cook and Reckhow (1973) carried this work even further with the register machine and random access machine models—but basically all are just multi-tape Turing machines with an arithmetic-like instruction set.

1970–present: the Turing machine as a model of computation

Today the counter, register and random-access machines and their sire the Turing machine continue to be the models of choice for theorists investigating questions in the theory of computation. In particular, computational complexity theory makes use of the Turing machine:

Depending on the objects one likes to manipulate in the computations (numbers like nonnegative integers or alphanumeric strings), two models have obtained a dominant position in machine-based complexity theory:
the off-line multitape Turing machine..., which represents the standard model for string-oriented computation, and
the *random access machine (RAM)* as introduced by Cook and Reckhow ..., which models the idealized Von Neumann style computer.
—van Emde Boas 1990:4
Only in the related area of analysis of algorithms this role is taken over by the RAM model.
—van Emde Boas 1990:16

Kantorovitz (2005), was the first to show the most simple obvious representation of Turing Machines published academically which unifies Turing Machines with mathematical analysis and analog computers.

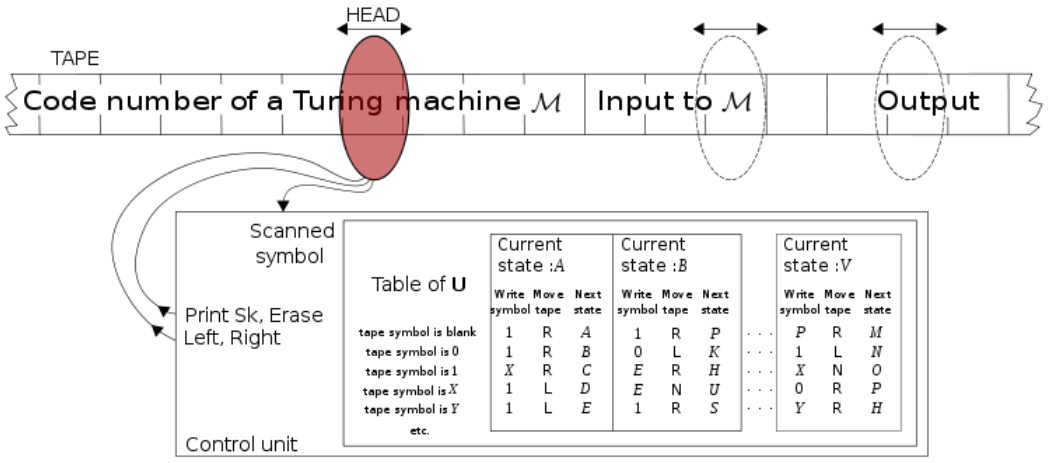
Chapter 2

Universal Turing Machine

In computer science, a **universal Turing machine** is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of machine to be simulated as well as the input thereof from its own tape. Alan Turing introduced this machine in 1936–1937. This model is considered by some (for example, Martin Davis (2000)) to be the origin of the stored program computer—used by John von Neumann (1946) for the "Electronic Computing Instrument" that now bears von Neumann's name: the von Neumann architecture. It is also known as **universal computing machine**, **universal machine**, **machine U**, **U**.

In terms of computational complexity, a multi-tape universal Turing machine need only be slower by logarithmic factor compared to the machines it simulates.

Introduction



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:

"It is possible to invent a single machine which can be used to compute any computable sequence. If this machine **U** is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine **M**, then **U** will compute the same sequence as **M**."

Stored-program computer

Davis makes a persuasive argument that Turing's conception of what is now known as "the stored-program computer", of placing the "action table" -- the instructions for the machine—in the same "memory" as the input data, strongly influenced John von Neumann's conception of the first discrete-symbol (as opposed to analog) computer—the EDVAC. Davis quotes *Time* magazine to this effect, that "everyone who taps at a keyboard... is working on an incarnation of a Turing machine," and that "John von Neumann [built] on the work of Alan Turing" (Davis 2000:193 quoting *Time* magazine of 29 March 1999).

Davis makes a case that Turing's Automatic Computing Engine (ACE) computer "anticipated" the notions of microprogramming (microcode) and RISC processors (Davis 2000:188). Knuth cites Turing's work on the ACE computer as designing "hardware to facilitate subroutine linkeage" (Knuth 1973:225); Davis also references this work as Turing's use of a hardware "stack" (Davis 2000:237 footnote 18).

As the Turing Machine was encouraging the construction of computers, the UTM was encouraging the development of the fledgling computer sciences. An early, if not the very first, assembler was proposed "by a young hot-shot programmer" for the EDVAC (Davis 2000:192). Von Neumann's "first serious program ... [was] to simply sort data efficiently" (Davis 2000:184). Knuth observes that the subroutine return embedded in the program itself rather than in special registers is attributable to von Neumann and Goldstine. Knuth furthermore states that

"The first interpretive routine may be said to be the "Universal Turing Machine" ... Interpretive routines in the conventional sense were mentioned by John Mauchly in his lectures at the Moore School in 1946 ... Turing took part in this development also; interpretive systems for the Pilot ACE computer were written under his direction" (Knuth 1973:226).

Davis briefly mentions operating systems and compilers as outcomes of the notion of program-as-data (Davis 2000:185).

Some, however, might raise issues with this assessment. At the time (mid-1940s to mid-1950s) a relatively small cadre of researchers were intimately involved with the

architecture of the new "digital computers". Hao Wang (1954), a young researcher at this time, made the following observation:

Turing's theory of computable functions antedated but has not much influenced the extensive actual construction of digital computers. These two aspects of theory and practice have been developed almost entirely independently of each other. The main reason is undoubtedly that logicians are interested in questions radically different from those with which the applied mathematicians and electrical engineers are primarily concerned. It cannot, however, fail to strike one as rather strange that often the same concepts are expressed by very different terms in the two developments." (Wang 1954, 1957:63)

Wang hoped that his paper would "connect the two approaches." Indeed, Minsky confirms this: "that the first formulation of Turing-machine theory in computer-like models appears in Wang (1957)" (Minsky 1967:200). Minsky goes on to demonstrate Turing equivalence of a counter machine.

With respect to the reduction of computers to simple Turing equivalent models (and vice versa), Minsky's designation of Wang as having made "the first formulation" is open to debate. While both Minsky's paper of 1961 and Wang's paper of 1957 are cited by Shepherdson and Sturgis (1963), they also cite and summarize in some detail the work of European mathematicians Kaphenst (1959), Ershov (1959), and Péter (1958). The names of mathematicians Hermes (1954, 1955, 1961) and Kaphenst (1959) appear in the bibliographies of both Sheperdson-Sturgis (1963) and Elgot-Robinson (1961). Two other names of importance are Canadian researchers Melzak (1961) and Lambek (1961).

Mathematical theory

With this encoding of action tables as strings it becomes possible in principle for Turing machines to answer questions about the behaviour of other Turing machines. Most of these questions, however, are undecidable, meaning that the function in question cannot be calculated mechanically. For instance, the problem of determining whether any particular Turing machine will halt on a particular input, or on all inputs, known as the Halting problem, was shown to be, in general, undecidable in Turing's original paper. Rice's theorem shows that any non-trivial question about the behaviour or output of a Turing machine is undecidable.

A universal Turing machine can calculate any recursive function, decide any recursive language, and accept any recursively enumerable language. According to the Church-Turing thesis, the problems solvable by a universal Turing machine are exactly those problems solvable by an *algorithm* or an *effective method of computation*, for any reasonable definition of those terms. For these reasons, a universal Turing machine serves as a standard against which to compare computational systems, and a system that can simulate a universal Turing machine is called Turing complete.

An abstract version of the universal Turing machine is the universal function, a computable function which can be used to calculate any other computable function. The utm theorem proves the existence of such a function.

When Alan Turing came up with the idea of a universal machine he had in mind the simplest computing model powerful enough to calculate all possible functions which can be calculated. Claude Shannon first explicitly posed the question of finding the smallest possible universal Turing machine when in 1956 he showed that two symbols were sufficient, so long as enough states were used. Shannon himself proved that it was always possible to exchange states by symbols.

Efficiency

Without loss of generality, the input of Turing machine can be assumed to be in the alphabet {0, 1}; any other finite alphabet can be encoded over {0, 1}. The behavior of a Turing machine M is determined by its transition function. This function can be easily encoded as a string over the alphabet {0, 1} as well. The size of the alphabet of M, the number of tapes it has, and the size of the state space can be deduced from the transition function's table. The distinguished states and symbols can be identified by their position, e.g. the first two states can by convention be the start and stop states. Consequently, every Turing machine can be encoded as a string over the alphabet {0, 1}. Additionally, we convene that every invalid encoding maps to a trivial Turing machine that immediately halts, and that every Turing machine can have an infinite number of encodings by padding the encoding with an arbitrary number of (say) 1's at the end, just like comments work in a programming language. It should be no surprise that we can achieve this encoding given the existence of a Gödel number and computational equivalence between Turing machines and μ-recursive functions. Similarly, our construction associates to every binary string α, a Turing machine M_α.

Starting from the above encoding, in 1966 F. C. Hennie and R. E. Stearns showed that given a Turing machine M_α that halts on input x within N steps, then there exists a multi-tape universal Turing machine that halts on inputs α, x (given on different tapes) in $CN \log N$, where C is a machine-specific constant that does not depend on the length of the input x, but does depend on M's alphabet size, number of tapes, and number of states. Effectively this is a $O(N \log N)$ simulation.

Smallest machines

After some time, the smallest known universal Turing machine was due to Marvin Minsky who in 1962 discovered a 7-state 4-symbol universal Turing machine using 2-tag systems. Applying Shannon's result to Minsky's UTM upon conversion to a 2-symbol machine Minsky machine would require 43 states.

Other smaller universal Turing machines have since been found. If we denote by (m,n) the class of UTMs with m states and n symbols the following tuples were found by Yurii Rogozhin in 1996: (24, 2), (10, 3), (7, 4), (5, 5), (4, 6), (3, 10), and (2, 18). In 1985,

Stephen Wolfram conjectured a 2-state 5-symbol universal Turing machine. This conjecture was proved by Matthew Cook working as a research assistant to Stephen Wolfram. The proof was based on emulating the Rule 110 Elementary Cellular Automaton. The model had, at the time, the smallest product (2,5)=10 of any known universal Turing machine. According to Wolfram other smaller UTMs should exist and he proposed a 2-state 3-symbol Turing Machine as a candidate. On 24 October 2007, Wolfram announced the Turing equivalence of the system had been proven by Alex Smith -- an undergraduate studying electronic and computer engineering at the University of Birmingham -- responding to a contest established by Wolfram. However, on 29 October 2007 Vaughan Pratt of Stanford University claimed that he discovered a flaw in the proof. Wolfram Research and Smith himself disputed Pratt's interpretation . Pratt's main point was that the same argument that would make Wolfram's 2,3 Turing machine universal would make a Linear Bounded Automaton (LBA) universal. Smith explained that the LBA would need to be restarted at running time to perform a computation, while the 2, 3 Turing machine restarts automatically, therefore the proof does not make an LBA universal as Pratt first thought. Other small (weak/semi-weak) universal Turing machines were found by Watanabe, Rogozhin, Margenstern and more recently Neary and Woods.

Example of universal-machine coding

The following example is taken from Turing (1936).

Turing used seven symbols { A, C, D, R, L, N, ; } to encode each 5-tuple; as described in Turing machine, his 5-tuples are only of types N1, N2, and N3. The number of each "m-configuration" (instruction, state) is represented by "D" followed by a unary string of A's, i.e. "q3" = DAAA. In a similar manner he encodes the symbols blank as "D", the symbol "0" is "DC", the symbol "1" as DCC, etc. The symbols "R", "L", and "N" remain as is.

After encoding each 5-tuple is then "assembled" into a string in order as shown in the following table:

Finally, the codes for all four 5-tuples are strung together into a code started by ";" and separated by ";" i.e.:

;DADDCRDAA**;**DAADDRDAAA**;**DAAADDCCRDAAAA**;**DAAAADDRDA

This code he placed on alternate squares—the "F-squares" -- leaving the "E-squares" (those liable to erasure) empty. The final assembly of the code on the tape for the U-machine consists of placing two special symbols ("e") one after the other, then the code separated out on alternate squares, and lastly the double-colon symbol "**::**" (blanks shown here with "." for clarity):

ee.**;**.D.A.D.D.C.R.D.A.A.**;**.D.A.A.D.D.R.D.A.A.A.**;**.D.A.A.A.D.D.C.C.R.D.A.A.A.A.**;**.D.A.A.A.A.D.D.R.D.A.**::**.....

The U-machine's action-table (state-transition table) is responsible for decoding the symbols. Turing's action table keeps track of its place with markers "u", "v", "x", "y", "z" by placing them in "E-squares" to the right of "the marked symbol" -- for example, to mark the current instruction **z** is placed to the right of ";" **x** is keeping the place with respect to the current "m-configuration" DAA. The U-machine's action table will shuttle these symbols around (erasing them and placing them in different locations) as the computation progresses:

> ee.**;** .D.A.D.D.C.R.D.A.A. **;**
> **z**D.A.A**x**D.D.R.D.A.A.A.**;**.D.A.A.A.D.D.C.C.R.D.A.A.A.A.**;**.D.A.A.A.A.D.D.R.D.A.**::**.....

Turing's action-table for his U-machine is very involved.

A number of other commentators (notably Penrose 1989) provide examples of ways to encode instructions for the Universal machine. As does Penrose, most commentators use only binary symbols i.e. only symbols { 0, 1 }, or { blank, mark | }. Penrose goes further and writes out his entire U-machine code (Penrose 1989:71–73). He asserts that it truly is a U-machine code, an enormous number that spans almost 2 full pages of 1's and 0's. For readers interested in simpler encodings for the Post-Turing machine the discussion of Davis in Steen (Steen 1980:251ff) may be useful.

Chapter 3

Turing Machine Equivalents and Turing Machine Examples

Turing machine equivalents

Machines equivalent to the Turing machine model

Turing equivalence:

Many machines that might be thought to have more computational capability than a simple universal Turing machine can be shown to have no more power (Hopcroft and Ullman p. 159, cf Minsky). They might compute faster, perhaps, or use less memory, or their instruction set might be smaller, but they cannot compute more powerfully (i.e. more mathematical functions). (Recall that the Church-Turing thesis *hypothesizes* this to be true: that anything that can be "computed" can be computed by some Turing machine.)

While none of the following models have been shown to have more power than the single-tape, one-way infinite, multi-symbol Turing-machine model, their authors defined and used them to investigate questions and solve problems more easily than they could have if they had stayed with Turing's *a*-machine model.

The sequential-machine models:

All of the following are called "sequential machine models" to distinguish them from "parallel machine models" (van Emde Boas (1990) p. 18).

Tape-based Turing machines

Turing's *a*-machine model: Turing's (1936) a-machine (his name) was left-ended, right-end-infinite. He provided symbols **əə** to mark the left end. Any of finite number of tape symbols were permitted. The instructions (if a universal machine), and the "input" and "out" were written on only on "F-squares", and markers were to appear on "E-squares". In essence he divided his machine into two tapes that always moved together. The

instructions appeared in a tabular form called "5-tuples" and were not executed sequentially.

Single-tape machines with restricted symbols and/or restricted instructions

The following models are single tape Turing machines but restricted with (i) restricted tape symbols { mark, blank }, and/or (ii) sequential, computer-like instructions, and/or (iii) machine-actions fully-atomized.

Post's "Formulation 1" model of computation

Emil Post (1936) in an independent description of a computational process, reduced the symbols allowed to the equivalent binary set of marks on the tape { "mark", "blank"=not_mark }. He changed the notion of "tape" from 1-way infinite to the right to an infinite set of rooms each with a sheet of paper in both directions. He atomized the Turing 5-tuples into 4-tuples—motion instructions separate from print/erase instructions. Although his (1936) model is ambiguous about this, Post's (1947) model did not require sequential instruction execution.

His extremely simple model can emulate any Turing machine, and although his 1936 *Formulation 1* does not use the word "program" or "machine", it is effectively a formulation of a very primitive programmable computer and associated programming language, with the boxes acting as an unbounded bitstring memory, and the set of instructions constituting a program.

Wang machines

In an influential paper, Hao Wang (1954, 1957) reduced Post's "formulation 1" to machines that still use a two-way infinite binary tape, but whose instructions are simpler — being the "atomic" components of Post's instructions — and are by default executed sequentially (like a "computer program"). His stated principal purpose was to offer, as an alternative to Turing's theory, one that "is more economical in the basic operations". His results were "program formulations" of a variety of such machines, including the 5-instruction Wang **W-machine** with the instruction-set

> { SHIFT-LEFT, SHIFT-RIGHT, MARK-SQUARE, ERASE-SQUARE, JUMP-IF-SQUARE-MARKED-to xxx }

and his most-severely reduced 4-instruction Wang B-machine ("B" for "basic") with the instruction-set

> { SHIFT-LEFT, SHIFT-RIGHT, MARK-SQUARE, JUMP-IF-SQUARE-MARKED-to xxx }

which has not even an ERASE-SQUARE instruction.

Many authors later introduced variants of the machines discussed by Wang:

Minsky (1961) evolved Wang's notion with his version of the (multi-tape) "counter machine" model that allowed SHIFT-LEFT and SHIFT-RIGHT motion of the separate heads but no printing at all. In this case the tapes would be left-ended, each end marked with a single "mark" to indicate the end. He was able to reduce this to a single tape, but at the expense of introducing multi-tape-square motion equivalent to multiplication and division rather than the much simpler { SHIFT-LEFT = DECREMENT, SHIFT-RIGHT = INCREMENT }.

Davis, adding an explicit HALT instruction to one of the machines discussed by Wang, used a model with the instruction-set

> { SHIFT-LEFT, SHIFT-RIGHT, ERASE, MARK, JUMP-IF-SQUARE-MARKED-to xxx, JUMP-to xxx, HALT }

and also considered versions with tape-alphabets of size larger than 2.

Böhm's theoretical machine language P"

In keeping with Wang's project to seek a Turing-equivalent theory "economical in the basic operations", and wishing to avoid unconditional jumps, a notable theoretical language is the 4-instruction language P" introduced by Corrado Böhm in 1964 — the first "GOTO-less" imperative "structured programming" language to be proved Turing-complete.

Multi-tape Turing machines

In practical analysis, various types of **multi-tape Turing machines** are often used. Multi-tape machines are similar to single-tape machines, but there is some constant *k* number of independent tapes.

The TABLE has full independent control over all the heads, any of all of which move and print/erase their own tapes (cf Aho-Hopcroft-Ullman 1974 p. 26). Most models have tapes with left ends, right ends unbounded.

This model intuitively seems much more powerful than the single-tape model, but any multi-tape machine, no matter how large the *k*, can be simulated by a single-tape machine using only quadratically more computation time (Papadimitriou 1994, Thrm 2.1). Thus, multi-tape machines cannot calculate any more functions than single-tape machines, and none of the robust complexity classes (such as polynomial time) are affected by a change between single-tape and multi-tape machines.

Two-stack Turing machine

Two-stack Turing machines have a read-only input and two storage tapes. If a head moves left on either tape a blank is printed on that tape, but one symbol from a "library" can be printed.

Formal definition: multi-tape Turing machine

A k-tape Turing machine can be described as a 6-tuple $M = \langle Q, \Gamma, s, b, F, \delta \rangle$ where

- Q is a finite set of states
- Γ is a finite set of the tape alphabet
- $s \in Q$ is the initial state
- $b \in \Gamma$ is the blank symbol
- $F \subseteq Q$ is the set of final or accepting states
- $\delta : Q \times \Gamma^k \to Q \times (\Gamma \times \{L, R, S\})^k$ is a partial function called the transition function, where L is left shift, R is right shift, S is no shift.

Deterministic and non-deterministic Turing machines

If the action table has at most one entry for each combination of symbol and state then the machine is a "deterministic Turing machine" (DTM). If the action table contains multiple entries for a combination of symbol and state then the machine is a "non-deterministic Turing machine" (NDTM). The two are computationally equivalent, that is, it is possible to turn any NDTM into a DTM (and *vice versa*).

Oblivious Turing machines

An **oblivious** Turing machine is a Turing machine where movement of the various heads are fixed functions of time, independent of the input. In other words, there is a predetermined sequence in which the various tapes are scanned, advanced, and written to. Pippenger and Fischer (1979) showed that any computation that can be performed by a multi-tape Turing machine in *n* steps can be performed by an oblivious two-tape Turing machine in *O(n log n)* steps.

Register machine models

van Emde Boas (1990) includes all machines of this type in one category (group, class, collection) -- "the register machine". However, historically the literature has also called the most primitive member of this group i.e. "the counter machine" -- "the register machine". And the most primitive embodiment of a "counter machine" is sometimes called the "Minsky machine".

The "counter machine", also called a "register machine" model

The primitive model register machine is, in effect, a multitape 2-symbol Post-Turing machine with its behavior restricted so its tapes act like simple "counters".

By the time of Melzak, Lambek, and Minsky (all 1961) the notion of a "computer program" produced a different type of simple machine with many left-ended tapes cut from a Post-Turing tape. In all cases the models permit only two tape symbols { mark, blank }.

Some versions represent the positive integers as only a strings/stack of marks allowed in a "register" (i.e. left-ended tape), and a blank tape represented by the count "0". Minsky (1961) eliminated the PRINT instruction at the expense of providing his model with a mandatory single mark at the left-end of each tape.

In this model the single-ended tapes-as-registers are thought of as "counters", their instructions restricted to only two (or three if the TEST/DECREMENT instruction is atomized). Two common instruction sets are the following:

(1): { INC (r), DEC (r), JZ (r,z) }, i.e.
{ INCrement contents of register #r; DECrement contents of register #r; IF contents of #r=Zero THEN Jump-to Instruction #z}
(2): { CLR (r); INC (r); JE (r_i, r_j, z) }, i.e.
{ CLeaR contents of register r; INCrement contents of r; compare contents of r_i to r_j and if Equal then Jump to instruction z}

Although his model is more complicated than this simple description, the Melzak "pebble" model extended this notion of "counter" to permit multi- pebble adds and subtracts.

The Random Access Machine (RAM) model

Melzak (1961) recognized a couple serious defects in his register/counter-machine model: (i) Without a form of indirect addressing he would be not be able to "easily" show the model is Turing equivalent, (ii) The program and registers were in different "spaces", so self-modifying programs would not be easy. When Melzak added indirect addressing to his model he created a random access machine model.

(However, with Gödel numbering of the instructions Minsky (1961) offered a proof that with such numbering the general recursive functions were indeed possible; in Minsky (1967) he offers proof that μ recursion is indeed possible).

Unlike the RASP model, the RAM model does not allow the machine's actions to modify its instructions. Sometimes the model works only register-to-register with no accumulator, but most models seem to include an accumulator.

van Emde Boas (1990) divides the various RAM models into a number of sub-types:

- SRAM, the "successor RAM" with only one arithmetic instruction, the successor (INCREMENT h). The others include "CLEAR h", and an IF equality-between-register THEN jump-to xxx.
- RAM: the standard model with addition and subtraction
- MRAM: the RAM agumented with multiplication and division
- BRAM, MBRAM: Bitwise Boolean versions of the RAM and MRAM
- N****: Non-deterministic versions of any of the above with an N before the name

The Random Access Stored Program (RASP) machine model

The RASP is a RAM with the instructions stored together with their data in the same 'space' -- i.e. sequence of registers. The notion of a RASP was described at least as early as Kiphengst (1959). His model had a "mill" -- an accumulator, but now the instructions were in the registers with the data—the so-called von Neumann architecture. When the RASP has alternating even and odd registers—the even holding the "operation code" (instruction) and the odd holding its "operand" (parameter), then indirect addressing is achieved by simply modifying an instruction's operand (cf Cook and Reckhow 1973).

The original RASP model of Elgot and Robinson (1964) had only three instructions in the fashion of the register-machine model, but they placed them in the register space together with their data. (Here COPY takes the place of CLEAR when one register e.g. "z" or "0" starts with and always contains 0. This trick is not unusual. The unit 1 in register "unit" or "1" is also useful.)

$$\{ \text{INC} (r), \text{COPY} (r_i, r_j), \text{JE} (r_i, r_i, z) \}$$

The RASP models allow indirect as well as direct-addressing; some allow "immediate" instructions too, e.g. "Load accumulator with the constant 3". The instructions may be of a highly-restricted set such as the following 16 instructions of Hartmanis (1971). This model uses an accumulator A. The mnemonics are those that the authors used (their CLA is "load accumulator" with constant or from register; STO is "store accumulator"). Their syntax is the following, excepting the jumps: "n, <n>, <<n>>" for "immediate", "direct" and "indirect"). Jumps are via two "Transfer instructions" TRA—unconditional jump by directly "n" or indirectly "< n >" jamming contents of register n into the instruction counter, TRZ (conditional jump if Accumulator is zero in the same manner as TRA):

{ ADD n , ADD < n >, ADD << n >>, SUB n, SUB < n >, SUB << n >>, CLA n, CLA < n >, CLA << n >>, STO < n >, STO << n >>, TRA n, TRA < n >, TRZ n, TRA < n >, HALT }

The Pointer machine model

A relative late-comer is Schönhage's Storage Modification Machine (1970) or pointer machine. Another version is the Kolmogorov-Uspensii machine, and the Knuth "linking

automaton" proposal. Like a state-machine diagram, a node emits at least two labelled "edges" (arrows) that point to another node or nodes which in turn point to other nodes, etc. The outside world points at the center node.

Machines with input and output

Any of the above tape-based machines can be equipped with input and output tapes; any of the above register-based machines can be equipped with dedicated input and output registers. For example, the Schönhage pointer-machine model has two instructions called "*input* λ_0,λ_1" and "*output* β" (Schönhage 1990 p. 493)

It is difficult to study sublinear space complexity on multi-tape machines with the traditional model, because an input of size n already takes up space n. Thus, to study small DSPACE classes, we must use a different model. In some sense, if we never "write to" the input tape, we don't want to charge ourself for this space. And if we never "read from" our output tape, we don't want to charge ourself for this space.

We solve this problem by introducing a ***k*-string Turing machine with input and output**. This is the same as an ordinary k-string Turing machine, except that the transition function δ is restricted so that the input tape can never be changed, and so that the output head can never move left. This model allows us to define deterministic space classes smaller than linear. Turing machines with input-and-output also have the same time complexity as other Turing machines; in the words of Papaditriou 1994 Prop 2.2:

For any k-string Turing machine M operating within time bound $f(n)$ there is a $(k+2)$-string Turing machine M' with input and output, which operates within time bound $O(f(n))$.

k-string Turing machines with input and output are used in the formal definition of the complexity resource DSPACE in, for example, Papadimitriou 1994 (Def. 2.6).

Other equivalent machines and methods

- Multidimensional Turing machine: For example, a model by Schönhage (1990) uses the four head-movement commands { **N**orth, **S**outh, **E**ast, **W**est }.

- Single-tape, multi-head Turing machine: In an undecidability proof of the "problem of tag", Minsky 1961 and Shepherdson and Sturgis (1963) described machines with a single tape that could write along the tape with one head and read further along the tape with another.

- Markov's (1954) Normal Algorithm is another remarkably simple computational model equivalent to the Turing machines.

- Lambda calculus

- Queue machine

Turing machine examples

Turing's very first example

The following table is Turing's very first example (Turing 1937):

"1. A machine can be constructed to compute the sequence 0 1 0 1 0 1..." (0 <blank> 1 <blank> 0...) (*Undecidable* p. 119)

Configuration		Behavior	
m-configuration (state)	**Tape symbol**	**Tape operations**	**Final m-configuration (state)**
b	blank	P0, R	c
c	blank	R	e
e	blank	P1, R	f
f	blank	R	b

With regard to what actions the machine actually does, Turing (1936) (*Undecidable* p. 121) states the following:

"This [example] table (and all succeeding tables of the same kind) is to be understood to mean that for a configuration described in the first two columns the operations in the third column are carried out successively, and the machine then goes over into the m-configuration in the final column." (Undecidable p. 121)

He makes this very clear when he reduces the above table to a single instruction called "b" (*Undecidable* p. 120), but his instruction consists of 3 lines. Instruction "b" has three different symbol possibilities {None, 0, 1}. Each possibility is followed by a sequence of actions until we arrive at the rightmost column, where the final m-configuration is "b":

Current m-configuration (instruction)	**Tape symbol**	**Operations on the tape**	**Final m-configuration (instruction)**
b	None	P0	b
b	0	R, R, P1	b
b	1	R, R, P0	b

As observed by a number of commentators including Turing (1937) himself, (e.g., Post (1936), Post (1947), Kleene (1952), Wang (1954)) the Turing instructions are not atomic

— further simplifications of the model can be made without reducing its computational power.

As stated in Turing machine, Turing proposed that his table be further atomized by allowing only a single print/erase followed by a single tape movement L/R/N. He gives us this example of the first little table converted (*Undecidable*, p. 127):

Current m-configuration (Turing state)	Tape symbol	Print-operation	Tape-motion	Final m-configuration (Turing state)
q_1	blank	P0	R	q_2
q_2	blank	P blank, i.e. E	R	q_3
q_3	blank	P1	R	q_4
q_4	blank	P blank, i.e. E	R	q_1

Turing's statement still implies five atomic operations. At a given instruction (m-configuration) the machine:

1. observes the tape-symbol underneath the head
2. based on the observed symbol goes to the appropriate instruction-sequence to use
3. prints symbol S_j or erases or does nothing
4. moves tape left, right or not at all
5. goes to the final m-configuration for that symbol

Because a Turing machine's actions are not atomic, a simulation of the machine must atomize each 5-tuple into a sequence of simpler actions. One possibility — used in the following examples of "behaviors" of his machine — is as follows:

(q_i) Test tape-symbol under head: If the symbol is S_0 go to $q_i.01$, if symbol S_1 go to $q_i.11$, if symbol S_2 go to $q_i.21$, etc.
($q_i.01$) print symbol S_j0 or erase or do nothing then go to $q_i.02$
($q_i.02$) move tape left or right nor not at all then go to qm0
($q_i.11$) print symbol S_j1 or erase or do nothing then go to $q_i.12$
($q_i.12$) move tape left or right nor not at all then go to qm1
($q_i.21$) print symbol S_j2 or erase or do nothing then go to $q_i.22$
($q_i.22$) move tape left or right nor not at all then go to qm2
(etc — all symbols must be accounted for)

So-called "canonical" finite state machines do the symbol tests "in parallel".

In the following example of what the machine does, we will note some peculiarities of Turing's models:

"The convention of writing the figures only on alternate squares is very useful: I shall always make use of it." (Undecidable p. 121)

Thus when printing he skips every other square. The printed-on squares are called F-squares; the blank squares in between may be used for "markers" and are called "E-squares" as in "liable to erasure." The F-squares in turn are his "Figure squares" and will only bear the symbols 1 or 0 — symbols he called "figures" (as in "binary numbers").

In this example the tape starts out "blank", and the "figures" are then printed on it. For brevity only the TABLE-states are shown here:

Sequence	**Instruction identifier**	**Head**
		. . . .
1	**1**	. . . .
2	**2**	. . . . . **0** . . . . .
3	**3**	. . . . . . **0** . . . . .
4	**4**	. . . . . **1** . **0** . . . . .
5	**1**	. . . . . . **1** . **0** . . . . .
6	**2**	. . . . . **0** . **1** . **0** . . . . .
7	**3**	. . . . . . **0** . **1** . **0** . . . . .
8	**4**	. . . . . **1** . **0** . **1** . **0** . . . .
9	**1**	. . . . . . **1** . **0** . **1** . **0** . . . .
10	**2**	. . . . . **0** . **1** . **0** . **1** . **0** . . .
11	**3**	. . . . . . **0** . **1** . **0** . **1** . **0** . .
12	**4**	. . . . . **1** . **0** . **1** . **0** . **1** . **0** .
13	**1**	. . . . . . **1** . **0** . **1** . **0** . **1** . **0**
14	**2**	. . . . . **0** . **1** . **0** . **1** . **0** . **1** . **0**

The same "run" with all the intermediate tape-printing and movements is shown here:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE
1																															Current instruction #:
2																															Instruction:
3																															If Sj=blank, Jump-to #:
4																															If Sj=0, Jump-to #:
5															TAPE																If Sj=1, Jump-to #:
6	Inst														Head															Next Inst #	Current Inst #
7		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1	
8	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		2	1
9	P0	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	.	.	.	.	.	.	.	.	.	.	.	.		3	2
10	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	.	.	.	.	.	.	.	.	.	.	.		6	3
11	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	.	.	.	.	.	.	.	.	.	.	.		7	6
12	E	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	.	.	.	.	.	.	.	.	.	.	.		8	7
13	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	.	.	.	.	.	.	.	.	.	.		11	8
14	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	.	.	.	.	.	.	.	.	.	.		12	11
15	P1	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	.	.	.	.	.	.	.	.	.	.		13	12
16	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	.	.	.	.	.	.	.	.	.		16	13
17	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	.	.	.	.	.	.	.	.	.		17	16
18	E	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	.	.	.	.	.	.	.	.	.		18	17
19	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	.	.	.	.	.	.	.	.		1	18
20	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	.	.	.	.	.	.	.	.		2	1
21	P0	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	.	.	.	.	.	.	.	.		3	2
22	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	.	.	.	.	.	.	.		6	3
23	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	.	.	.	.	.	.	.		7	6
24	E	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	.	.	.	.	.	.	.		8	7
25	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	.	.	.	.	.	.		11	8
26	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	.	.	.	.	.	.		12	11
27	P1	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	.	.	.	.	.	.		13	12
28	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	.	.	.	.	.		16	13
29	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	.	.	.	.	.		17	16
30	E	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	.	.	.	.	.		18	17
31	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	.	.	.	.		1	18
32	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	.	.	.	.		2	1
33	P0	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	1	.	0	.	.	.	.	.		3	2
34	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	1	.	0	.	.	.	.		6	3
35	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	1	.	0	.	.	.	.		7	6
36	E	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	1	.	0	.	.	.	.		8	7
37	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	1	.	0	.	.	.		11	8
38	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	1	.	0	.	.	.		12	11
39	P1	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	1	.	0	.	.	.		13	12
40	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	1	.	0	.	.		16	13
41	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	1	.	0	.	.		17	16
42	E	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	1	.	0	.	.		18	17
43	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	1	.	0	.		1	18
44	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	0	.	1	.	0	.	1	.	0	.		2	1
45	P0	.	.	.	.	.	.	.	.	.	.	.	.	.	0	.	1	.	0	.	1	.	0	.	1	.	0	.		3	2

	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO	AP	AQ	AR	AS	AT	AU	AV	AW
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	1	P0	R			2	E	R			3	P1	R			4	E	R
3	2	3	6			7	8	11			12	13	16			17	18	1
4	4	3	6			9	8	11			14	13	16			19	18	1
5	4	3	6			9	8	11			14	13	16			19	18	1
8	1																	
9		P0																
10			R															
11						2												
12							E											
13								R										
14											3							
15												P1						
16													R					
17																4		
18																	E	
19																		R
20	1																	
21		P0																
22			R															
23						2												
24							E											
25								R										
26											3							
27												P1						
28													R					
29																4		
30																	E	
31																		R
32	1																	
33		P0																
34			R															
35						2												
36							E											
37								R										
38											3							
39												P1						
40													R					
41																4		
42																	E	
43																		R
44	1																	
45		P0																

A close look at the table reveals certain problems with Turing's own example—not all the symbols are accounted for.

For example, suppose his tape was not initially blank. What would happen? The Turing machine would read different values than the intended values.

A copy subroutine

This is a very important subroutine used in the "multiply" routine.

The example Turing machine handles a string of 0s and 1s, with 0 represented by the blank symbol. Its task is to double any series of 1s encountered on the tape by writing a 0 between them. For example, when the head reads "111", it will write a 0, then "111". The output will be "1110111".

In order to accomplish its task, this Turing machine will need only 5 states of operation, which are called $\{s_1, s_2, s_3, s_4, s_5\}$. Each state does 4 actions:

1. Read the symbol under the head
2. Write the output symbol decided by the state
3. Move the tape to the left or to the right decided by the state
4. Switch to the following state decided by the current state

Initial m-configuration (current instruction)	Tape symbol	Print operation	Tape motion	Final m-configuration (next instruction)
s_1	0	N	N	H
s_1	1	E	R	s_2
s_2	0	E	R	s_3
s_2	1	P1	R	s_2
s_3	0	P1	L	s_4
s_3	1	P1	R	s_3
s_4	0	E	L	s_5
s_4	1	P1	L	s_4
s_5	0	P1	R	s_1
s_5	1	P1	L	s_5
H	-	-	-	

A "run" of the machine sequences through 16 machine-configurations (aka Turing states):

Sequence	Instruction identifier		Head	
1	s_1	0 0 0 0 **1**	**1**	0 0 0 0 0
2	s_2	0 0 0 0 0	**1**	0 0 0 0 0
3	s_2	0 0 0 0 0	0	**1** 0 0 0 0
4	s_3	0 0 0 0 0	0	0 **1** 0 0 0
5	s_4	0 0 0 0 **1**	0	**1** 0 0 0 0
6	s_5	0 0 0 **1** 0	**1**	0 0 0 0 0
7	s_5	0 0 **1** 0 **1**	0	0 0 0 0 0
8	s_1	0 0 0 **1** 0	**1**	**1** 0 0 0 0
9	s_2	0 0 0 0 **1**	0	0 **1** 0 0 0
10	s_3	0 0 0 0 0	**1**	0 0 **1** 0 0
11	s_3	0 0 0 0 0	0	**1** 0 0 **1** 0
12	s_4	0 0 0 0 **1**	**1**	0 0 **1** 0 0
13	s_4	0 0 0 **1 1**	0	0 **1** 0 0 0
14	s_5	0 0 **1 1** 0	0	**1** 0 0 0 0
15	s_1	0 0 0 **1 1**	0	**1 1** 0 0 0
16	H	0 0 0 **1 1**	0	**1 1** 0 0 0

The behavior of this machine can be described as a loop: it starts out in s_1, replaces the first 1 with a 0, then uses s_2 to move to the right, skipping over 1s and the first 0 encountered. s_3 then skips over the next sequence of 1s (initially there are none) and replaces the first 0 it finds with a 1. s_4 moves back to the left, skipping over 1s until it

finds a 0 and switches to s_5. s_5 then moves to the left, skipping over 1s until it finds the 0 that was originally written by s_1.

It replaces that 0 with a 1, moves one position to the right and enters s_1 again for another round of the loop.

This continues until s_1 finds a 0 (this is the 0 in the middle of the two strings of 1s) at which time the machine halts.

Alternative description

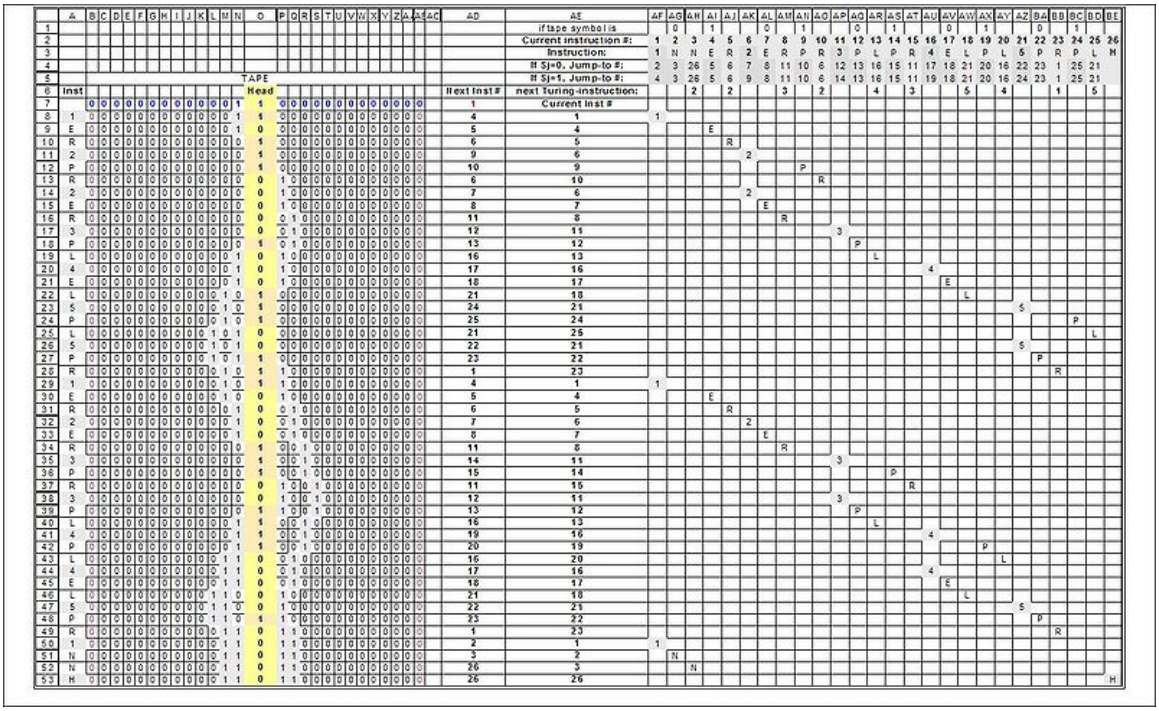
Another description sees the problem as how to keep track of how many "1"s there are. We can't use one state for each possible number (a state for each of 0,1,2,3,4,5,6 etc), because then we'd need infinite states to represent all the natural numbers, and the state machine is *finite* - we'll have to track this using the tape in some way.

The basic way it works is by copying each "1" to the other side, by moving back and forth - it is intelligent enough to remember which part of the trip it is on. In more detail, it carries each "1" across to the other side, by recognizing the separating "0" in the middle, and recognizing the "0" on the other side to know it's reached the end. It comes back using the same method, detecting the middle "0", and then the "0" on the original side. This "0" on the original side is the key to the puzzle of how it keeps track of the number of 1's.

The trick is that before carrying the "1", it marks that digit as "taken" by replacing it with an "0". When it returns, it fills that "0" back in with a "1", *then moves on to the next one*, marking it with an "0" and repeating the cycle, carrying that "1" across and so on. *With each trip across and back, the marker "0" moves one step closer to the centre*. This is how it keeps track of how many "1"'s it has taken across.

Interestingly, when it returns, the marker "0" looks like the end of the collection of "1"s to it - any "1"s that have already been taken across are invisible to it (on the other side of the marker "0") and so it is as if it is working on an (N-1) number of "1"s - similar to a proof by Mathematical induction.

A full "run" showing the results of the intermediate "motions".

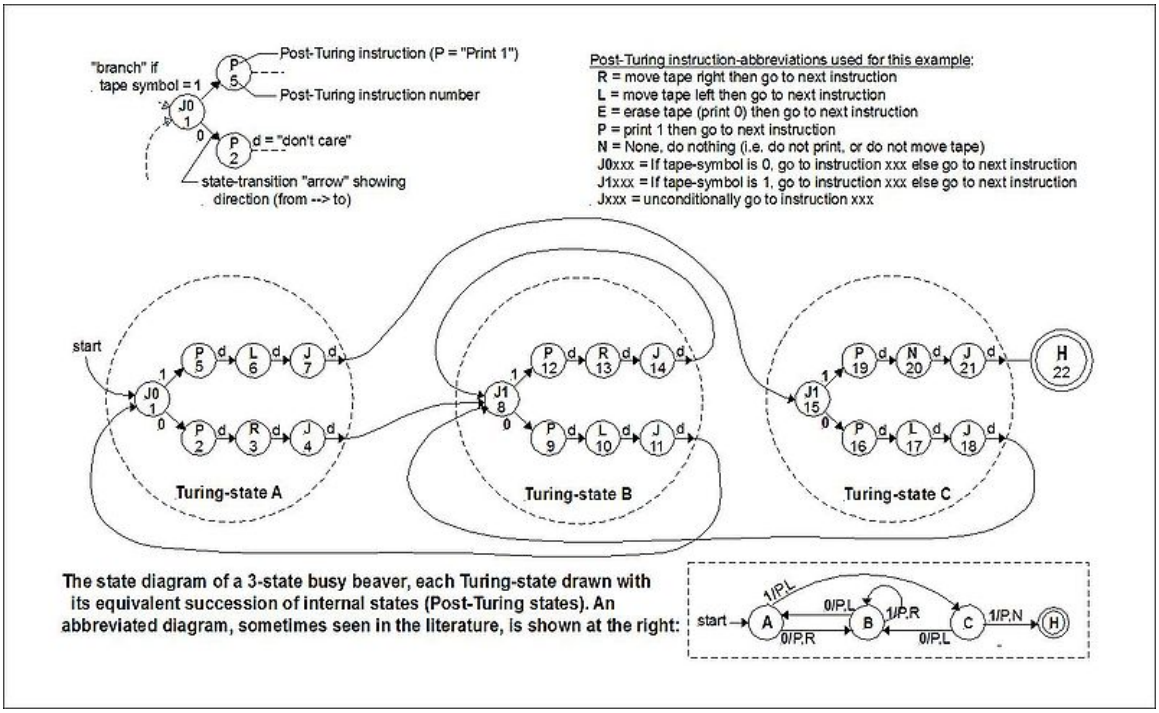


3-state Busy Beaver

The following Turing table of instructions was derived from Peterson (1988) page 198, Figure 7.15. Peterson moves the head; in the following model the tape moves.

Tape symbol	**Current state A**			**Current state B**			**Current state C**		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	1	R	**B**	1	L	**A**	1	L	**B**
1	1	L	**C**	1	R	**B**	1	N	**HALT**

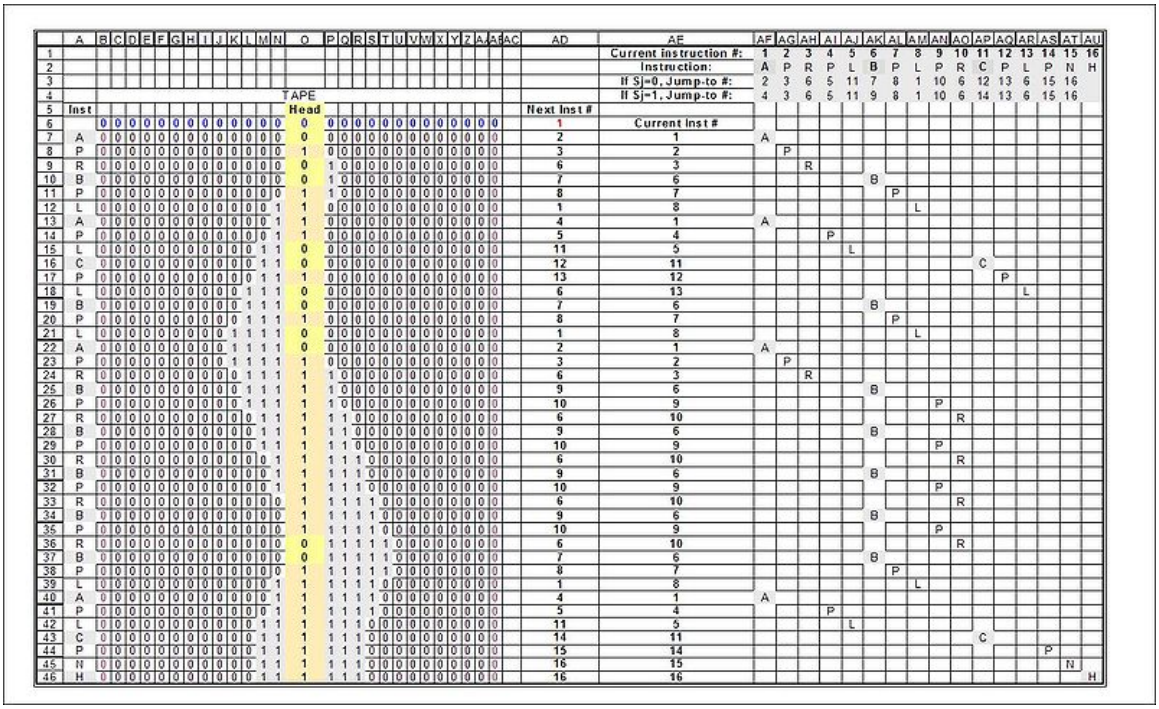
The "state" drawing of the 3-state busy beaver shows the internal sequences of events required to actually perform "the state". As noted above Turing (1937) makes it perfectly clear that this is the proper interpretation of the 5-tuples that describe the instruction (*Undecidable*, p. 119).



The following table shows the "compressed" run — just the Turing states:

Sequence	Instruction identifier		Head	
1	**A**	0 0 0 0 0 0	**0**	0 0 0 0 0 0 0
2	**B**	0 0 0 0 0 0	**0**	1 0 0 0 0 0 0
3	**A**	0 0 0 0 0 **1**	**1**	0 0 0 0 0 0 0
4	**C**	0 0 0 0 **1 1**	**0**	0 0 0 0 0 0 0
5	**B**	0 0 0 **1 1 1**	**0**	0 0 0 0 0 0 0
6	**A**	0 0 **1 1 1 1**	**0**	0 0 0 0 0 0 0
7	**B**	0 0 0 **1 1 1**	**1**	1 0 0 0 0 0 0
8	**B**	0 0 0 0 **1 1**	**1**	**1 1** 0 0 0 0 0
9	**B**	0 0 0 0 0 **1**	**1**	**1 1 1** 0 0 0 0
10	**B**	0 0 0 0 0 0	**1**	**1 1 1 1** 0 0 0
11	**B**	0 0 0 0 0 0	**0**	**1 1 1 1 1** 0 0
12	A	0 0 0 0 0 **1**	**1**	**1 1 1 1** 0 0 0
13	**C**	0 0 0 0 **1 1**	**1**	**1 1 1** 0 0 0 0
14	**H**	0 0 0 0 **1 1**	**1**	**1 1 1** 0 0 0 0

The full "run" of the 3-state busy beaver. The resulting Turing-states (what Turing called the "m-configurations" — "machine-configurations") are shown highlighted in grey in column A, and also under the machine's instructions (columns AF-AU)):



Wolfram Turing machine examples

In 1985, Stephen Wolfram conjectured that a one-dimensional two-state cellular automaton he later called Rule 110 was Turing complete (universal). Matthew Cook verified that conjecture in 2000. Wolfram's book *A New Kind of Science* gives useful examples of how Turing machines can be visually represented, and further conjectured that an even simpler 2-state 3-symbol Turing machine might also be universal. In October 2007, Wolfram Research awarded Alex Smith a $25,000 prize for verifying the second conjecture, though Vaughan Pratt disputed the validity of the proof, which later Wolfram Research and Smith himself clarified. Pratt's main argument is that the proof argument would make a Linear Bounded Automata (LBA) universal. Smith later explained however that the LBA would need to be restarted at running time to perform a computation, while the 2, 3 Turing machine would restart automatically, therefore the proof does not make an LBA universal as Pratt first thought.

Chapter 4

Post–Turing Machine and Langton's Ant

Post–Turing machine

A **Post–Turing machine** is a "program formulation" of an especially simple type of Turing machine, comprising a variant of Emil Post's Turing-equivalent model of computation described below. (Post's model and Turing's model, though very similar to one another, were developed independently. Turing's paper was received for publication in May of 1936, followed by Post's in October.) A Post–Turing machine uses a binary alphabet, an infinite sequence of binary storage locations, and a primitive programming language with instructions for bi-directional movement among the storage locations and alteration of their contents one at a time. The names "Post–Turing program" and "Post–Turing machine" were used by Martin Davis in 1973–1974 (Davis 1973, p.69ff). Later in 1980, Davis used the name "Turing–Post program" (Davis, in Steen p. 241).

1936: Post model

In his 1936 paper "Finite combinatory processes—formulation 1" (which can be found on page 289 of *The Undecidable*), Emil Post described a model of extreme simplicity which he conjectured is "logically equivalent to recursiveness", and which was later proved to be so. The quotes in the following are from this paper.

Post's model of a computation differs from the Turing-machine model in a further "atomization" of the acts a human "computer" would perform during a computation.

Post's model employs a "symbol space" consisting of a "two-way infinite sequence of spaces or boxes", each box capable of being in either of two possible conditions, namely "marked" (as by a single vertical stroke) and "unmarked" (empty). Initially, finitely-many of the boxes are marked, the rest being unmarked. A "worker" is then to move among the boxes, being in and operating in only one box at a time, according to a fixed finite "set of directions" (instructions), which are numbered in order (1,2,3,...,n). Beginning at a box "singled out as the starting point", the worker is to follow the set of instructions one at a time, beginning with instruction 1.

The instructions may require the worker to perform the following "basic acts" or "operations":

(a) *Marking the box he is in (assumed empty),*
(b) *Erasing the mark in the box he is in (assumed marked),*
(c) *Moving to the box on his right,*
(d) *Moving to the box on his left,*
(e) *Determining whether the box he is in, is or is not marked.*

Specifically, the i^{th} "direction" (instruction) given to the worker is to be one of the following forms:

(A) *Perform operation* O_i [O_i = (a), (b), (c) *or* (d)] *and then follow direction* j_i,
(B) *Perform operation* (e) *and according as the answer is yes or no correspondingly follow direction* j_i' *or* j_i'' ,
(C) *Stop*.

(The above indented text and italics are as in the original.) Post remarks that this formulation is "in its initial stages" of development, and mentions several possibilities for "greater flexibility" in its final "definitive form", including

(1) replacing the infinity of boxes by a finite extensible symbol space, "extending the primitive operations to allow for the necessary extension of the given finite symbol space as the process proceeds",
(2) using an alphabet of more than two symbols, "having more than one way to mark a box",
(3) introducing finitely-many "physical objects to serve as pointers, which the worker can identify and move from box to box".

1947: Post's formal reduction of the Turing 5-tuples to 4-tuples

As briefly mentioned in Turing machine, Post, in his paper of 1947 (*Recursive Unsolvability of a Problem of Thue*) atomized the Turing 5-tuples to 4-tuples:

"Our quadruplets are quintuplets in the Turing development. That is, where our standard instruction orders either a printing (overprinting) **or** motion, left or right, Turing's standard instruction always order a printing **and** a motion, right, left, or none"(footnote 12, Undecidable p. 300)

Like Turing he defined erasure as printing a symbol "S0". And so his model admitted quadruplets of only three types (cf p. 294 *Undecidable*):

$q_i\ S_j\ L\ q_l$,
$q_i\ S_j\ R\ q_l$,
$q_i\ S_j\ S_k\ q_l$

At this time he was still retaining the Turing state-machine convention – he had not formalized the notion of an assumed *sequential* execution of steps until a specific test of a symbol "branched" the execution elsewhere.

1954, 1957: Wang model

For an even further reduction – to only four instructions – of the Wang model presented here see Wang B-machine.

Wang (1957, but presented to the ACM in 1954) is often cited (cf Minsky (1967) p. 200) as the source of the "program formulation" of binary-tape Turing machines using numbered instructions from the set

write 0
write 1
move left
move right
if scanning 0 then goto instruction *i*
if scanning 1 then goto instruction *j*

where *sequential execution* is assumed, and Post's single "if ... then ... else" has been "atomised" into two "if ... then ..." statements. (Here '1' and '0' are used where Wang used "marked" and "unmarked", respectively, and the initial tape is assumed to contain only '0's except for finitely-many '1's.)

Wang noted the following:

- "Since there is no separate instruction for halt (stop), it is understood that the machine will stop when it has arrived at a stage that the program contains no instruction telling the machine what to do next." (p.65)
- "In contrast with Turing who uses a one-way infinite tape that has a beginning, we are following Post in the use of a 2-way infinite tape." (p. 65)
- Unconditional gotos are easily derived from the above instructions, so "we can freely use them too". (p.84)

Any binary-tape Turing machine is readily converted to an equivalent "Wang program" using the above instructions.

1974: first Davis model

Martin Davis was a undergraduate student of Emil Post's. Along with Stephen Kleene he completed his PhD under Alonzo Church (Davis (2000) 1st and 2nd footnotes p. 188).

The following model he presented in a series of lectures to the Courant Institute at NYU in 1973–1974. This is the model to which Davis formally applied the name "Post–Turing machine" with its "Post–Turing language". The instructions are assumed to be executed sequentially (Davis 1974, p. 71):

"Write 1
"Write B
"To A if read 1
"To A if read B
"RIGHT
"LEFT

Note that there is no "halt" or "stop".

1978 second Davis model

The following model appears as an essay *What is a computation?* in Steen pages 241–267. For some reason Davis has renamed his model a "Turing–Post machine" (with one back-sliding on page 256.)

In the following model Davis assigns the numbers "1" to Post's "mark/slash" and "0" to the blank square. To quote Davis: "We are now ready to introduce the Turing–Post Programming Language. In this language there are seven kinds of instructions:

"PRINT 1
"PRINT 0
"GO RIGHT
"GO LEFT
"GO TO STEP i IF **1** IS SCANNED
"GO TO STEP i IF **0** IS SCANNED
"STOP

"A Turing–Post program is then a list of instructions, each of which is of one of these seven kinds. Of course in an actual program the letter *i* in a step of either the fifth or sixth kind must replaced with a definite (positive whole) number." (Davis in Steen, p. 247).

- Confusion arises if one does not realize that a "blank" tape is actually printed with all zeroes — there is no "blank".
- Splits Post's "GO TO" ("branch" or "jump") instruction into two, thus creating a larger (but easier-to-use) instruction set of seven rather than Post's six instructions.
- Does not mention that instructions PRINT 1, PRINT 0, GO RIGHT and GO LEFT imply that, after execution, the "computer" must go to the next step in numerical sequence.

1994 (2nd Edition) Davis–Sigal–Weyuker's Post–Turing program model

"Although the formulation of Turing we have presented is closer in spirit to that originally given by Emil Post, it was Turing's analysis of the computation that has made

this formulation seem so appropriate. This language has played a fundamental role in theoretical computer science." (Davis et al. (1994) p. 129)

This model allows for the printing of multiple symbols. The model allows for B (blank) instead of S_0. The tape is infinite in both directions. Either the head or the tape moves, but their definitions of RIGHT and LEFT always specify the same outcome in either case (Turing used the same convention).

PRINT σ ;Replace scanned symbol with σ
IF σ GOTO L ;IF scanned symbol is σ THEN goto "the first" instruction labelled L
RIGHT ;Scan square immediately right of the square currently scanned
LEFT ;Scan square immediately left of the square currently scanned

Note that only one type of "jump" – a conditional GOTO – is specified; for an unconditional jump a string of GOTO's must test each symbol.

This model reduces to the binary { 0, 1 } versions presented above, as shown here:

PRINT 0 = ERASE ;Replace scanned symbol with 0 = B = BLANK
PRINT 1 ;Replace scanned symbol with 1
IF 0 GOTO L ;IF scanned symbol is 0 THEN goto "the first" instruction labelled L
IF 1 GOTO L ;IF scanned symbol is 1 THEN goto "the first" instruction labelled L
RIGHT ;Scan square immediately right of the square currently scanned
LEFT ;Scan square immediately left of the square currently scanned

Examples of the Post–Turing machine

Atomizing Turing quintuples into a sequence of Post–Turing instructions

The following "reduction" (decomposition, atomizing) method – from 2-symbol Turing 5-tuples to a sequence of 2-symbol Post–Turing instructions – can be found in Minsky (1961). He states that this reduction to "a *program* ... a sequence of *Instructions*" is in the spirit of Hao Wang's B-machine (italics in original, cf Minsky (1961) p. 439).

(Minsky's reduction to what he calls "a sub-routine" results in 5 rather than 7 Post–Turing instructions. He did not atomize Wi0: "Write symbol Si0; go to new state Mi0", and Wi1: "Write symbol Si1; go to new state Mi1". The following method further atomizes Wi0 and Wi1; in all other respects the methods are identical.)

This reduction of Turing 5-tuples to Post–Turing instructions may not result in an "efficient" Post–Turing program, but it will be faithful to the original Turing-program.

In the following example, each Turing 5-tuple of the 2-state busy beaver converts into

(i) an initial conditional "jump" (goto, branch), followed by

(ii) 2 tape-action instructions for the "0" case – Print or Erase or None, followed by Left or Right or None, followed by
(iii) an unconditional "jump" for the "0" case to its next instruction
(iv) 2 tape-action instructions for the "1" case – Print or Erase or None, followed by Left or Right or None, followed by
(v) an unconditional "jump" for the "1" case to its next instruction

for a total of 1 + 2 + 1 + 2 + 1 = 7 instructions per Turing-state.

For example, the 2-state busy beaver's "A" Turing-state, written as two lines of 5-tuples, is:

Initial m-configuration (Turing state)	Tape symbol	Print operation	Tape motion	Final m-configuration (Turing state)
A	0	P	R	**B**
A	1	P	L	**B**

The table represents just a single Turing "instruction", but we see that it consists of two lines of 5-tuples, one for the case "tape symbol under head = 1", the other for the case "tape symbol under head = 0". Turing observed (Undecidable, p. 119) that the left-two columns – "m-configuration" and "symbol" – represent the machine's current "configuration" – its state including both Tape and Table at that instant – and the last three columns are its subsequent "behavior". As the machine cannot be in two "states" at once, the machine must "branch" to either one configuration or the other:

Initial m-configuration and symbol S	Print operation	Tape motion	Final m-configuration
S=0 -->	P -->	R -->	**B**
--> **A** <			
S=1 -->	P -->	L -->	**B**

After the "configuration branch" (J1 xxx) or (J0 xxx) the machine follows one of the two subsequent "behaviors". We list these two behaviors on one line, and number (or label) them sequentially (uniquely). Beneath each jump (branch, go to) we place its jump-to "number" (address, location):

	Initial m-configuration & symbol S	Print operation	Tape motion	Final m-configuration case S=0	Print operation	Tape motion	Final m-configuration case S=1
	If S=0 then:	P	R	**B**			
	---> **A** <						
	If S=1 then:				P	L	**B**
instruction #	1	2	3	4	5	6	7
Post–Turing instruction	**J1**	**P**	**R**	**J**	**P**	**L**	**J**
jump-to instruction #	**5**			**B**			**B**

Per the Post–Turing machine conventions each of the Print, Erase, Left, and Right instructions consist of two actions:

(i) Tape action: { P, E, L, R}, then
(ii) Table action: go to next instruction in sequence

And per the Post–Turing machine conventions the conditional "jumps" J0xxx, J1xxx consist of two actions:

(i) Tape action: look at symbol on tape under the head
(ii) Table action: If symbol is 0 (1) and J0 (J1) then go to xxx else go to next instruction in sequence

And per the Post–Turing machine conventions the unconditional "jump" Jxxx consists of a single action, or if we want to regularize the 2-action sequence:

(i) Tape action: look at symbol on tape under the head
(ii) Table action: If symbol is 0 then go to xxx else if symbol is 1 then go to xxx.

Which, and how many, jumps are necessary? The unconditional jump **J**xxx is simply **J0** followed immediately by **J1** (or vice versa). Wang (1957) also demonstrates that only one conditional jump is required, i.e. either **J0**xxx or **J1**xxx. However, with this restriction the machine becomes difficult to write instructions for. Often only two are used, i.e.

(i) { **J0**xxx, **J1**xxx }
(ii) { **J1**xxx, **J**xxx }
(iii) { **J0**xxx, **J**xxx },

but the use of all three { **J0**xxx, **J1**xxx, **J**xxx } does eliminate extra instructions. In the 2-state Busy Beaver example that we use only { **J1**xxx, **J**xxx }.

2-state Busy Beaver

The mission of the busy beaver is to print as many ones as possible before halting. The "Print" instruction writes a 1, the "Erase" instruction (not used in this example) writes a 0 (i.e. it is the same as P0). The tape moves "Left" or "Right" (i.e. the "head" is stationary).

State table for a 2-state Turing-machine busy beaver:

	Current state **A**:			Current state **B**:		
	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:
tape symbol is 0:	1	R	**B**	1	L	**A**
tape symbol is 1:	1	L	**B**	1	N	**H**

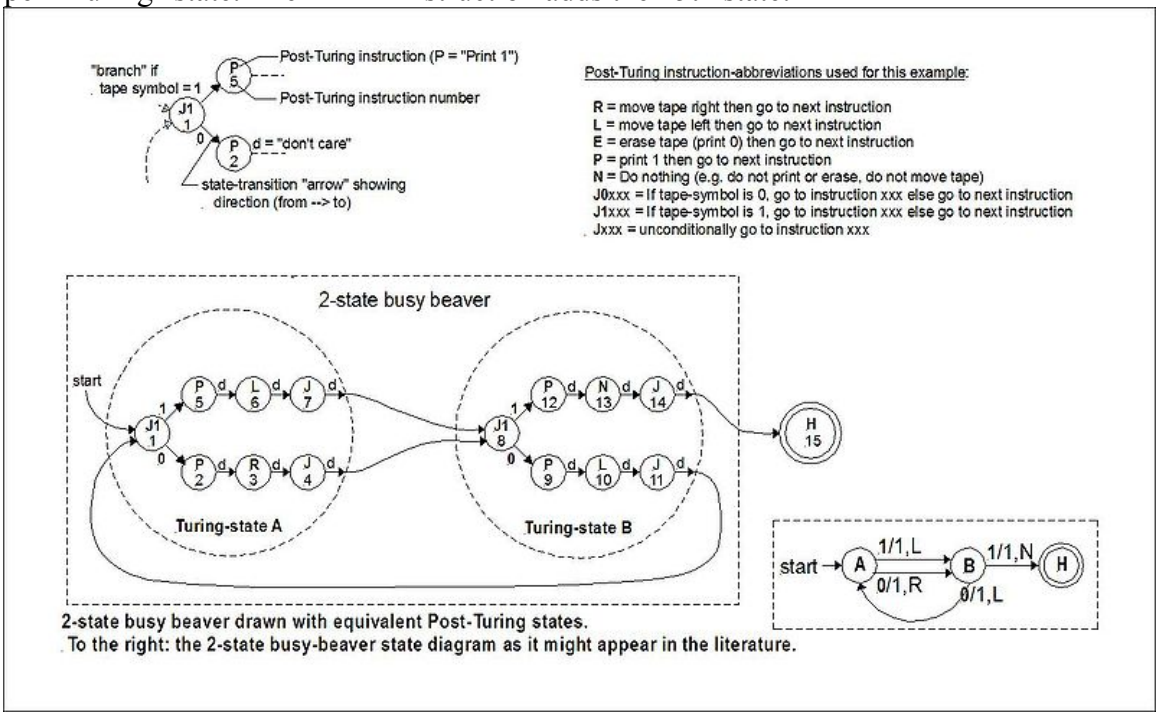
Instructions for the Post–Turing version of a 2-state busy beaver: observe that all the instructions are on the same line and in sequence. This is a significant departure from the "Turing" version and is in the same format as what is called a "computer program":

Instruction #:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Instruction:	J1	P	R	J	P	L	J	J1	P	L	J	P	N	J	H
Jump-to #:	5			8			8	12			1			15	
Turing-state label:	**A**							**B**							**H**

Alternately, we might write the table as a string. The use of "parameter separators" ":" and instruction-separators "," are entirely our choice and do not appear in the model. There are no conventions, for some useful ideas of how to combine state diagram conventions with the instructions – i.e. to use arrows to indicate the destination of the jumps). In the example immediately below, the instructions are *sequential* starting from "1", and the parameters/"operands" are considered part of their instructions/"opcodes":

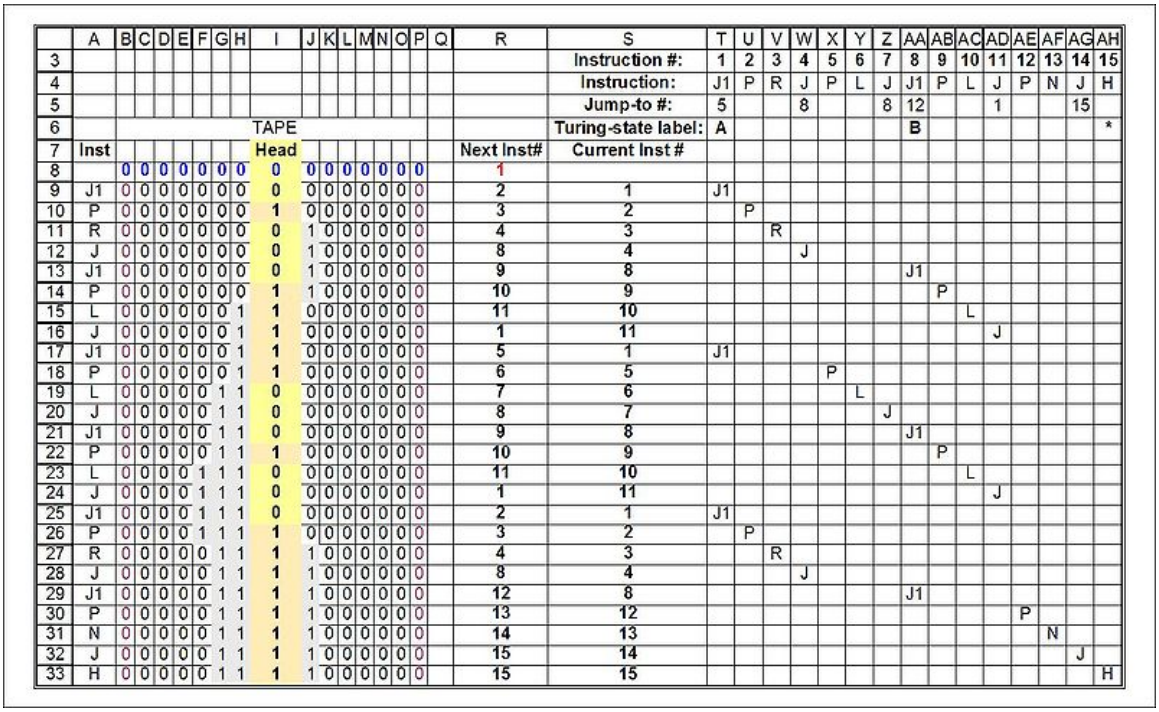
J1:5, P, R, J:8, P, L, J:8, J1:12, P, L, J1:1, P, N, J:15, H

The state diagram of a two-state busy beaver (little drawing, right-hand corner) converts to the equivalent Post–Turing machine with the substitution of 7 Post–Turing instructions per "Turing" state. The HALT instruction adds the 15th state:



2-state busy beaver drawn with equivalent Post-Turing states.
To the right: the 2-state busy-beaver state diagram as it might appear in the literature.

A "run" of the 2-state busy beaver with all the intermediate steps of the Post–Turing machine shown:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH
3																			Instruction #:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4																			Instruction:	J1	P	R	J	P	L	J	J1	P	L	J	P	N	J	H
5																			Jump-to #:	5			8			8	12			1			15	
6							TAPE												Turing-state label:	A							B							*
7	Inst								Head									Next Inst#	Current Inst #															
8		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		1																
9	J1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		2	1	J1														
10	P	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		3	2		P													
11	R	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		4	3			R												
12	J	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		8	4				J											
13	J1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		9	8								J1							
14	P	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0		10	9									P						
15	L	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		11	10										L					
16	J	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		1	11											J				
17	J1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		5	1	J1														
18	P	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		6	5					P										
19	L	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		7	6						L									
20	J	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		8	7							J								
21	J1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		9	8								J1							
22	P	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0		10	9									P						
23	L	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		11	10										L					
24	J	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		1	11											J				
25	J1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		2	1	J1														
26	P	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0		3	2		P													
27	R	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		4	3			R												
28	J	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		8	4				J											
29	J1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		12	8								J1							
30	P	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		13	12												P			
31	N	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		14	13													N		
32	J	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		15	14														J	
33	H	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		15	15															H

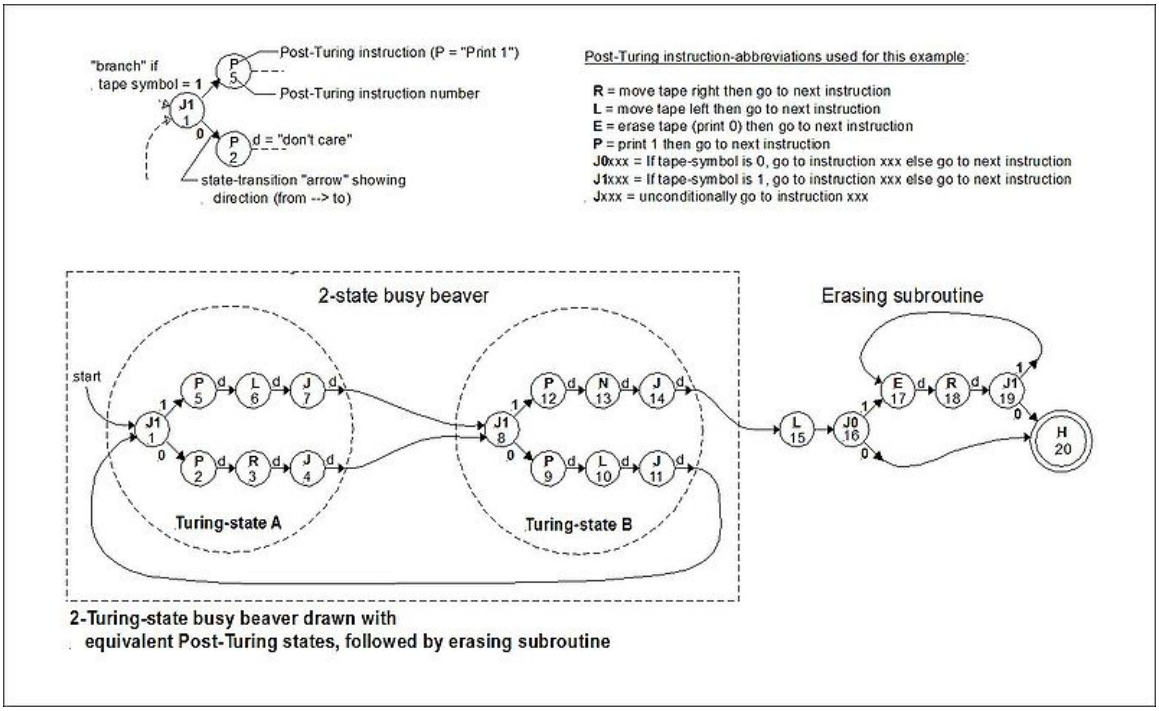
Two state busy beaver followed by "tape cleanup"

The following is a two-state Turing busy beaver with additional instructions 15–20 to demonstrate the use of "Erase", J0, etc. These will erase the 1's written by the busy beaver:

Instruction #:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Instruction:	J1	P	R	J	P	L	J	J1	P	L	J	P	N	J	L	J0	E	R	J1	H
Jump-to #:	5			8			8	12			1			15		20			17	
Turing-state label:	**A**							**B**							*					

Additional Post–Turing instructions 15 through 20 erase the symbols created by the busy beaver. These "atomic" instructions are more "efficient" than their Turing-state equivalents (of 7 Post–Turing instructions). To accomplish the same task a Post–Turing machine will (usually) require fewer Post–Turing states than a Turing-machine, because (i) a jump (go-to) can occur to any Post–Turing instruction (e.g. P, E, L, R) within the Turing-state, (ii) a grouping of move-instructions such as L, L, L, P are possible, etc.:

Instruction #:	**16**	**17**	**18**	**19**	**20**
Instruction:	J0	E	R	J1	H
Jump-to #:	20			17	



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM
3																			Instruction #:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
4																			Instruction:	J1	P	R	J	P	L	J	J1	P	L	J	P	N	J	L	J0	E	R	J1	H
5																			Jump-to #:	5			8			8	12			1			15		20			17	
6									TAPE										Turing-state label:	A							B												H
7	Inst								Head									Next Inst#	Current Inst #																				
8		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		1																					
9	J1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		2	1	J1																			
10	P	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		3	2		P																		
11	R	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		4	3			R																	
12	J	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		8	4				J																
13	J1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		9	8								J1												
14	P	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0		10	9									P											
15	L	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		11	10										L										
16	J	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		1	11											J									
17	J1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		5	1	J1																			
18	P	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		6	5					P															
19	L	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		7	6						L														
20	J	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		8	7							J													
21	J1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		9	8								J1												
22	P	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0		10	9									P											
23	L	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		11	10										L										
24	J	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		1	11											J									
25	J1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		2	1	J1																			
26	P	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0		3	2		P																		
27	R	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		4	3			R																	
28	J	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		8	4				J																
29	J1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		12	8								J1												
30	P	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		13	12												P								
31	N	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		14	13													N							
32	J	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		15	14														J						
33	L	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0		16	15															L					
34	J0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0		17	16																J0				
35	E	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		18	17																	E			
36	R	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0		19	18																		R		
37	J1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0		17	19																			J1	
38	E	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		18	17																	E			
39	R	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		19	18																		R		
40	J1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		17	19																			J1	
41	E	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0		18	17																	E			
42	R	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		19	18																		R		
43	J1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		17	19																			J1	
44	E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		18	17																	E			
45	R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		19	18																		R		
46	J1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		20	19																			J1	
47	H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		20	20																				H

Example: Multiply 3 × 4 with a Post–Turing machine

An example of "multiply" $a \times b = c$ on a Post–Turing machine. At the start, the tape (shown on the left) has two numbers on it – **a'** = 3' (4 marks), **b'** = 4' (5 marks). (A single mark would represent "0".) At the end the tape will have the product **c'** = 12' (13 marks) to the right of b. Note "top" and "bottom" are there just to clarify what the P–T machine is doing.

This example is a reference to show how a "multiply" computation would proceed on a single-tape, 2-symbol { blank, 1 } Post–Turing machine model.

This particular "multiply" algorithm is recursive through two loops. The head moves. It starts to the far left (the top) of the string of unary marks representing **a'** :

- Move head far right. Establish (i.e. "clear") register **c** by placing a single blank and then a mark to the right of **b**
- **a_loop**: Move head right once, test for the bottom of **a'** (a blank). If blank then done else erase mark;
- Move head right to **b'** . Move head right once past the top mark of **b'** ;
- **b_loop**: If head is at the bottom of **b'** (a blank) then move head to far left of **a'** , else:

- Erase a mark to locate counter (a blank) in **b'** .
- Increment **c'** : Move head right to top of **c'** and increment **c'** .
- Move head left to the counter inside **b'** ,
- Repair counter: print a mark in the blank counter.
- Decrement **b'** −count: Move head right once.
- Return to b_loop.

Multiply **a** × **b** = **c**, for example: 3 × 4 = 12. The scanned square is indicated by brackets around the mark i.e. [|]. An extra mark serves to indicate the symbol "0":
At the start of the computation **a'** is 4 unary marks, then a separator blank, **b'** is 5 unary marks, then a separator mark. An unbounded number of empty spaces must be available for **c** to the right:

....a'.b'.... = : ....**[|]** | | | . | | | | | ....

During the computation the head shuttles back and forth from **a'** to **b'** to **c'** back to **b'** then to **c'** , then back to **b'** , then to **c'** ad nauseam while the machine counts through **b'** and increments **c'** . Multiplicand **a'** is slowly counted down (its marks erased – shown for reference with x's below). A "counter" inside **b'** moves to the right through **b** (an erased mark shown being read by the head as **[.]**) but is reconstructed after each pass when the head returns from incrementing **c'** :

....**a**.**b**.... = : ....xxx | . | | **[.]** | | . | | | | | | ...

At end of computation: **c'** is 13 marks = "successor of 12" appearing to the right of **b'** . **a'** has vanished in process of the computation

....**b**.**c** = ... | | | | | . | | | | | | | | | | | | ...

Footnotes

^ a: Difference between Turing- and Post–Turing machine models

In his chapter XIII *Computable Functions*, Kleene adopts the Post model; Kleene's model uses a blank and one symbol "tally mark ¤" (Kleene p. 358), a "treatment closer in some respects to Post 1936. Post 1936 considered computation with a 2-way infinite tape and only 1 symbol" (Kleene p. 361). Kleene observes that Post's treatment provided a further

reduction to "atomic acts" (Kleene p. 357) of "the Turing act" (Kleene p. 379). As described by Kleene "The Turing act" is the combined 3 (time-sequential) actions specified on a line in a Turing table: (i) print-symbol/erase/do-nothing followed by (ii) move-tape-left/move-tape-right/do-nothing followed by (iii) test-tape-go-to-next-instruction: e.g. "s1Rq1" means "Print symbol "¤", then move tape right, then if tape symbol is "¤" then go to state q1".

Kleene observes that Post atomized these 3-actions further into two types of 2-actions. The first type is a "print/erase" action, the second is a "move tape left/right action": (1.i) print-symbol/erase/do-nothing followed by (1.ii) test-tape-go-to-next-instruction, OR (2.ii) move-tape-left/move-tape-right/do-nothing followed by (2.ii) test-tape-go-to-next-instruction.

But Kleene observes that while

"Indeed it could be argued that the Turing machine act is already compound, and consists psychologically in a printing and change in state of mind, followed by a motion and another state of mind [, and] Post 1947 does thus separate the Turing act into two; we have not here, primarily because it saves space in the machine tables not to do so."(Kleene p. 379)

In fact Post's treatment (1936) is ambiguous; both (1.1) and (2.1) could be followed by "(.ii) go to next instruction in numerical sequence". This represents a further atomization into three types of instructions: (1) print-symbol/erase/do-nothing then go-to-next-instruction-in-numerical-sequence, (2) move-tape-left/move-tape-right/do-nothing then go-to-next-instruction-in-numerical-sequence (3)test-tape then go-to-instruction-xxx-else-go-to-next-instruction-in-numerical-sequence.

Langton's ant

Langton's ant after 11000 steps. A red pixel shows the ant's location.

Langton's ant is a two-dimensional Turing machine with a very simple set of rules but complicated emergent behavior. It was invented by Chris Langton in 1986 and runs on a square lattice of black and white cells. The universality of Langton's ant was proven in 2000. The idea has been generalized in several different ways, such as turmites which add more colors and more states.

Rules

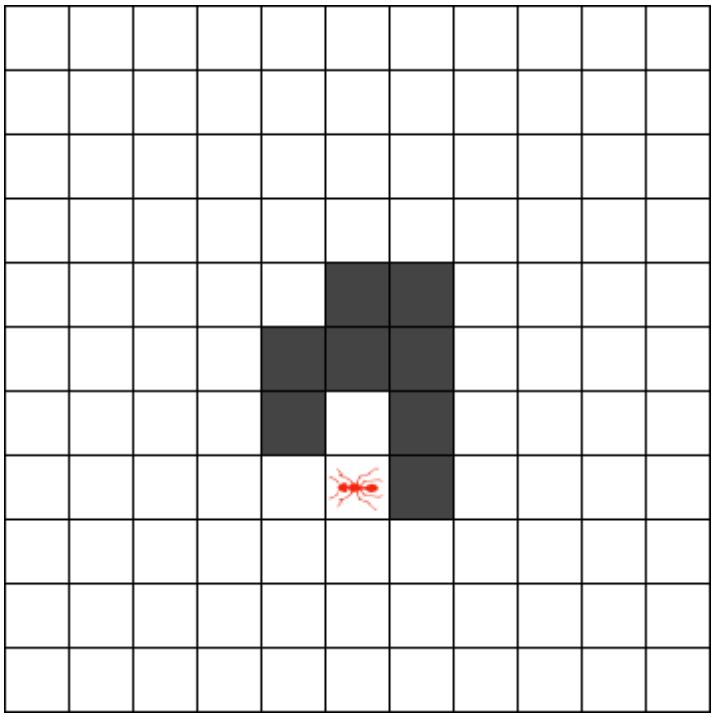


Image of first 200 steps of Langton's ant

Squares on a plane are colored variously either black or white. We arbitrarily identify one square as the "ant". The ant can travel in any of the four cardinal directions at each step it takes. The ant moves according to the rules below:

- At a white square, turn 90° right, flip the color of the square, move forward one unit
- At a black square, turn 90° left, flip the color of the square, move forward one unit

These simple rules lead to surprisingly complex behavior: after an initial period of apparently chaotic behavior, that lasts for about 10,000 steps (in the simplest case), the ant starts building a recurrent "highway" pattern of 104 steps that repeat indefinitely. All finite initial configurations tested eventually converge to the same repetitive pattern suggesting that the "highway" is an attractor of Langton's ant, but no one has been able to prove that this is true for all such initial configurations. It is only known that the ant's trajectory is always unbounded regardless of the initial configuration - this is known as the **Cohen-Kung theorem**.

Langton's ant can also be described as a cellular automaton, where most of the grid is colored black or white, and the "ant" square has one of eight different colors assigned to encode the combination of black/white state and the current direction of motion of the ant.

Universality

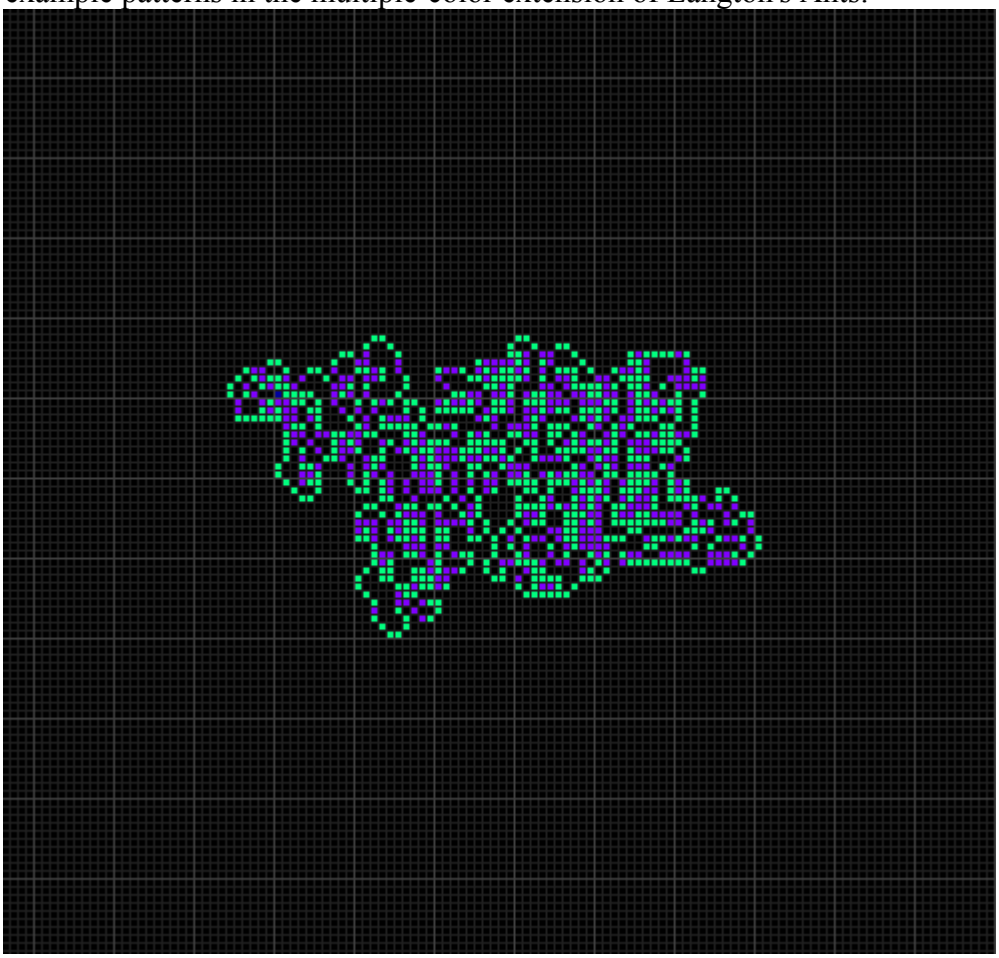
In 2000, Gajardo et al. showed a construction that calculates any boolean circuit using the trajectory of a single instance of Langton's ant. Thus, it would be possible to simulate a Turing machine using the ant's trajectory for computation. This means that the ant is capable of universal computation.

Extension to multiple colors

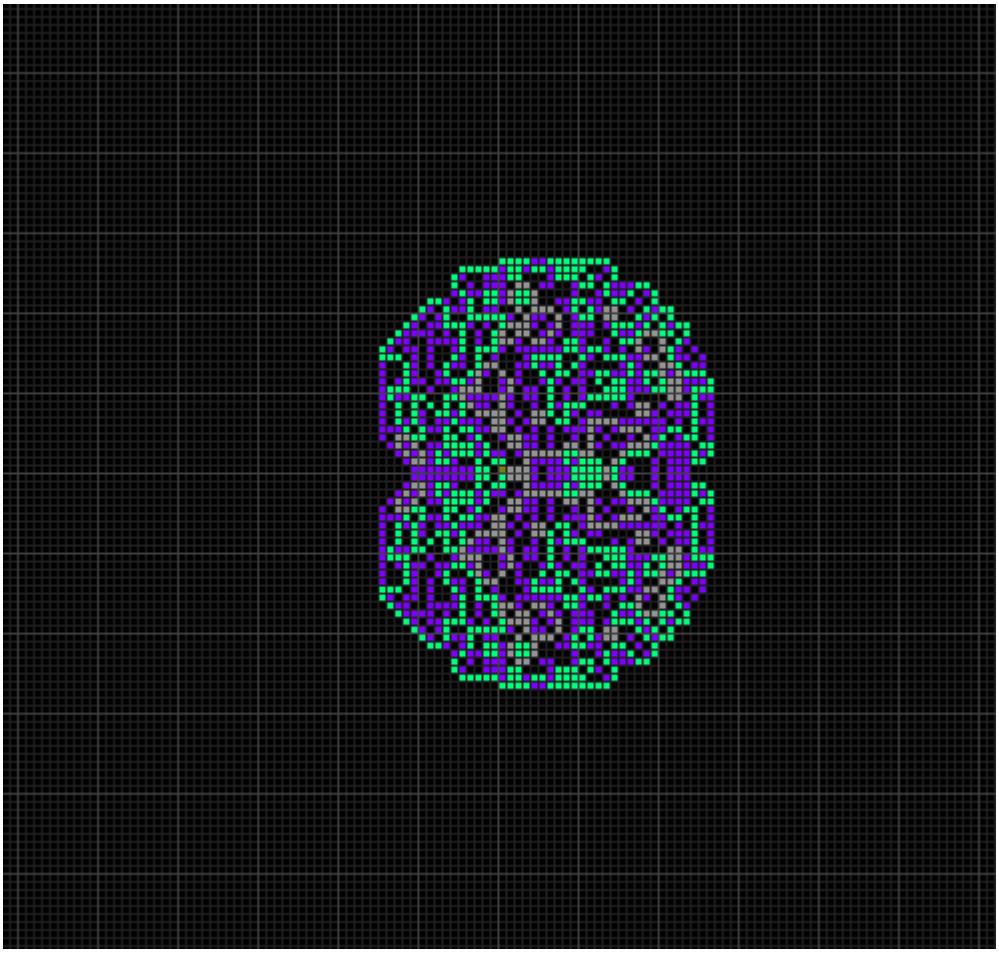
Greg Turk and Jim Propp considered a simple extension to Langton's ant where instead of just two colors, more colors are used. The colors are modified in a cyclic fashion. A simple naming scheme is used: for each of the successive colors, a letter 'L' or 'R' is used to indicate whether a left or right turn should be taken. Langton's ant has the name 'RL' in this naming scheme.

Some of these extended Langton's ants produce patterns that become symmetric over and over again. One of the simplest examples is the ant 'RLLR'. One sufficient condition for this to happen is that the ant's name, seen as a cyclic list, consists of consecutive pairs of identical letters 'LL' or 'RR' (the term "cyclic list" indicates that the last letter may pair with the first one.) The proof involves Truchet tiles.

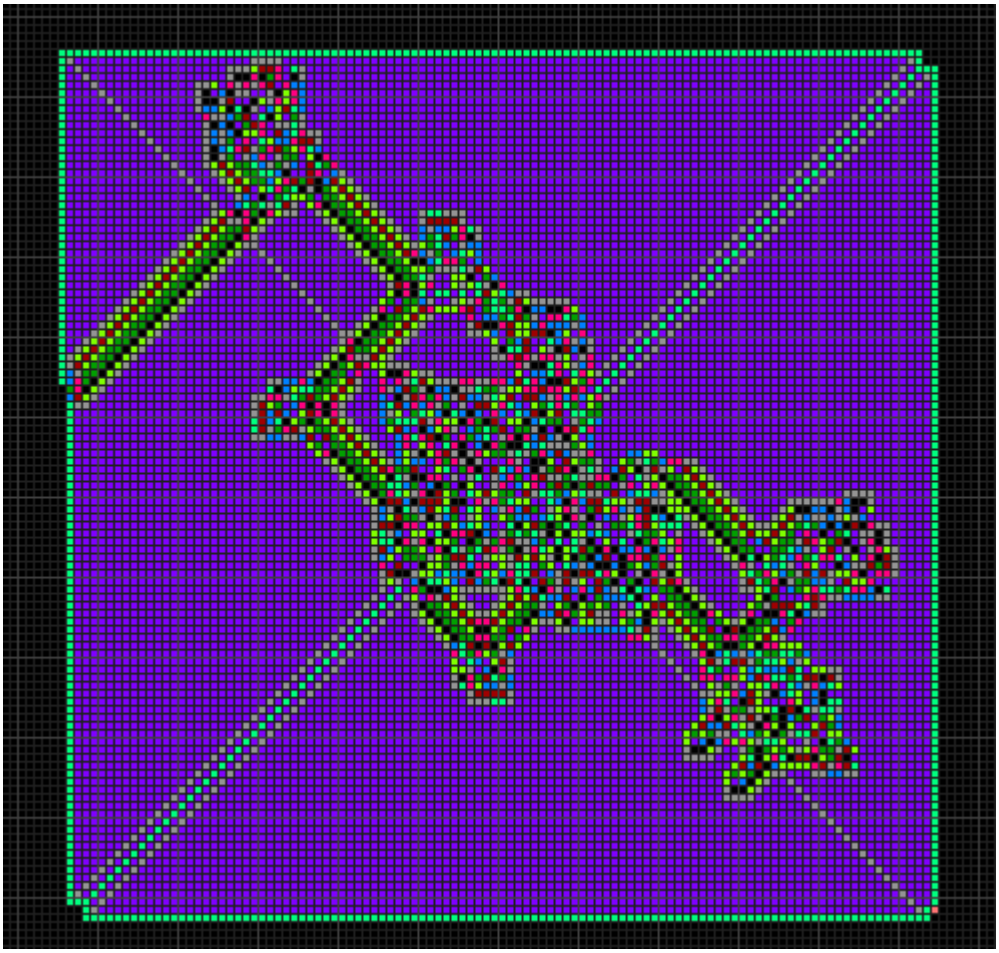
Some example patterns in the multiple-color extension of Langton's Ants:



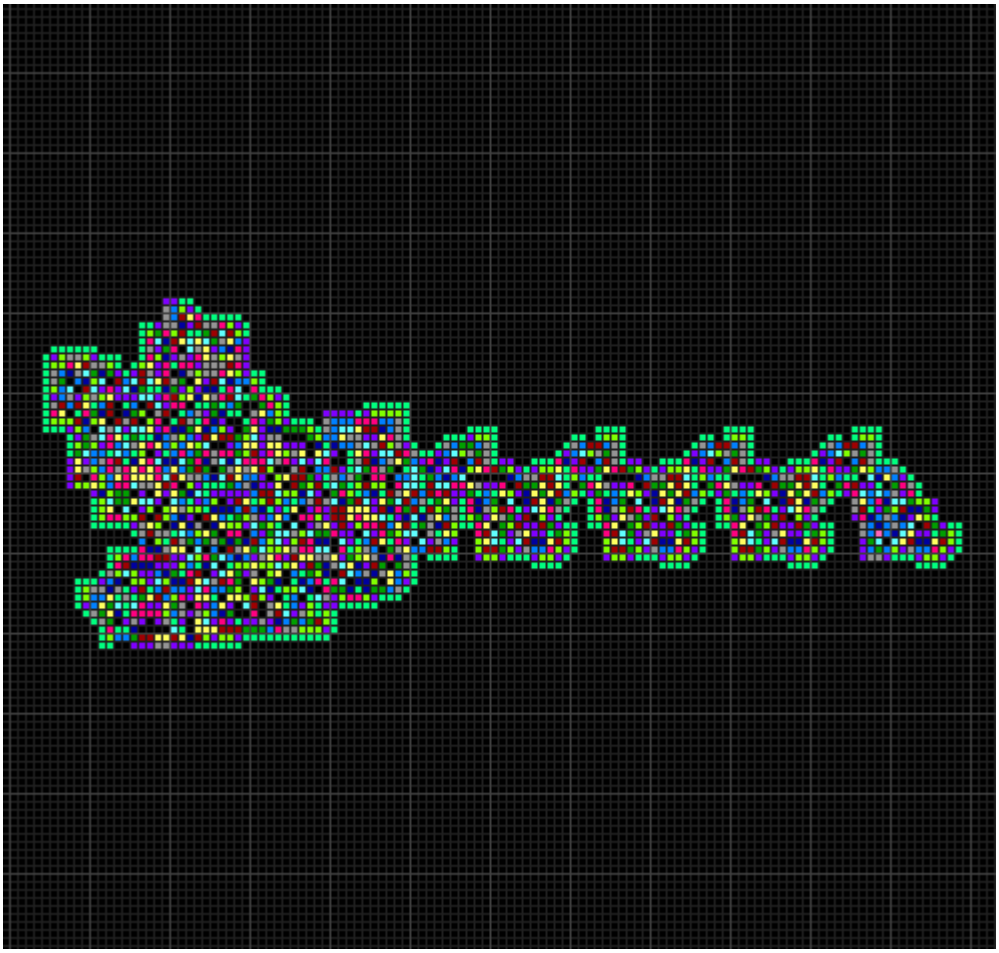
RLR: grows chaotically. It is not known if this ant ever produces a highway.



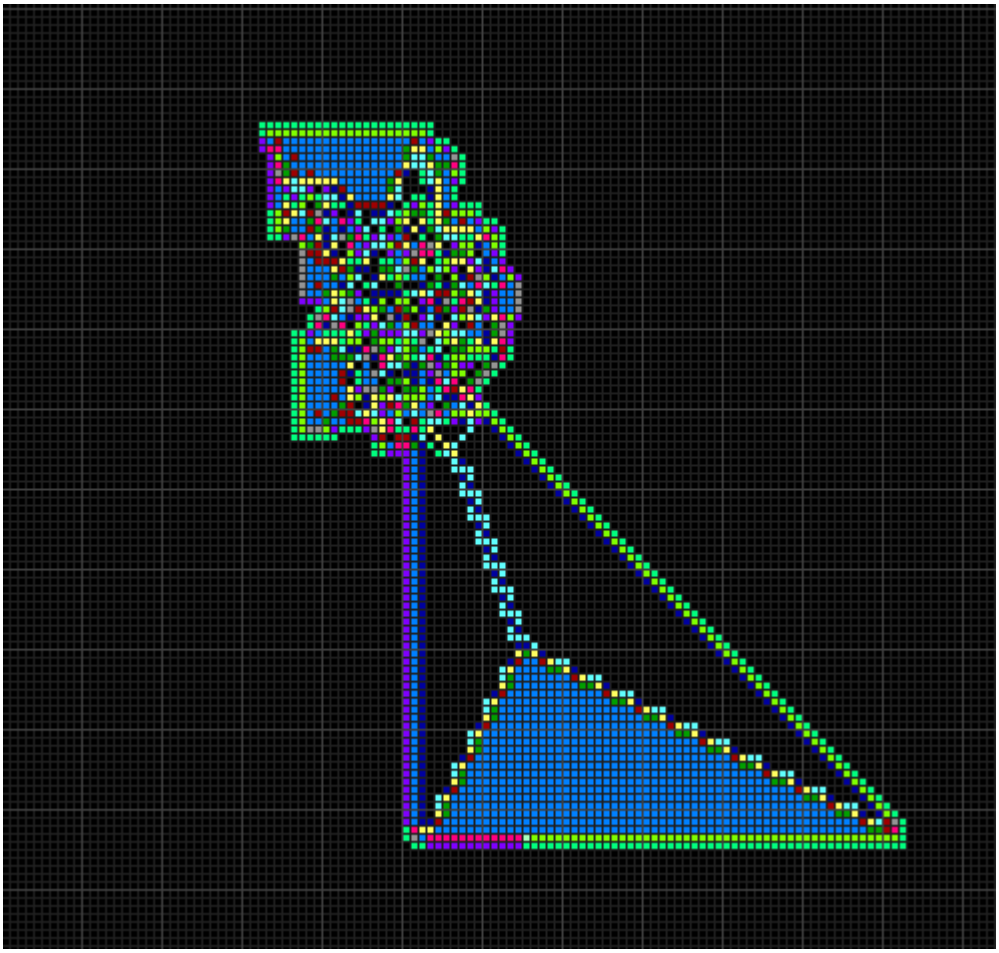
LLRR: grows symmetrically.



LRRRRRLLR: fills space in a square around itself.



LLRRRLRLRLLR: creates a convoluted highway.

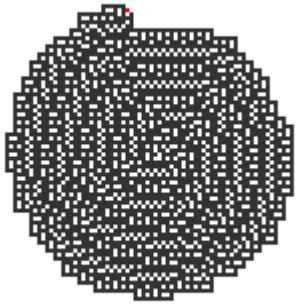


RRLLLRLLLRRR: creates a filled triangle shape that grows and moves.

Extension to multiple states

A further extension of Langton's Ants is to consider multiple states of the Turing machine - as if the ant itself has a color that can change. These ants are called turmites, a contraction of "Turing machine termites". Common behaviours include the production of highways, chaotic growth and spiral growth.

Some example turmites:



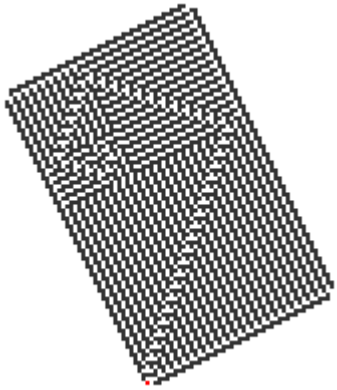
Spiral growth.

Semi-chaotic growth.

Production of a highway after a period of chaotic growth.

Chaotic growth with a distinctive texture.

Growth with a distinctive texture inside an expanding frame.



Constructing a Fibonacci spiral.

Extension to multiple ants

Multiple Langton's ants can co-exist on the 2D plane, as long as there is a rule for what happens when they meet. Ed Pegg, Jr. considered turmites that can turn for example *both* left and right, splitting in two and annihilating each other when they meet.

Chapter 5

Non-Deterministic Turing Machine and Turmite

Non-deterministic Turing machine

In theoretical computer science, a Turing machine is a theoretical machine that is used in thought experiments to examine the abilities and limitations of computers.

In essence, a Turing machine is imagined to be a simple computer that reads and writes symbols one at a time on an endless tape by strictly following a set of rules. It determines what action it should perform next according to its internal "state" and what number it currently sees. An example of one of a Turing Machine's rules might thus be: "If you are in state 2 and you see an 'A', change it to a 'B' and move left."

In a **deterministic Turing machine**, the set of rules prescribes at most one action to be performed for any given situation. A **non-deterministic Turing machine** (**NTM**), by contrast, may have a set of rules that prescribes more than one action for a given situation. For example, a non-deterministic Turing machine may have both "If you are in state 2 and you see an 'A', change it to a 'B' and move left" and "If you are in state 2 and you see an 'A', change it to a 'C' and move right" in its rule set.

An ordinary (deterministic) Turing machine (DTM) has a transition function that, for a given state and symbol under the tape head, specifies three things: the symbol to be written to the tape, the direction (left or right) in which the head should move, and the subsequent state of the finite control. For example, an X on the tape in state 3 might make the DTM write a Y on the tape, move the head one position to the right, and switch to state 5.

A non-deterministic Turing machine (NTM) differs in that the state and tape symbol no longer *uniquely* specify these things; rather, many different actions may apply for the same combination of state and symbol. For example, an X on the tape in state 3 might now allow the NTM to write a Y, move right, and switch to state 5 *or* to write an X, move left, and stay in state 3.

Definition

A nondeterministic Turing machine can be formally defined as a 6-tuple $M = (Q, \Sigma, \iota, \sqcup, A, \delta)$, where

- Q is a finite set of states
- Σ is a finite set of symbols (the tape alphabet)
- $\iota \in Q$ is the initial state
- $\sqcup \in \Sigma$ is the blank symbol
- $A \subseteq Q$ is the set of accepting states
- $\delta \subseteq (Q \backslash A \times \Sigma) \times (Q \times \Sigma \times \{L, R\})$ is a relation on states and symbols called the *transition relation*..

The difference with a standard (deterministic) Turing machine is that for those, the transition relation is a function (the transition function).

Configurations and the *yields* relation on configurations, which describes the possible actions of the Turing machine given any possible contents of the tape, are as for standard Turing machines, except that the yields relation is no longer single-valued. The notion of string acceptance is unchanged: a non-deterministic Turing machine accepts a string if, when the machine is started on the configuration in which the tape head is on the first character of the string (if any), and the tape is all blank otherwise, at least one of the machine's possible computations from that configuration puts the machine into a state in *A*. (If the machine is deterministic, the possible computations are the prefixes of a single, possibly infinite, path.)

Resolution of multiple rules

How does the NTM "know" which of these actions it should take? There are two ways of looking at it. One is to say that the machine is the "luckiest possible guesser"; it always picks the transition which eventually leads to an accepting state, if there is such a transition. The other is to imagine that the machine "branches" into many copies, each of which follows one of the possible transitions. Whereas a DTM has a single "computation path" that it follows, an NTM has a "computation tree". If any branch of the tree halts with an "accept" condition, we say that the NTM accepts the input.

Variations

Equivalence with DTMs

In particular, nondeterministic Turing machines are equivalent with deterministic Turing machines. This equivalency refers to what can be computed, as opposed to how quickly.

NTMs effectively include DTMs as special cases, so it is immediately clear that DTMs are not more powerful. It might seem that NTMs are more powerful than DTMs, since

they can allow trees of possible computations arising from the same initial configuration, accepting a string if any one branch in the tree accepts it.

But it is possible to simulate NTMs with DTMs.

One approach is to use a DTM of which the configurations represent multiple configurations of the NTM, and the DTM's operation consists of visiting each of them in turn, executing a single step at each visit, and spawning new configurations whenever the transition relation defines multiple continuations; this is effectively how a multitasking operating system implements the execution of multiple concurrent processes with a single processor and a single memory array.

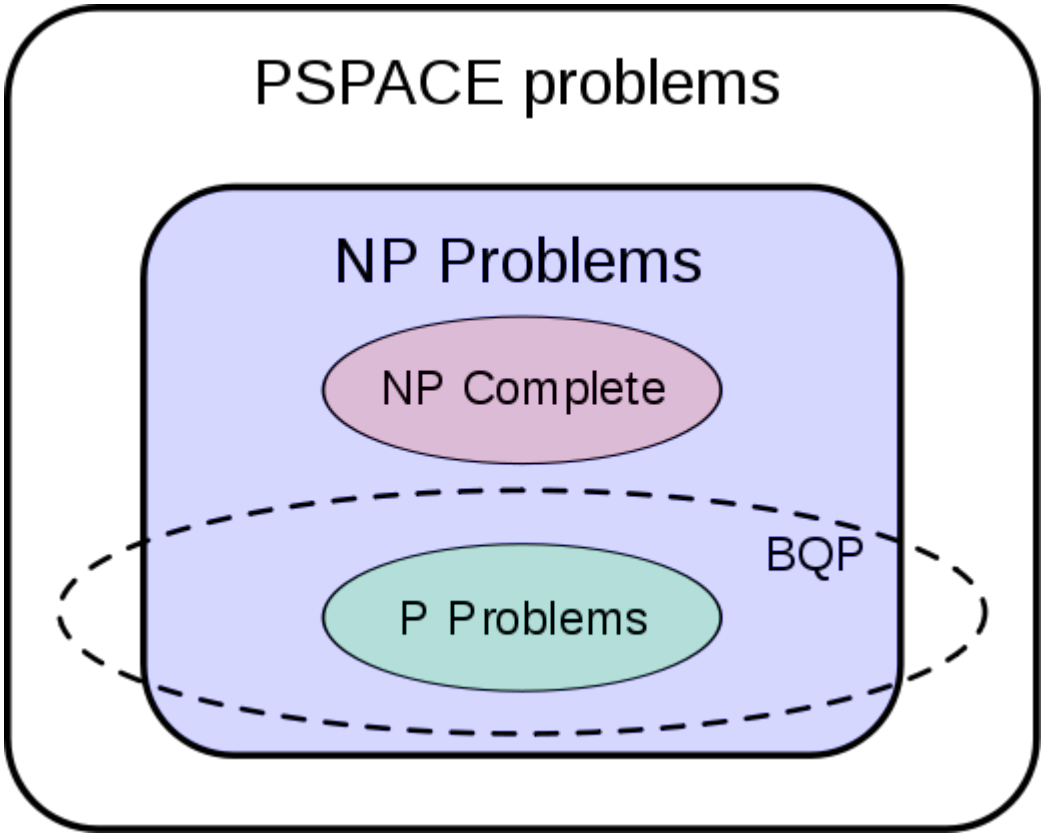
Another construction simulates NTMs with 3-tape DTMs, of which the first tape always holds the original input string, the second is used to simulate a particular computation of the NTM, and the third encodes a path in the NTM's computation tree. The 3-tape DTMs are easily simulated with a normal single-tape DTM.

In this construction, the resulting DTM effectively performs a breadth-first search of the NTM's computation tree, visiting all possible computations of the NTM in order of increasing length until it finds an accepting one. Therefore, the length of an accepting computation of the DTM is, in general, exponential in the length of the shortest accepting computation of the NTM. This is considered to be a general property of simulations of NTMs by DTMs; the most famous unresolved question in computer science, the P = NP problem, is related to this issue.

Bounded non-determinism

An NTM has the property of bounded non-determinism, *i.e.*, if an NTM always halts on a given input tape *T* then it halts in a bounded number of steps, and therefore can only have a bounded number of possible configurations.

Comparison with quantum computers



The suspected shape of the range of problems solvable by quantum computers in polynomial time. Note that the figure suggests $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{N}P \neq \mathbf{PSPACE}$, if this is not true then the figure should look otherwise.

It is a common misconception that quantum computers are NTMs. It is believed but has not been proven that the power of quantum computers is incomparable to that of NTMs. That is, problems likely exist that an NTM could efficiently solve but that a quantum computer cannot. A likely example of problems solvable by NTMs but not by quantum computers in polynomial time are NP-complete problems.

Turmite

A 2-state 2-color turmite on a square grid. Starting from an empty grid, after 8342 steps the turmite (a red pixel) has exhibited both chaotic and regular movement phases.

In computer science, a **turmite** is a Turing machine which has an orientation as well as a current state and a "tape" that consists of an infinite two-dimensional grid of cells. The terms **ant** and **vant** are also used. Langton's ant is a well-known type of turmite defined on the cells of a square grid. Paterson's worms are a type of turmite defined on the edges of an isometric grid.

It has been shown that turmites in general are exactly equivalent in power to one-dimensional Turing machines with an infinite tape, as either can simulate the other.

History

Langton's ants were invented in 1986 and declared "equivalent to Turing machines". Independently, in 1988, Allen H. Brady considered the idea of two-dimensional Turing machines with an orientation and called them "TurNing machines".

Apparently independently of both of these, Greg Turk investigated the same kind of system and wrote to A. K. Dewdney about them. A. K. Dewdney named them "tur-mites" in his "Computer Recreations" column in Scientific American in 1989. Rudy Rucker relates the story as follows:

Dewdney reports that, casting about for a name for Turk's creatures, he thought, "Well, they're Turing machines studied by Turk, so they should be tur-something. And they're like little insects, or mites, so I'll call them tur-mites! And that sounds like termites!" With the kind permission of Turk and Dewdney, I'm going to leave out the hyphen, and call them turmites.
—Rudy Rucker, *Artificial Life Lab*

Relative vs. Absolute Turmites

Turmites can be categorised as being either *relative* or *absolute*. Relative turmites, alternatively known as 'Turning machines', have an internal orientation. Langton's Ant is such an example. Relative turmites are, by definition, isotropic; rotating the turmite does not affect its outcome. Relative turmites are so named because the directions are encoded *relative* to the current orientation, equivalent to using the words 'left' or 'backwards'. Absolute turmites, by comparison, encode their directions in absolute terms: a particular instruction may direct the turmite to move 'North'. Absolute turmites are two-dimensional analogues of conventional Turing machines, so are occasionally referred to as simply "Two-dimensional Turing machines".

Specification

The following specification is specific to turmites on a two-dimensional square grid, the most studied type of turmite. Turmites on other grids can be specified in a similar fashion.

As with Langton's ant, turmites perform the following operations each timestep:

1. turn on the spot (by some multiple of 90°)
2. change the color of the square
3. move forward one square

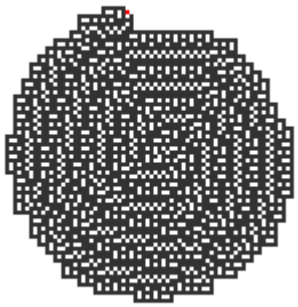
As with Turing machines, the actions are specified by a state transition table listing the current internal state of the turmite and the color of the cell it is currently standing on. For example, the turmite shown in the image at the top of this page is specified by the following table:

		Current color					
		0			**1**		
		Write color	**Turn**	**Next state**	**Write color**	**Turn**	**Next state**
Current state	**0**	1	R	0	1	R	1
	1	0	N	0	0	N	1

The direction to turn is one of **L** (90° left), **R** (90° right), **N** (no turn) and **U** (180° U-turn).

Examples

Some example 2-state 2-color turmites on a square grid, all starting from an empty configuration:

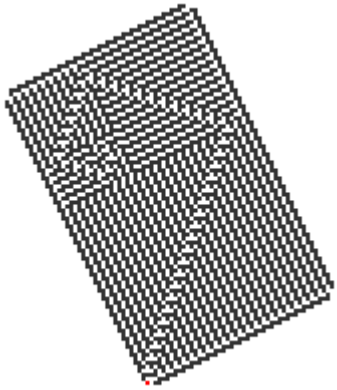


Spiral growth.

Production of a highway after a period of chaotic growth.

Chaotic growth with a distinctive texture.

Growth with a distinctive texture inside an expanding frame.



Constructing a Fibonacci spiral.

Starting from an empty grid or other configurations, the most commonly observed behaviours are chaotic growth, spiral growth and 'highway' construction. Rare examples become periodic after a certain number of steps.

Turmites and the Busy Beaver game

Allen H. Brady searched for terminating turmites (the equivalent of busy beavers) and found a 2-state 2-color machine that printed 37 1's before halting, and another that took 121 steps before halting. He also considered turmites that move on a triangular grid, finding several busy beavers here too.

Ed Pegg, Jr. considered another approach to the busy beaver game. He suggested turmites that can turn for example *both* left and right, splitting in two. Turmites that later meet annihilate each other. In this system, a Busy Beaver is one that from a starting pattern of a single turmite lasts the longest before all the turmites annihilate each other.

Other grids

Following Allen H. Brady's initial work of turmites on a triangular grid, hexagonal tilings have also been explored. Much of this work is due to Tim Hutton, and his results are on the Rule Table Repository. He has also considered Turmites in three dimensions, and collected some preliminary results. Allen H. Brady and Tim Hutton have also investigated one-dimensional relative turmites on the integer lattice, which Brady termed *flippers*. (One-dimensional *absolute* turmites are of course simply known as Turing machines.)

Chapter 6

System Image and System Prevalence

System image

A **system image** in computing is a copy of the entire state of a computer system stored in some non-volatile form such as a file. A system is said to be capable of using system images if it can be shut down and later restored to exactly the same state. In such cases, system images can be used for backup.

Notebook hibernation is an example that uses an image of the entire machine's RAM.

Disk images

If a system has all its state written to a disk, then a system image can be produced by simply copying that disk to a file elsewhere, often with disk cloning applications. On many systems a complete system image cannot be created by a disk cloning program running within that system because information can be held outside of disks and volatile memory, for example in non-volatile memory like boot ROMs.

Process images

A process image is a copy of its state at a given point in time. It is often used for persistence. A common example is a database management system (DBMS). Most DBMS can store the state of its database or databases to a file before being closed down. The DBMS can then be restarted later with the information in the database intact and proceed as though the software had never stopped. Another example would be the hibernate feature of many operating systems. Here the state of all RAM memory is stored to disk, the computer is brought into an energy saving mode, then later restored to normal operation.

Some emulators provide a facility to save an image of the system being emulated. This is often called a savestate.

Programming language support

Some programming languages provide a command to take a system image of a program. This is normally a standard feature in Smalltalk (inspired by FLEX) and Lisp, among other languages. Development in these languages is often quite different from many other programming languages. For example in Lisp the programmer may load packages or other code into a running Lisp implementation using the read-eval-print loop, which usually compiles the programs. Data is loaded into the running Lisp system. The programmer may then dump a system image, containing that pre-compiled and possibly customized code - and also all loaded application data. Often this image is an executable, and can be run on other machines. This system image can be the form in which executable programs are distributed — this method has often been used by programs (such as TeX and Emacs) largely implemented in Lisp, Smalltalk, or idiosyncratic languages to avoid spending time repeating the same initialization work every time they start up.

Similar, Lisp Machines were booted from Lisp images, called Worlds. The World contains the complete operating system, its applications and its data in a single file. It was also possible to save incremental Worlds, that contain only the changes from some base World. Before saving the World, the Lisp Machine operating system could optimize the contents of memory (better memory layout, compacting data structures, sorting data, ...).

Java provides an object serialization mechanism that can be used to conveniently produce system images for object-oriented systems that have their object graph accessible through a single root object.

Although its purpose is different, a "system image" is often similar in structure to a core dump.

System Prevalence

System prevalence is a simple transparent persistence technique for computer system state storage and retrieval. It combines system images and transaction journaling to overcome weaknesses of both techniques.

In a prevalent system, state is kept in memory in native format, rather than being written to an RDBMS or other data storage system. System images are regularly saved to disk, and in addition to this, all transactions are journaled.

System images and transaction journals can be stored in language-specific serialization format for speed or in XML format for cross-language portability.

The first usage of the term and generic, publicly-available implementation of a system prevalence layer was Prevayler, written for Java by Klaus Wuestefeld in 2001.

Advantages

Simply keeping system state in memory in its normal, natural, language-specific format is orders of magnitude faster and more programmer-friendly than the multiple conversions that are needed when it is stored and retrieved from a DBMS.

Requirement

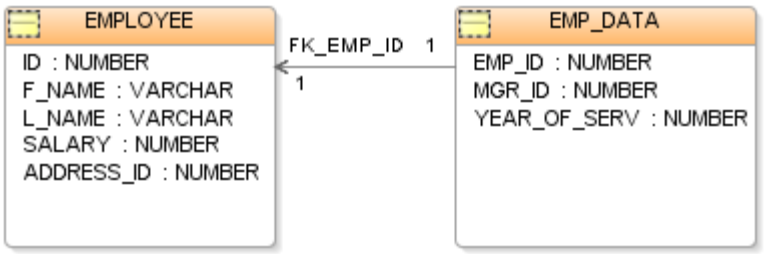
A prevalent system needs enough memory to hold its entire state in RAM (the "prevalent hypothesis"). Prevalence advocates claim this is continuously alleviated by decreasing RAM prices, and the fact that many business databases are small enough already to fit in memory. Programmers need skill in working with business state natively in RAM, rather than using explicit API calls for storage and queries for retrieval.

Implementations

- Prevayler (Java)
- Space4J (Java)
- Bamboo.Prevalence (.NET)
- Pypersyst (Python)
- Madeleine (Ruby)
- cl-prevalence (Common Lisp)
- Perlvayler (Perl)
- SPrevayler (Smalltalk)
- EntityORM (.NET)

Chapter 7

Database Management System



A **Database Management System** (**DBMS**) is a set of computer programs that controls the creation, maintenance, and the use of a database. It allows organizations to place control of database development in the hands of database administrators (DBAs) and other specialists. A DBMS is a system software package that helps the use of integrated collection of data records and files known as databases. It allows different user application programs to easily access the same database. DBMSs may use any of a variety of database models, such as the network model or relational model. In large systems, a DBMS allows users and other software to store and retrieve data in a structured way. Instead of having to write computer programs to extract information, user can ask simple questions in a query language. Thus, many DBMS packages provide Fourth-generation programming language (4GLs) and other application development features. It helps to specify the logical organization for a database and access and use the information within a database. It provides facilities for controlling data access, enforcing data integrity, managing concurrency, and restoring the database from backups. A DBMS also provides the ability to logically present database information to users.

Overview

A DBMS is a set of software programs that controls the organization, storage, management, and retrieval of data in a database. DBMSs are categorized according to their data structures or types. The DBMS accepts requests for data from an application program and instructs the operating system to transfer the appropriate data. The queries and responses must be submitted and received according to a format that conforms to one or more applicable protocols. When a DBMS is used, information systems can be changed more easily as the organization's information requirements change. New categories of data can be added to the database without disruption to the existing system.

Database servers are dedicated computers that hold the actual databases and run only the DBMS and related software. Database servers are usually multiprocessor computers, with generous memory and RAID disk arrays used for stable storage. Hardware database accelerators, connected to one or more servers via a high-speed channel, are also used in large volume transaction processing environments. DBMSs are found at the heart of most database applications. DBMSs may be built around a custom multitasking kernel with built-in networking support, but modern DBMSs typically rely on a standard operating system to provide these functions.

History

Databases have been in use since the earliest days of electronic computing. Unlike modern systems which can be applied to widely different databases and needs, the vast majority of older systems were tightly linked to the custom databases in order to gain speed at the expense of flexibility. Originally DBMSs were found only in large organizations with the computer hardware needed to support large data sets.

1960s Navigational DBMS

As computers grew in speed and capability, a number of general-purpose database systems emerged; by the mid-1960s there were a number of such systems in commercial use. Interest in a standard began to grow, and Charles Bachman, author of one such product, the Integrated Data Store (IDS), founded the "Database Task Group" within CODASYL, the group responsible for the creation and standardization of COBOL. In 1971 they delivered their standard, which generally became known as the "Codasyl approach", and soon a number of commercial products based on this approach were made available.

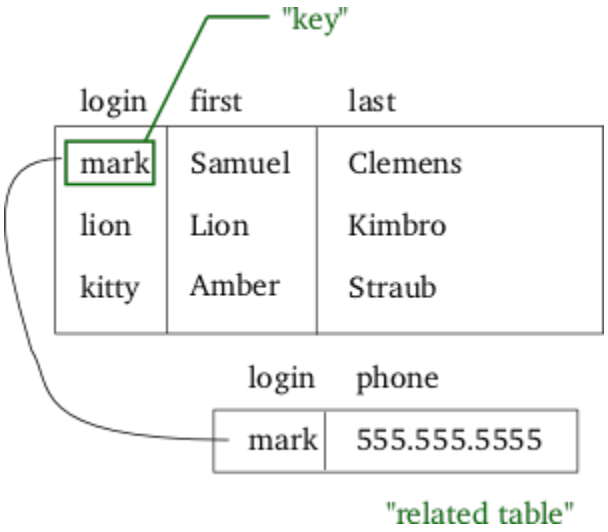
The Codasyl approach was based on the "manual" navigation of a linked data set which was formed into a large network. When the database was first opened, the program was handed back a link to the first record in the database, which also contained pointers to other pieces of data. To find any particular record the programmer had to step through these pointers one at a time until the required record was returned. Simple queries like "find all the people in India" required the program to walk the entire data set and collect the matching results one by one. There was, essentially, no concept of "find" or "search". This may sound like a serious limitation today, but in an era when most data was stored on magnetic tape such operations were too expensive to contemplate anyway.

IBM also had their own DBMS system in 1968, known as *IMS*. IMS was a development of software written for the Apollo program on the System/360. IMS was generally similar in concept to Codasyl, but used a strict hierarchy for its model of data navigation instead of Codasyl's network model. Both concepts later became known as navigational databases due to the way data was accessed, and Bachman's 1973 Turing Award award presentation was *The Programmer as Navigator*. IMS is classified as a hierarchical database. IMS and IDMS, both CODASYL databases, as well as CINCOMs TOTAL database, are classified as network databases.

1970s Relational DBMS

Edgar Codd worked at IBM in San Jose, California, in one of their offshoot offices that was primarily involved in the development of hard disk systems. He was unhappy with the navigational model of the Codasyl approach, notably the lack of a "search" facility. In 1970, he wrote a number of papers that outlined a new approach to database construction that eventually culminated in the groundbreaking *A Relational Model of Data for Large Shared Data Banks*.

In this paper, he described a new system for storing and working with large databases. Instead of records being stored in some sort of linked list of free-form records as in Codasyl, Codd's idea was to use a "table" of fixed-length records. A linked-list system would be very inefficient when storing "sparse" databases where some of the data for any one record could be left empty. The relational model solved this by splitting the data into a series of normalized tables, with optional elements being moved out of the main table to where they would take up room only if needed.



In the relational model, related records are linked together with a "key".

For instance, a common use of a database system is to track information about users, their name, login information, various addresses and phone numbers. In the navigational approach all of these data would be placed in a single record, and unused items would simply not be placed in the database. In the relational approach, the data would be *normalized* into a user table, an address table and a phone number table (for instance). Records would be created in these optional tables only if the address or phone numbers were actually provided.

Linking the information back together is the key to this system. In the relational model, some bit of information was used as a "key", uniquely defining a particular record. When information was being collected about a user, information stored in the optional (or *related*) tables would be found by searching for this key. For instance, if the login name

of a user is unique, addresses and phone numbers for that user would be recorded with the login name as its key. This "re-linking" of related data back into a single collection is something that traditional computer languages are not designed for.

Just as the navigational approach would require programs to loop in order to collect records, the relational approach would require loops to collect information about any one record. Codd's solution to the necessary looping was a set-oriented language, a suggestion that would later spawn the ubiquitous SQL. Using a branch of mathematics known as tuple calculus, he demonstrated that such a system could support all the operations of normal databases (inserting, updating etc.) as well as providing a simple system for finding and returning *sets* of data in a single operation.

Codd's paper was picked up by two people at Berkeley, Eugene Wong and Michael Stonebraker. They started a project known as INGRES using funding that had already been allocated for a geographical database project, using student programmers to produce code. Beginning in 1973, INGRES delivered its first test products which were generally ready for widespread use in 1979. During this time, a number of people had moved "through" the group — perhaps as many as 30 people worked on the project, about five at a time. INGRES was similar to System R in a number of ways, including the use of a "language" for data access, known as QUEL — QUEL was in fact relational, having been based on Codd's own Alpha language, but has since been corrupted to follow SQL, thus violating much the same concepts of the relational model as SQL itself.

IBM itself did one test implementation of the relational model, PRTV, and a production one, Business System 12, both now discontinued. Honeywell did MRDS for Multics, and now there are two new implementations: Alphora Dataphor and Rel. All other DBMS implementations usually called *relational* are actually SQL DBMSs. In 1968, the University of Michigan began development of the Micro DBMS . It was used to manage very large data sets by the US Department of Labor, the Environmental Protection Agency and researchers from University of Alberta, the University of Michigan and Wayne State University. It ran on mainframe computers using Michigan Terminal System. The system remained in production until 1996.

End 1970s SQL DBMS

IBM started working on a prototype system loosely based on Codd's concepts as *System R* in the early 1970s. The first version was ready in 1974/5, and work then started on multi-table systems in which the data could be split so that all of the data for a record (some of which is optional) did not have to be stored in a single large "chunk". Subsequent multi-user versions were tested by customers in 1978 and 1979, by which time a standardized query language - SQL - had been added. Codd's ideas were establishing themselves as both workable and superior to Codasyl, pushing IBM to develop a true production version of System R, known as *SQL/DS*, and, later, *Database 2* (DB2).

Many of the people involved with INGRES became convinced of the future commercial success of such systems, and formed their own companies to commercialize the work but with an SQL interface. Sybase, Informix, NonStop SQL and eventually Ingres itself were all being sold as offshoots to the original INGRES product in the 1980s. Even Microsoft SQL Server is actually a re-built version of Sybase, and thus, INGRES. Only Larry Ellison's Oracle started from a different chain, based on IBM's papers on System R, and beat IBM to market when the first version was released in 1978.

Stonebraker went on to apply the lessons from INGRES to develop a new database, Postgres, which is now known as PostgreSQL. PostgreSQL is often used for global mission critical applications (the .org and .info domain name registries use it as their primary data store, as do many large companies and financial institutions).

In Sweden, Codd's paper was also read and Mimer SQL was developed from the mid-70s at Uppsala University. In 1984, this project was consolidated into an independent enterprise. In the early 1980s, Mimer in c introduced transaction handling for high robustness in applications, an idea that was subsequently implemented on most other DBMS.

1980s Object Oriented Databases

The 1980s, along with a rise in object oriented programming, saw a growth in how data in various databases were handled. Programmers and designers began to treat the data in their databases as objects. That is to say that if a person's data were in a database, that person's attributes, such as their address, phone number, and age, were now considered to belong to that person instead of being extraneous data. This allows for relationships between data to be relation to objects and their attributes and not to individual fields.

Another big game changer for databases in the 1980s was the focus on increasing reliability and access speeds. In 1989, two professors from the University of Michigan at Madison published an article at an ACM associated conference outlining their methods on increasing database performance. The idea was to replicate specific important, and often queried information, and store it in a smaller temporary database that linked these key features back to the main database. This meant that a query could search the smaller database much quicker, rather than search the entire dataset. This eventually leads to the practice of indexing, which is used by almost every operating system from Windows to the system that operates Apple iPod devices.

Current trends

In 1998, database management was in need of a new style of databases to solve current database management problems. Researchers realized that the old trends of database management were becoming too complex and there was a need for automated configuration and management. Surajit Chaudhuri, Gerhard Weikum and Michael Stonebraker were the pioneers that dramatically affected the thought of database management systems. They believed that database management needed a more modular

approach and there were too many specifications needed for users. Since this new development process of database management there are more possibilities. Database management is no longer limited to “monolithic entities”. Many solutions have been developed to satisfy the individual needs of users. The development of numerous database options has created flexibility in database management.

There are several ways database management has affected the field of technology. Because organizations' demand for directory services has grown as they expand in size, businesses use directory services that provide prompted searches for company information. Mobile devices are able to store more than just the contact information of users, and can cache and display a large amount of information on smaller displays. Search engine queries are able to locate data within the World Wide Web. Retailers have also benefited from the developments with data warehousing, recording customer transactions. Online transactions have become tremendously popular for e-business. Consumers and businesses are able to make payments securely through some company websites. These current developments would not have been possible without the evolution of database management. Even with the progress of database management, there is a demonstrated need for new development as specifications and needs change.

As the speeds of consumer internet connectivity increase, and as data availability and computing become more ubiquitous, databases are seeing a migration to web services. Web-based languages such as XML and PHP are used to process databases. These languages allow databases to live in "the cloud." As with products such as Google's Gmail, Microsoft's Office 2010, and Carbonite's online backup services, many services are beginning to move to web based services due to increasing internet reliability, data storage efficiency, and the lack of a need for dedicated IT staff to manage the hardware. Faculty at Rochester Institute of Technology published a paper regarding the use of databases in the cloud and state that their university plans to add cloud-based database computing to their curriculum to "keep [their] information technology (IT) curriculum at the forefront of technology."

Building blocks

Components

- **DBMS Engine** accepts logical requests from various other DBMS subsystems, converts them into physical equivalents, and actually accesses the database and data dictionary as they exist on a storage device.
- **Data Definition Subsystem** helps the user create and maintain the data dictionary and define the structure of the files in a database.
- **Data Manipulation Subsystem** helps the user to add, change, and delete information in a database and query it for valuable information. Software tools within the data manipulation subsystem are most often the primary interface between user and the information contained in a database. It allows the user to specify its logical information requirements.

- **Application Generation Subsystem** contains facilities to help users develop transaction-intensive applications. It usually requires that the user perform a detailed series of tasks to process a transaction. It facilitates easy-to-use data entry screens, programming languages, and interfaces.
- **Data Administration Subsystem** helps users manage the overall database environment by providing facilities for backup and recovery, security management, query optimization, concurrency control, and change management.

Modeling language

A modeling language is a data modeling language to define the schema of each database hosted in the DBMS, according to the DBMS database model. Database management systems (DBMS) are designed to use one of five database structures to provide simplistic access to information stored in databases. The five database structures are:

- the hierarchical model,
- the network model,
- the relational model,
- the multidimensional model, and
- the object model.

Inverted lists and other methods are also used. A given database management system may provide one or more of the five models. The optimal structure depends on the natural organization of the application's data, and on the application's requirements, which include transaction rate (speed), reliability, maintainability, scalability, and cost.

The **hierarchical structure** was used in early mainframe DBMS. Records' relationships form a treelike model. This structure is simple but nonflexible because the relationship is confined to a one-to-many relationship. IBM's IMS system and the RDM Mobile are examples of a hierarchical database system with multiple hierarchies over the same data. RDM Mobile is a newly designed embedded database for a mobile computer system. The hierarchical structure is used primarily today for storing geographic information and file systems.

The **network structure** consists of more complex relationships. Unlike the hierarchical structure, it can relate to many records and accesses them by following one of several paths. In other words, this structure allows for many-to-many relationships.

The **relational structure** is the most commonly used today. It is used by mainframe, midrange and microcomputer systems. It uses two-dimensional rows and columns to store data. The tables of records can be connected by common key values. While working for IBM, E.F. Codd designed this structure in 1970. The model is not easy for the end user to run queries with because it may require a complex combination of many tables.

The **multidimensional structure** is similar to the relational model. The dimensions of the cube-like model have data relating to elements in each cell. This structure gives a

spreadsheet-like view of data. This structure is easy to maintain because records are stored as fundamental attributes - in the same way they are viewed - and the structure is easy to understand. Its high performance has made it the most popular database structure when it comes to enabling online analytical processing (OLAP).

The **object oriented structure** has the ability to handle graphics, pictures, voice and text, types of data, without difficultly unlike the other database structures. This structure is popular for multimedia Web-based applications. It was designed to work with object-oriented programming languages such as Java.

The dominant model in use today is the ad hoc one embedded in SQL, despite the objections of purists who believe this model is a corruption of the relational model since it violates several fundamental principles for the sake of practicality and performance. Many DBMSs also support the Open Database Connectivity API that supports a standard way for programmers to access the DBMS.

Before the database management approach, organizations relied on file processing systems to organize, store, and process data files. End users criticized file processing because the data is stored in many different files and each organized in a different way. Each file was specialized to be used with a specific application. File processing was bulky, costly and nonflexible when it came to supplying needed data accurately and promptly. Data redundancy is an issue with the file processing system because the independent data files produce duplicate data so when updates were needed each separate file would need to be updated. Another issue is the lack of data integration. The data is dependent on other data to organize and store it. Lastly, there was not any consistency or standardization of the data in a file processing system which makes maintenance difficult. For these reasons, the database management approach was produced.

Data structure

Data structures (fields, records, files and objects) optimized to deal with very large amounts of data stored on a permanent data storage device (which implies relatively slow access compared to volatile main memory).

Database query language

A database query language and report object allows users to interactively interrogate the database, analyze its data and update it according to the users privileges on data. It also controls the security of the database. Data security prevents unauthorized users from viewing or updating the database. Using passwords, users are allowed access to the entire database or subsets of it called *subschemas*. For example, an employee database can contain all the data about an individual employee, but one group of users may be authorized to view only payroll data, while others are allowed access to only work history and medical data.

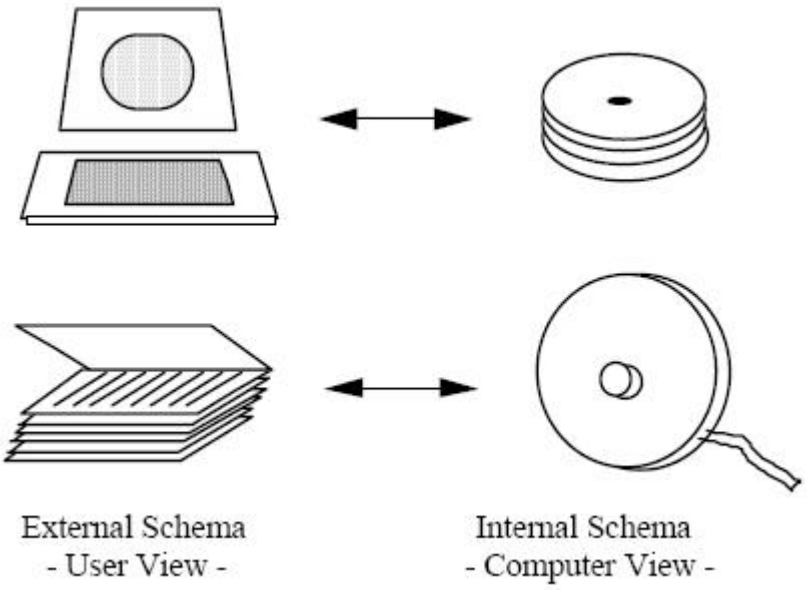
If the DBMS provides a way to interactively enter and update the database, as well as interrogate it, this capability allows for managing personal databases. However, it may not leave an audit trail of actions or provide the kinds of controls necessary in a multi-user organization. These controls are only available when a set of application programs are customized for each data entry and updating function.

Transaction mechanism

A database transaction mechanism ideally guarantees ACID properties in order to ensure data integrity despite concurrent user accesses (concurrency control), and faults (fault tolerance). It also maintains the integrity of the data in the database. The DBMS can maintain the integrity of the database by not allowing more than one user to update the same record at the same time. The DBMS can help prevent duplicate records via unique index constraints; for example, no two customers with the same customer numbers (key fields) can be entered into the database.

Topics

External, Logical and Internal view



Traditional View of Data

A database management system provides the ability for many different users to share data and process resources. As there can be many different users, there are many different database needs. The question is: How can a single, unified database meet varying requirements of so many users?

A DBMS minimizes these problems by providing three views of the database data: an external view (or user view), logical view (or conceptual view) and physical (or internal)

view. The user's view of a database program represents data in a format that is meaningful to a user and to the software programs that process those data.

One strength of a DBMS is that while there is typically only one conceptual (or logical) and physical (or internal) view of the data, there can be an endless number of different external views. This feature allows users to see database information in a more business-related way rather than from a technical, processing viewpoint. Thus the logical view refers to the way the user views the data, and the physical view refers to the way the data are physically stored and processed.

Features and capabilities

Alternatively, and especially in connection with the relational model of database management, the relation between attributes drawn from a specified set of domains can be seen as being primary. For instance, the database might indicate that a car that was originally "red" might fade to "pink" in time, provided it was of some particular "make" with an inferior paint job. Such higher arity relationships provide information on all of the underlying domains at the same time, with none of them being privileged above the others.

Simple definition

A database management system is the system in which related data is stored in an efficient and compact manner. "Efficient" means that the data which is stored in the DBMS can be accessed quickly and "compact" means that the data takes up very little space in the computer's memory. The phrase "related data" is means that the data stored pertains to a particular topic.

Specialized databases have existed for scientific, imaging, document storage and like uses. Functionality drawn from such applications has begun appearing in mainstream DBMS's as well. However, the main focus, at least when aimed at the commercial data processing market, is still on descriptive attributes on repetitive record structures.

Thus, the DBMSs of today roll together frequently needed services or features of attribute management. By externalizing such functionality to the DBMS, applications effectively share code with each other and are relieved of much internal complexity. Features commonly offered by database management systems include:

Query ability
: Querying is the process of requesting attribute information from various perspectives and combinations of factors. Example: "How many 2-door cars in Texas are green?" A database query language and report writer allow users to interactively interrogate the database, analyze its data and update it according to the users privileges on data.

Backup and replication

Copies of attributes need to be made regularly in case primary disks or other equipment fails. A periodic copy of attributes may also be created for a distant organization that cannot readily access the original. DBMS usually provide utilities to facilitate the process of extracting and disseminating attribute sets. When data is replicated between database servers, so that the information remains consistent throughout the database system and users cannot tell or even know which server in the DBMS they are using, the system is said to exhibit replication transparency.

Rule enforcement
: Often one wants to apply rules to attributes so that the attributes are clean and reliable. For example, we may have a rule that says each car can have only one engine associated with it (identified by Engine Number). If somebody tries to associate a second engine with a given car, we want the DBMS to deny such a request and display an error message. However, with changes in the model specification such as, in this example, hybrid gas-electric cars, rules may need to change. Ideally such rules should be able to be added and removed as needed without significant data layout redesign.

Security
: For security reasons, it is desirable to limit who can see or change specific attributes or groups of attributes. This may be managed directly on an individual basis, or by the assignment of individuals and privileges to groups, or (in the most elaborate models) through the assignment of individuals and groups to roles which are then granted entitlements.

Computation
: Common computations requested on attributes are counting, summing, averaging, sorting, grouping, cross-referencing, and so on. Rather than have each computer application implement these from scratch, they can rely on the DBMS to supply such calculations.

Change and access logging
: This describes who accessed which attributes, what was changed, and when it was changed. Logging services allow this by keeping a record of access occurrences and changes.

Automated optimization
: For frequently occurring usage patterns or requests, some DBMS can adjust themselves to improve the speed of those interactions. In some cases the DBMS will merely provide tools to monitor performance, allowing a human expert to make the necessary adjustments after reviewing the statistics collected.

Meta-data repository

Metadata is data describing data. For example, a listing that describes what attributes are allowed to be in data sets is called "meta-information".

Advanced DBMS

An example of an advanced DBMS is Distributed Data Base Management System (DDBMS), a collection of data which logically belong to the same system but are spread out over the sites of the computer network. The two aspects of a distributed database are distribution and logical correlation:

- Distribution: The fact that the data are not resident at the same site, so that we can distinguish a distributed database from a single, centralized database.
- Logical Correlation: The fact that the data have some properties which tie them together, so that we can distinguish a distributed database from a set of local databases or files which are resident at different sites of a computer network.

Chapter 8

Serialization

In computer science, in the context of data storage and transmission, **serialization** is the process of converting a data structure or object into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and "resurrected" later in the same or another computer environment. When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward.

This process of serializing an object is also called **deflating** or **marshalling** an object. The opposite operation, extracting a data structure from a series of bytes, is **deserialization** (which is also called **inflating** or **unmarshalling**).

Uses

Serialization provides:

- a method of persisting objects which is more convenient than writing their properties to a text file on disk, and re-assembling them by reading this back in.
- a method of issuing remote procedure calls, e.g., as in SOAP
- a method for distributing objects, especially in software componentry such as COM, CORBA, etc.
- a method for detecting changes in time-varying data.

For some of these features to be useful, architecture independence must be maintained. For example, for maximal use of distribution, a computer running on a different hardware architecture should be able to reliably reconstruct a serialized data stream, regardless of endianness. This means that the simpler and faster procedure of directly copying the memory layout of the data structure cannot work reliably for all architectures. Serializing the data structure in an architecture independent format means that we do not suffer from the problems of byte ordering, memory layout, or simply different ways of representing data structures in different programming languages.

Inherent to any serialization scheme is that, because the encoding of the data is by definition serial, extracting one part of the serialized data structure requires that the entire object be read from start to end, and reconstructed. In many applications this linearity is an asset, because it enables simple, common I/O interfaces to be utilized to hold and pass on the state of an object. In applications where higher performance is an issue, it can make sense to expend more effort to deal with a more complex, non-linear storage organization.

Even on a single machine, primitive pointer objects are too fragile to save, because the objects to which they point may be reloaded to a different location in memory. To deal with this, the serialization process includes a step called *unswizzling* or *pointer unswizzling* and the deserialization process includes a step called *pointer swizzling*.

Since both serializing and deserializing can be driven from common code, (for example, the *Serialize* function in Microsoft Foundation Classes) it is possible for the common code to do both at the same time, and thus 1) detect differences between the objects being serialized and their prior copies, and 2) provide the input for the next such detection. It is not necessary to actually build the prior copy, since differences can be detected "on the fly". This is a way to understand the technique called differential execution. It is useful in the programming of user interfaces whose contents are time-varying — graphical objects can be created, removed, altered, or made to handle input events without necessarily having to write separate code to do those things.

Consequences

Serialization, however, breaks the opacity of an abstract data type by potentially exposing private implementation details. To discourage competitors from making compatible products, publishers of proprietary software often keep the details of their programs' serialization formats a trade secret. Some deliberately obfuscate or even encrypt the serialized data.

Yet, interoperability requires that applications be able to understand each other's serialization formats. Therefore, remote method call architectures such as CORBA define their serialization formats in detail and often provide methods of checking the consistency of any serialized stream when converting it back into an object.

Human-readable serialization

In the late 1990s, a push to provide an alternative to the standard serialization protocols started: XML was used to produce a human readable text-based encoding. Such an encoding can be useful for persistent objects that may be read and understood by humans, or communicated to other systems regardless of programming language. It has the disadvantage of losing the more compact, byte-stream-based encoding - but then recall that file size (whether for storage or particularly transmission) is tremendously less of an issue than in the early days of computing. A future approach to this consideration could be transparent compression schemes.

XML is today often used for asynchronous transfer of structured data between client and server in Ajax web applications. An alternative for this use case is JSON, a more lightweight text-based serialization protocol that uses JavaScript syntax but is supported in other programming languages as well.

Another alternative, YAML, is effectively a superset of JSON and includes features that make it more powerful for serialization, more "human friendly," and potentially more compact. These features include a notion of tagging data types, support for non-hierarchical data structures, the option to data structure with indentation, and multiple forms of scalar data quoting.

Another human-readable serialization format is the property list format used in NeXTSTEP, GNUstep, and Mac OS X Cocoa.

Scientific serialization

For large volume scientific datasets, such as satellite data and output of numerical climate, weather, or ocean models, specific binary serialization standards have been developed, e.g. HDF, netCDF and the older GRIB.

Programming language support

Several object-oriented programming languages directly support *object serialization* (or *object archival*), either by syntactic sugar elements or providing a standard interface for doing so.

Some of these programming languages are Ruby, Smalltalk, Python, PHP, Objective-C, Java, and the .NET family of languages.

There are also libraries available that add serialization support to languages that lack native support for it.

.NET Framework

In the .NET languages, classes can be serialized and deserialized by adding the `Serializable` attribute to the class.

If new members are added to a serializable class, they can be tagged with the `OptionalField` attribute to allow previous versions of the object to be deserialized without error. This attribute affects only deserialization, and prevents the runtime from throwing an exception if a member is missing from the serialized stream. A member can also be marked with the `NonSerialized` attribute to indicate that it should not be serialized. This will allow the details of those members to be kept secret.

To modify the default deserialization (for example, to automatically initialize a member marked `NonSerialized`), the class must implement the `IDeserializationCallback` interface and define the `IDeserializationCallback.OnDeserialization` method.

Objects may be serialized in binary format for deserialization by other .NET applications. The framework also provides the `SoapFormatter` and `XmlSerializer` objects to support serialization in human-readable, cross-platform XML.

Objective-C

In the Objective-C programming language, serialization (more commonly known as *archiving*) is achieved by overriding the `write:` and `read:` methods in the Object root class. (NB This is in the GNU runtime variant of Objective-C. In the NeXT-style runtime, the implementation is very similar.)

Java

Java provides automatic serialization which requires that the object be marked by implementing the `java.io.Serializable` interface. Implementing the interface marks the class as "okay to serialize," and Java then handles serialization internally. There are no serialization methods defined on the `Serializable` interface, but a serializable class can optionally define methods with certain special names and signatures that if defined, will be called as part of the serialization/deserialization process. The language also allows the developer to override the serialization process more thoroughly by implementing another interface, the `Externalizable` interface, which includes two special methods that are used to save and restore the object's state.

There are three primary reasons why objects are not serializable by default and must implement the `Serializable` interface to access Java's serialization mechanism.

1. Not all objects capture useful semantics in a serialized state. For example, a `Thread` object is tied to the state of the current JVM. There is no context in which a deserialized `Thread` object would maintain useful semantics.
2. The serialized state of an object forms part of its class's compatibility contract. Maintaining compatibility between versions of serializable classes requires additional effort and consideration. Therefore, making a class serializable needs to be a deliberate design decision and not a default condition.
3. Serialization allows access to non-transient private members of a class that are not otherwise accessible. Classes containing sensitive information (for example, a password) should not be serializable or externalizable.

The standard encoding method uses a simple translation of the fields into a byte stream. Primitives as well as non-transient, non-static referenced objects are encoded into the stream. Each object that is referenced by the serialized object and not marked as `transient` must also be serialized; and if any object in the complete graph of non-transient object references is not serializable, then serialization will fail. The developer

can influence this behavior by marking objects as transient, or by redefining the serialization for an object so that some portion of the reference graph is truncated and not serialized.

It is possible to serialize Java objects through JDBC and store them into a database.

While Swing components do implement the Serializable interface, they are **not portable** between different versions of the Java Virtual Machine. As such, a Swing component, or any component which inherits it, may be serialized to an array of bytes, but it is not guaranteed that this storage will be readable on another machine.

ColdFusion

ColdFusion allows data structures to be serialized to WDDX with the `<cfwddx>` tag and to JSON with the SerializeJSON() function.

OCaml

OCaml's standard library provides marshalling through the `Marshal` module (its documentation) and the Pervasives functions `output_value` and `input_value`. While OCaml programming is statically type-checked, uses of the `Marshal` module may break type guarantees, as there is no way to check whether an unmarshalled stream represents objects of the expected type. In OCaml it is difficult to marshal a function or a data structure which contains a function (e.g. an object which contains a method), because executable code in functions cannot be transmitted across different programs. (There is a flag to marshal the code position of a function but it can only be unmarshalled in exactly the same program.)

Perl

Several Perl modules available from CPAN provide serialization mechanisms, including `Storable` and `FreezeThaw`.

Storable includes functions to serialize and deserialize Perl data structures to and from files or Perl scalars.

In addition to serializing directly to files, `Storable` includes the `freeze` function to return a serialized copy of the data packed into a scalar, and `thaw` to deserialize such a scalar. This is useful for sending a complex data structure over a network socket or storing it in a database.

When serializing structures with `Storable`, there are network safe functions that always store their data in a format that is readable on any computer at a small cost of speed. These functions are named `nstore`, `nfreeze`, etc. There are no "n" functions for deserializing these structures — the regular `thaw` and `retrieve` deserialize structures serialized with the "`n`" functions and their machine-specific equivalents.

C/C++

C and C++ do not provide direct support for serialization. However, compiler-based solutions, such as the ODB ORM system for C++, are capable of automatically producing serialization code with few or no modifications to class declarations.

Python

Python implements serialization through the standard library module `pickle`, and to a lesser extent, the older `marshal`. `marshal` does offer the ability to serialize Python code objects, unlike `pickle`. In addition, Python offers the cPickle module, which (as the name suggests) is a C implementation of the pickle module. It can be up to 1000 times faster than the pure Python pickle module, but has a few limitations. The `shelve` module is based on the `pickle` module and can be regarded as a serialized Python dictionary.

As of version 2.6, Python's standard library also includes support for JSON and for XML-encoded property lists. However, these modules only handle basic Python types like strings, integers, and collections of basic types, whereas `pickle` is intended for arbitrary objects.

PHP

PHP implements serialization through the built-in `serialize` and `unserialize` functions. PHP can serialize any of its data types except resources (file pointers, sockets, etc.).

For objects (as of at least PHP 4) there are two "magic methods" that can be implemented within a class — `__sleep()` and `__wakeup()` — that are called from within `serialize()` and `unserialize()`, respectively, that can clean up and restore an object. For example, it may be desirable to close a database connection on serialization and restore the connection on deserialization; this functionality would be handled in these two magic methods. They also permit the object to pick which properties are serialized.

As of PHP 5.1.0 there's another method to hook into internal serialize/unserialize mechanism - `Serializable` interface.

REBOL

REBOL will serialize to file (`save/all`) or to a `string!` (`mold/all`). Strings and files can be deserialized using the polymorphic `load` function.

Ruby

Ruby includes the standard module `Marshal` with 2 methods `dump` and `load`, akin to the standard Unix utilities `dump` and `restore`. These methods serialize to the standard class `String`, that is, they effectively become a sequence of bytes.

Some objects cannot be serialized (doing so would raise a `TypeError` exception):

- bindings,
- procedure objects,
- instances of class IO,
- singleton objects

If a class requires custom serialization (for example, it requires certain cleanup actions done on dumping / restoring), it can be done by implementing 2 methods: `_dump` and `_load`. The instance method `_dump` should return a `String` object containing all the information necessary to reconstitute objects of this class and all referenced objects up to a maximum depth given as an integer parameter (a value of -1 implies that depth checking should be disabled). The class method `_load` should take a `String` and return an object of this class.

Smalltalk

Squeak Smalltalk

There are several ways in Squeak Smalltalk to serialize and store objects. The easiest and most used method will be shown below (where?). Other classes of interest in Squeak for serializing objects are `SmartRefStream` and `ImageSegment`.

Cincom Smalltalk and Smalltalk/X

Both provide a so called "binary-object storage framework", which support serialization into and retrieval from a compact binary form. Both handle cyclic, recursive and shared structures, storage/retrieval of class and metaclass info and include mechanisms for "on the fly" object migration (i.e. to convert instances which were written by an older version of a class with a different object layout). The APIs are similar (storeBinary/readBinary), but the encoding details are different, making these two formats incompatible. However, the Smalltalk/X code is open source and free and can be loaded into other Smalltalks to allow for cross-dialect object interchange.

Other Smalltalk dialects

Object serialization is not part of the ANSI Smalltalk specification. As a result, the code to serialize an object varies by Smalltalk implementation. The resulting binary data also varies. For instance, a serialized object created in Squeak Smalltalk cannot be restored in Ambrai Smalltalk. Consequently, various applications that do work on multiple Smalltalk

implementations that rely on object serialization cannot share data between these different implementations. These applications include the MinneStore object database and some RPC packages. A solution to this problem is SIXX , which is an package for multiple Smalltalks that uses an XML-based format for serialization.

Lisp

Generally a Lisp data structure can be serialized with the functions "`read`" and "`print`". A variable foo containing, for example, a list of arrays would be printed by `(print foo)`. Similarly an object can be read from a stream named s by `(read s)`. These two parts of the Lisp implementation are called the Printer and the Reader. The output of "`print`" is human readable; it uses lists demarked by parentheses, for example: `(4 2.9 "x" y)`.

In many types of Lisp, including Common Lisp, the printer cannot represent every type of data because it is not clear how to do so. In Common Lisp for example the printer cannot print CLOS objects. Instead the programmer may write a method on the generic function `print-object`, this will be invoked when the object is printed. This is somewhat similar to the method used in Ruby.

Lisp code itself is written in the syntax of the reader, called read syntax. Most languages use separate and different parsers to deal with code and data, Lisp only uses one. A file containing lisp code may be read into memory as a data structure, transformed by another program, then possibly executed or written out.

Notice that not all readers/writers support cyclic, recursive or shared structures.

Haskell

In Haskell, serialization is supported for types by inheritance of the Read and Show type classes. Every type that inherits the `Read` class defines a function that will extract the data from the string representation of the dumped data. The `Show` class, in turn, contains the show function from which a string representation of the object can be generated.

The programmer need not define the functions explicitly—merely declaring a type to be deriving Read or deriving Show, or both, can make the compiler generate the appropriate functions for many cases (function types, for example, cannot automatically derive Show or Read).

Windows PowerShell

Windows PowerShell implements serialization through the built-in cmdlet `Export-CliXML`. `Export-CliXML` serializes .NET objects and stores the resulting XML in a file.

To reconstitute the objects, use the `Import-CliXML` cmdlet, which generates a deserialized object from the XML in the exported file. Deserialized objects, often known

as "property bags" are not live objects; they are snapshots that have properties, but no methods.

Two dimensional data structures can also be (de)serialized in CSV format using the built-in cmdlets `Import-CSV` and `Export-CSV`.

Chapter 9

Carbonado (Java) and Persistent Data Structure

Carbonado (Java)

Carbonado is an open source relational database mapping framework, written in Java. Rather than following a typical O/R mapping approach, the relational model is preserved, while still being object-oriented. Not being tied to specific features of SQL or JDBC, Carbonado also supports non-SQL database products such as Berkeley DB. In doing so, relational features such as queries and indexes are supported, without the overhead of SQL.

History

Carbonado was originally developed for internal use by Amazon.com, as a revision to an earlier framework. It was released as an Apache licensed open-source project in October 2006.

Entity definitions

Relational entities are known as *Storables* in Carbonado, and they are defined by an interface or abstract class. Annotations are required to specify features which cannot defined by Java interface alone. Every Storable must have an annotation describing the primary key of the entity.

```
@PrimaryKey("entityId")
public interface MyEntity extends Storable {
    long getEntityId();
    void setEntityId(long id);

    String getMessage();
    void setMessage(String message);
}
```

Carbonado Storables are not pure POJOs, and they must always extend the Storable superclass. By doing so, they gain access to various methods built into it. A Storable definition may also contain business logic, following the active record pattern.

The actual implementation of the Storable is generated at runtime by Carbonado itself. The standard object methods of toString, equals and hashCode are also generated. This greatly simplifies the process of defining new entities, since no boilerplate code needs to be written.

The process of loading a Storable by key starts by calling a factory method to create an uninitialized instance:

```
Repository repo = ...
Storage<MyEntity> storage = repo.storageFor(MyEntity.class);
MyEntity entity = storage.prepare();
```

Next, the key properties are set and load is called:

```
entity.setEntityId(id);
entity.load();
```

Repository usage

A *Repository* is a gateway to the underlying database. A few core implementations are available, which include:

- JDBC access
- Berkeley DB
- Berkeley DB Java Edition
- An in memory database

In addition, composite Repositories exist which support simple replication and logging.

All Repositories are created using a builder pattern. Each type of builder supports options specific to the Repository type. When a Repository instance is built, it only adheres to the standard interface. Access to specific features is provided by a *Capability* interface.

```
BDBRepositoryBuilder builder = new BDBRepositoryBuilder();
builder.setName(name);
builder.setEnvironmentHome(envHome);
builder.setTransactionWriteNoSync(true);
Repository repo = builder.build();
```

Query execution

Carbonado queries are defined by a simple filter expression and an order-by specification. Compared to SQL, the filter closely resembles a "where" clause. Filters can

include joined properties and they may also include sub filters. This simple example queries for entities with a given message:

```
Storage<MyEntity> storage = repo.storageFor(MyEntity.class);
Query<MyEntity> query = storage.query("message = ?").with(message);
List<MyEntity> matches = query.fetch().toList();
```

Transactions

Transactions are created from a Repository instance, and they define a thread-local scope. Multiple persist operations are automatically grouped together, and commit must be called to complete the transaction.

```
Transaction txn = repo.enterTransaction();
try {
    MyEntity entity = storage.prepare();
    entity.setEntityId(1);
    entity.setMessage("hello");
    entity.insert();

    entity = storage.prepare();
    entity.setEntityId(2);
    entity.setMessage("world");
    entity.insert();

    txn.commit();
} finally {
    txn.exit();
}
```

This design approach shows how Carbonado is not like an O/R mapping framework. Such frameworks typically hide the concept of transactions entirely, often by using sessions which track changes. In Carbonado, all actions are direct.

Persistent data structure

In computing, a **persistent data structure** is a data structure which always preserves the previous version of itself when it is modified; such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure. A persistent data structure is *not* a data structure committed to persistent storage, such as a disk; this is a different and unrelated sense of the word "persistent."

A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified. The data structure is **fully persistent** if every version can be both accessed and modified. If there is also a meld or merge operation that can create a

new version from two previous versions, the data structure is called **confluently persistent**. Structures that are not persistent are called **ephemeral**.

These types of data structures are particularly common in logical and functional programming, and in a purely functional program all data is immutable, so all data structures are automatically fully persistent. Persistent data structures can also be created using in-place updating of data and these may, in general, use less time or storage space than their purely functional counterparts.

While persistence can be achieved by simple copying, this is inefficient in time and space, because most operations make only small changes to a data structure. A better method is to exploit the similarity between the new and old versions to share structure between them, such as using the same subtree in a number of tree structures. However, because it rapidly becomes infeasible to determine how many previous versions share which parts of the structure, and because it is often desirable to discard old versions, this necessitates an environment with garbage collection.

Perhaps the simplest persistent data structure is the singly-linked list or *cons*-based list, a simple list of objects formed by each carrying a reference to the next in the list. This is persistent because we can take a *tail* of the list, meaning the last k items for some k, and add new nodes on to the front of it. The tail will not be duplicated, instead becoming shared between both the old list and the new list. So long as the contents of the tail are immutable, this sharing will be invisible to the program.

Many common reference-based data structures, such as red-black trees, and queues, can easily be adapted to create a persistent version.

Chapter 10

Java Persistence API and Snapshot (Computer Storage)

Java Persistence API

The **Java Persistence API**, sometimes referred to as **JPA**, is a Java programming language framework managing relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.

The Java Persistence API originated as part of the work of the JSR 220 Expert Group. JPA 2.0 is the work of the JSR 317 Expert Group.

Persistence in this context covers three areas:

- the API itself, defined in the `javax.persistence` package
- the Java Persistence Query Language (JPQL)
- object/relational metadata

History

The final release date of the JPA 1.0 specification is 11 May, 2006. The JPA 2.0 specification has been released at 10 Dec, 2009. JPA is a replacement for the much criticized EJB 2.0 and EJB 2.1 entity beans.

Entities

A persistence entity is a lightweight Java class whose state is typically persisted to a table in a relational database. Instances of such an entity correspond to individual rows in the table. Entities typically have relationships with other entities, and these relationships are expressed through object/relational metadata. Object/relational metadata can be specified directly in the entity class file by using annotations, or in a separate XML descriptor file distributed with the application.

The Java Persistence Query Language

The Java Persistence Query Language (JPQL) makes queries against entities stored in a relational database. Queries resemble SQL queries in syntax, but operate against entity objects rather than directly with database tables.

Motivation for creating the Java Persistence API

Many enterprise Java developers use lightweight persistent objects provided by open-source frameworks or Data Access Objects instead of entity beans: entity beans and enterprise beans had a reputation of being too heavyweight and complicated, and one could only use them in Java EE application servers. Many of the features of the third-party persistence frameworks were incorporated into the Java Persistence API, and as of 2006 projects like Hibernate (version 3.2) and Open-Source Version TopLink Essentials have become implementations of the Java Persistence API.

Related Technologies

Enterprise JavaBeans

The EJB 3.0 specification (itself part of the Java EE 5 platform) included a definition of the Java Persistence API. However, end-users do not need an EJB container or a Java EE application server in order to run applications that use this persistence API . Future versions of the Java Persistence API will be defined in a separate JSR and specification rather than in the EJB JSR/specification.

The Java Persistence API replaces the persistence solution of EJB 2.0 CMP (Container Managed Persistence).

Java Data Objects API

The Java Persistence API was developed in part to unify the Java Data Objects API, and the EJB 2.0 Container Managed Persistence (CMP) API. As of 2009 most products supporting each of those APIs support the Java Persistence API.

The Java Persistence API specifies relational persistence (ORM: Object relational mapping) only for RDBMS (although providers exists who support other datastores). The Java Data Objects specification(s) provides relational persistence (ORM), as well as persistence to other types of datastores.

Service Data Object API

The designers of the Java Persistence API aimed to provide for relational persistence, with many of the key areas taken from object-relational mapping tools such as Hibernate and TopLink. It is generally accepted that the Java Persistence API is a significant improvement on the EJB 2.0 specification. The Service Data Objects (SDO) API (JSR

235) has a very different objective to the Java Persistence API and is considered complementary. The SDO API is designed for service-oriented architectures, multiple data formats rather than only relational data, and multiple programming languages. The Java Community Process manages the Java version of the SDO API; the C++ version of the SDO API is managed via OASIS.

Hibernate

Hibernate provides an open source object-relational mapping framework for Java. Versions 3.2 and later provide an implementation for the Java Persistence API.

Gavin King founded Hibernate. He represented JBoss on JSR220, the JCP expert group charged with developing JPA. This led to ongoing controversy and speculation surrounding the relationship between JPA and Hibernate. Sun Microsystems has stated that ideas came from several frameworks, including Hibernate and JDO.

JPA 2.0

Development of a new version of JPA, namely JPA 2.0 JSR 317 was started in July 2007. JPA 2.0 was approved as final on December 10, 2009.

The focus of JPA 2.0 was to address features that were present in some of the popular ORM vendors but couldn't gain consensus approval for JPA 1.0.

The main features included in this update are:

- Expanded object/relational mapping functionality
  - support for collections of embedded objects
  - multiple levels of embedded objects
  - ordered lists
  - combinations of access types
- A criteria query API
- standardization of query 'hints'
- standardization of additional metadata to support DDL generation
- support for validation

Vendors supporting JPA 2.0

- DataNucleus (formerly JPOX)
- EclipseLink (formerly Oracle TopLink)
- JBoss Hibernate
- OpenJPA
- IBM, via its Feature Pack for OSGi Applications and JPA 2.0 for WebSphere Application Server

Snapshot (computer storage)

In computer systems, a **snapshot** is the state of a system at a particular point in time. The term was coined as an analogy to that in photography. It can refer to an actual copy of the state of a system or to a capability provided by certain systems.

Rationale

A full backup of a large data set may take a long time to complete. On multi-tasking or multi-user systems, there may be writes to that data while it is being backed up. This prevents the backup from being atomic and introduces a version skew that may result in data corruption. For example, if a user moves a file into a directory that has already been backed up, then that file would be completely missing on the backup media, since the backup operation had already taken place before the addition of the file. Version skew may also cause corruption with files which change their size or contents underfoot while being read.

One approach to safely backing up live data is to temporarily disable write access to data during the backup, either by stopping the accessing applications or by using the locking API provided by the operating system to enforce exclusive read access. This is tolerable for low-availability systems (on desktop computers and small workgroup servers, on which regular downtime is acceptable). High-availability 24/7 systems, however, cannot bear service stoppages.

To avoid downtime, high-availability systems may instead perform the backup on a *snapshot*—a read-only copy of the data set frozen at a point in time—and allow applications to continue writing to their data. Most snapshot implementations are efficient and can create snapshots in *O(1)*. In other words, the time and I/O needed to create the snapshot does not increase with the size of the data set, whereas the same for a direct backup is proportional to the size of the data set. In some systems once the initial snapshot is taken of a data set, subsequent snapshots copy the changed data only, and use a system of pointers to reference the initial snapshot. This method of pointer-based snapshots consumes less disk capacity than if the data set was repeatedly cloned.

Read-write snapshots are sometimes called branching snapshots, because they implicitly create diverging versions of their data. Aside from backups and data recovery, read-write snapshots are frequently used in virtualization, sandboxing and virtual hosting setups because of their usefulness in managing changes to large sets of files.

Implementations

Volume managers

Some Unix systems have snapshot-capable logical volume managers. These implement copy-on-write on entire block devices by copying changed blocks—just before they are

to be overwritten—to other storage, thus preserving a self-consistent past image of the block device. Filesystems on this image can later be mounted as if it were on read-only media. Block-level snapshotting is almost always less space-efficient than direct file system support for snapshots.

File systems

Some file systems, such as WAFL, fossil for Plan 9 from Bell Labs or ODS-5, internally track old versions of files and make snapshots available through a special namespace. Others, like UFS2, provide an operating system API for accessing file histories. In NTFS, access to snapshots is provided by the Volume Shadow-copying Service (VSS) in Windows XP and Windows Server 2003 and Shadow Copy in Windows Vista. Snapshots have also been available in the NSS (Novell Storage Services) file system on NetWare since version 4.11, and more recently on Linux platforms in the Open Enterprise Server product.

Sun Microsystems ZFS has a hybrid implementation which tracks read-write snapshots at the block level, but makes branched file sets nameable to user applications as "clones".

Time Machine, included in Apple's Mac OS X v10.5 operating system, is not a snapshotting scheme but a system-level incremental backup service: it merely watches mounted volumes for changes and copies changed files periodically to a specially-designated volume using hard links.

In databases

The SQL specification mandates four levels of transaction isolation. In the highest, *SERIALIZABLE*, a snapshot is implicitly created at the start of every transaction. The backup utilities for many popular SQL databases use this feature to generate self-consistent dumps of table data.

In virtualization

System emulators host a guest operating system in a virtual machine; some (including VMware, VirtualBox, Qemu and Virtual PC) can perform whole-system snapshots by dumping the entire machine state to a backing file and redirecting future guest writes to a second file, which then acts as a copy-on-write table.

Other applications

Software transactional memory is a scheme which applies the same concepts to data structures held only in memory.

Chapter 11

QuickDB ORM

QuickDB is an object-relational mapping framework for the Java software platform. It was developed by Diego Sarmentero along with others and is licensed under the LGPL License. Version for .NET, Python and PHP are also being developed.

Concepts

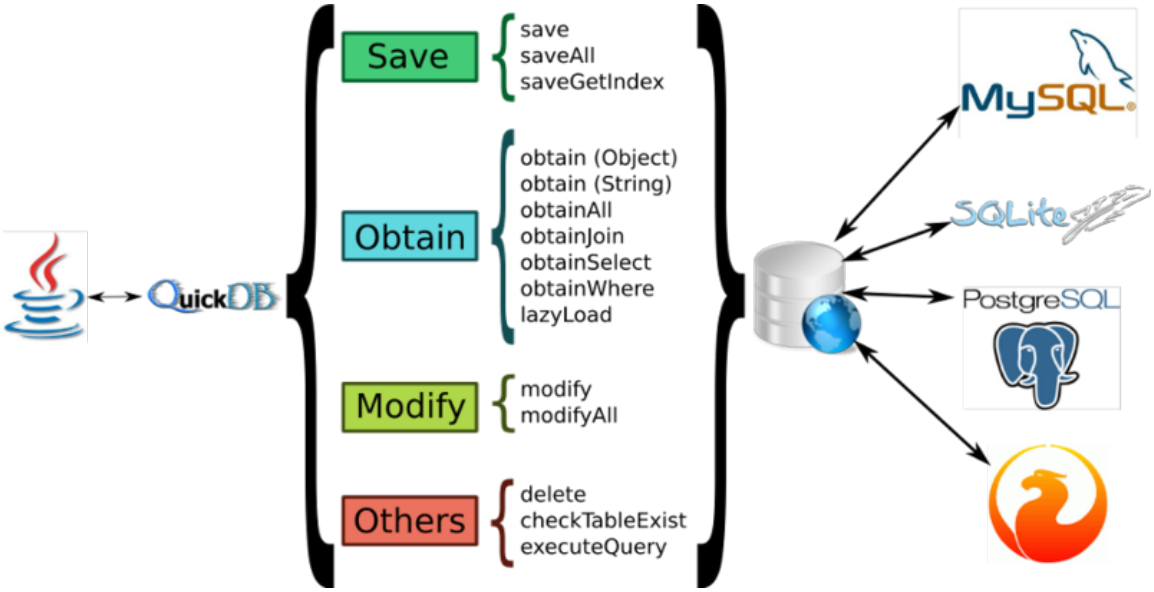
QuickDB allows a developer to focus on the definition of the entities that represent the tables of the database and perform operations that allow the interaction between these entities and the database without having to perform tedious configurations, leaving to the library the task to infer the object structure and make the mapping of the object to the Database.

The software aims not only to simplify the task of mapping the attributes between a traditional relational database and the objects from the data model, but in turn, make the use of the library intuitive for the developer, leaving aside configuration tasks. Where each operation involves only the task (save, modify, ...) and the subject where it should apply to (the object).

QuickDB as other tools for object-relational mapping, seeks to resolve the differences between the two co-existing data models: *Object-Oriented Model* and the *Relational Model*. To address this problem, it takes a fully object-oriented approach, where structures like "Objects composed of other objects", "Inheritance", "Collections" (one-to-many and many-to-many) are recognized by default as common entities, and also other features such as automatic table creation and modification of tables dynamically if the structure of the object change over time (addition of new attributes) are included. QuickDB not require the implementation of any interface, or the use of inheritance by the data model to be persist, it is based simply on certain naming conventions for attributes to infer relevant information about the Object. However, it is possible to use annotations to set certain characteristics of the object, which gives the assurance that everything that QuickDB recognize by default, can be also managed completely by the developer. For queries, QuickDB pretends to maintain this approach where the developer works with the data model in a fully object-oriented way, and therefore the SQL statements (although

they are permitted) are not necessary, and can be used by default a Query system where the condition to be evaluated is specified with a simple reference to the attributes in the objects from the data model.

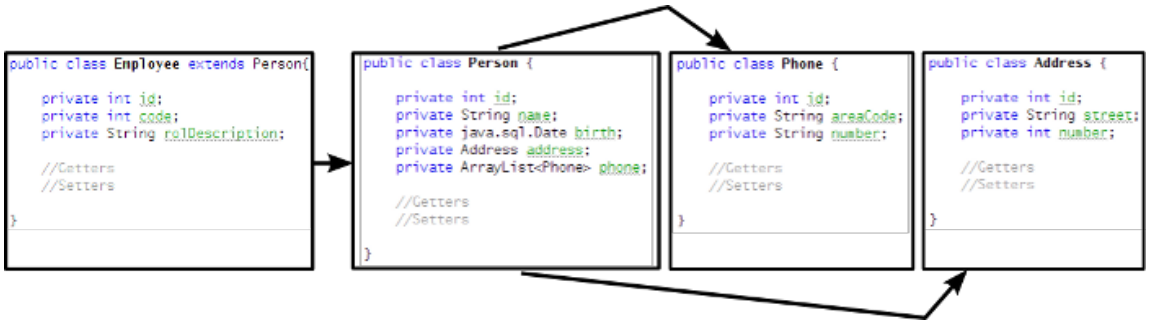
Capabilities



This capabilities can be combined with Inheritance, Compound Objects, Collections (Many to Many Relation, One to Many Relation) and Automatic Table Creation.

Using QuickDB

Example Classes::




```
//Create AdminBase instance for MySQL
AdminBase admin = new AdminBase(AdminBase.DATABASE.MYSQL, "localhost",
        "3306", "exampleQuickDB", "root", "");

//Create Address Object
Address a = new Address();
a.setNumber(123);
a.setStreet("unnamed street");
```

```
//Create a Collection of Phone Objects
Phone p1 = new Phone();
p1.setAreaCode("351");
p1.setNumber("123456");
Phone p2 = new Phone();
p2.setAreaCode("351");
p2.setNumber("4567890");
ArrayList<Phone> phones = new ArrayList<Phone>();
phones.add(p1);
phones.add(p2);

//Create Employee Object
Employee e = new Employee();
e.setCode(555);
e.setRolDescription("play ping pong");
e.setName("Diego Sarmentero");
e.setBirth(new java.sql.Date(100, 4, 20));
e.setAddress(a);
e.setPhone(phones);
```

Operation: save

```
//Create the Table automatically if not exists
admin.save(e); //Salvar Employee
```

Result: save

Tables are automatically created and the Employee object is saved with all the objects related to it.

Address

id	street	number
1	unnamed street	123

Phone

id	areaCode	number
1	351	123456
2	351	4567890

Person

id	name	birth	address
1	Diego Sarmentero	2000-05-20	1

PersonPhone

id base related

1 1 1

2 1 2

Employee

id code rolDescription parent_id

1 555 play ping pong 1

Operation: obtain

The Employee object which inherited attribute *address* has the value "unnamed street" in *street* is retrieved.

```
admin.obtain(e, "address.street = 'unnamed street'");
admin.obtain(e).where("street", Address.class).equal("unnamed
street").find();
```

Operation: modify

Change the name from the Employee object and add a new Phone to the collection (the obtain operation must be performed previously to retrieve the object).

```
e.setName("Leonardo");
Phone p = new Phone();
p.setAreaCode("123");
p.setNumber("98765");
e.getPhone().add(p);

admin.modify(e);
```

Operation: delete

Delete an specific Employee object (after retrieve the object with obtain operation).

```
admin.delete(e);
```

Inheritance: AdminBinding

QuickDB also provides the possibility of allowing the developer to create the data model extending the *AdminBinding* Class (but this is not required, this is another available resource to simplify some processes), which serves as a "link" between the entities and *AdminBase* to allow the operations that interact with the database to be executed from the object itself and not use *AdminBase* intermediary as directly.

AdminBinding not cover all the functionality that AdminBase provides, as it only handles those operations that are specific to the object that contains them, leaving out those that refer to collections, etc.

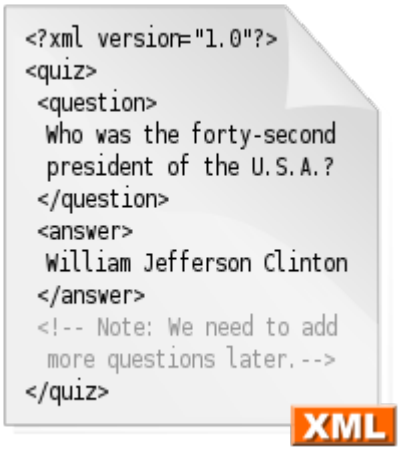
More Features

QuickDB includes many features that can be consulted on the Project Page. An important detail to take into account is that with every new feature that is added, the simplicity of QuickDB is maintained without sacrifice functionality.

Chapter 12

XML

XML



Filename extension	`.xml`
Internet media type	`application/xml, text/xml (deprecated in an expired draft)`
Uniform Type Identifier	public.xml
Developed by	World Wide Web Consortium
Type of format	Markup language
Extended from	SGML
Extended to	Numerous, including: XHTML, RSS, Atom, KML
Standard(s)	1.0 (Fifth Edition) November 26, 2008; 2 years ago 1.1 (Second Edition) August 16, 2006; 4 years ago

Open format?	Yes

Extensible Markup Language (XML)

Current Status	Published
Year Started	1996
Editors	Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, John Cowan
Related Standards	XML Schema
Domain	Data Serialization
Abbreviation	XML

Extensible Markup Language (**XML**) is a set of rules for encoding documents in machine-readable form. It is defined in the XML 1.0 Specification produced by the W3C, and several other related specifications, all gratis open standards.

XML's design goals emphasize simplicity, generality, and usability over the Internet. It is a textual data format with strong support via Unicode for the languages of the world. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures, for example in web services.

Many application programming interfaces (APIs) have been developed that software developers use to process XML data, and several schema systems exist to aid in the definition of XML-based languages.

As of 2009, hundreds of XML-based languages have been developed, including RSS, Atom, SOAP, and XHTML. XML-based formats have become the default for most office-productivity tools, including Microsoft Office (Office Open XML), OpenOffice.org (OpenDocument), and Apple's iWork.

Key terminology

The material in this section is based on the XML Specification. This is not an exhaustive list of all the constructs which appear in XML; it provides an introduction to the key constructs most often encountered in day-to-day use.

(Unicode) Character
: By definition, an XML document is a string of characters. Almost every legal Unicode character may appear in an XML document.

Processor and Application
: The *processor* analyzes the markup and passes structured information to an *application*. The specification places requirements on what an XML processor

must do and not do, but the application is outside its scope. The processor (as the specification calls it) is often referred to colloquially as an *XML parser*.

Markup and Content
: The characters which make up an XML document are divided into *markup* and *content*. Markup and content may be distinguished by the application of simple syntactic rules. All strings which constitute markup either begin with the character "<" and end with a ">", or begin with the character "&" and end with a ";". Strings of characters which are not markup are content.

Tag
: A markup construct that begins with "<" and ends with ">". Tags come in three flavors: *start-tags*, for example `<section>`, *end-tags*, for example `</section>`, and *empty-element tags*, for example `<line-break />`.

Element
: A logical component of a document which either begins with a start-tag and ends with a matching end-tag, or consists only of an empty-element tag. The characters between the start- and end-tags, if any, are the element's *content*, and may contain markup, including other elements, which are called *child elements*. An example of an element is `<Greeting>Hello, world.</Greeting>`.
  Another is `<line-break />`.

Attribute
: A markup construct consisting of a name/value pair that exists within a start-tag or empty-element tag. In the example (below) the element *img* has two attributes, *src* and *alt*:
  `<img src="madonna.jpg" alt='Foligno Madonna, by Raphael' />`.
  Another example would be `<step number="3">Connect A to B.</step>` where the name of the attribute is "number" and the value is "3".

XML Declaration
: XML documents may begin by declaring some information about themselves, as in the following example.

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Characters and escaping

XML documents consist entirely of characters from the Unicode repertoire. Except for a small number of specifically excluded control characters, any character defined by Unicode may appear within the content of an XML document. The selection of characters which may appear within markup is somewhat more limited but still large.

XML includes facilities for identifying the *encoding* of the Unicode characters which make up the document, and for expressing characters which, for one reason or another, cannot be used directly.

Valid characters

Unicode code points in the following ranges are valid in XML 1.0 documents:

- U+0009, U+000A, U+000D: these are the only C0 controls accepted in XML 1.0;
- U+0020–U+D7FF, U+E000–U+FFFD: this excludes *some* (not all) non-characters in the BMP (all surrogates, U+FFFE and U+FFFF are forbidden);
- U+10000–U+10FFFF: this includes *all* code points in supplementary planes, including non-characters.

XML 1.1 extends the set of allowed characters to include all the above, plus the remaining characters in the range U+0001–U+001F. At the same time, however, it restricts the use of C0 and C1 control characters other than U+0009, U+000A, U+000D, and U+0085 by requiring them to be written in escaped form (for example U+0001 must be written as &#x01; or its equivalent). In the case of C1 characters, this restriction is a backwards incompatibility; it was introduced to allow common encoding errors to be detected.

The code point U+0000 is the only character that is not permitted in any XML 1.0 or 1.1 document.

Encoding detection

The Unicode character set can be encoded into bytes for storage or transmission in a variety of different ways, called "encodings". Unicode itself defines encodings which cover the entire repertoire; well-known ones include UTF-8 and UTF-16. There are many other text encodings which pre-date Unicode, such as ASCII and ISO/IEC 8859; their character repertoires in almost every case are subsets of the Unicode character set.

XML allows the use of any of the Unicode-defined encodings, and any other encodings whose characters also appear in Unicode. XML also provides a mechanism whereby an XML processor can reliably, without any prior knowledge, determine which encoding is being used. Encodings other than UTF-8 and UTF-16 will not necessarily be recognized by every XML parser.

Escaping

XML provides *escape* facilities for including characters which are problematic to include directly. For example:

- The characters "<" and "&" are key syntax markers and may *never* appear in content outside of a CDATA section.
- Some character encodings support only a subset of Unicode: for example, it is legal to encode an XML document in ASCII, but ASCII lacks code points for Unicode characters such as "é".
- It might not be possible to type the character on the author's machine.
- Some characters have glyphs that cannot be visually distinguished from other characters: examples are non-breaking-space (` `) and Cyrillic Capital Letter A (`&#x410;`).

There are five *predefined entities*: `<` represents "<", `>` represents ">", `&` represents "&", `&apos;` represents ', and `&quot;` represents ". All permitted Unicode characters may be represented with a *numeric character reference*. Consider the Chinese character "中", whose numeric code in Unicode is hexadecimal 4E2D, or decimal 20,013. A user whose keyboard offers no method for entering this character could still insert it in an XML document encoded either as `&#20013;` or `&#x4e2d;`. Similarly, the string "`I <3 Jörg`" could be encoded for inclusion in an XML document as "`I <3 J&#xF6;rg`".

"`&#0;`" is not permitted, however, as the null character is one of the control characters excluded from XML, even when using a numeric character reference. An alternative encoding mechanism such as Base64 is needed to represent such characters.

Comments

Comments may appear anywhere in a document outside other markup. Comments cannot appear before the XML declaration. The string "`--`" (double-hyphen) is not allowed (as it is used to delimit comments). The ampersand has no special significance within comments, so entity and character references are not recognized as such, and there is no way to represent characters outside the character set of the document encoding.

An example of a valid comment: "`<!-- no need to escape <code> & such in comments -->`"

International use

XML supports the direct use of almost any Unicode character in element names, attributes, comments, character data, and processing instructions (other than the ones that have special symbolic meaning in XML itself, such as the less-than sign, "<"). Therefore, the following is a well-formed XML document, even though it includes both Chinese and Cyrillic characters:

```
<?xml version="1.0" encoding="UTF-8" ?>
<外语>Китайська мова</外语>
```

Well-formedness and error-handling

The XML specification defines an XML document as a text which is well-formed, i.e. it satisfies a list of syntax rules provided in the specification. The list is fairly lengthy; some key points are:

- It contains only properly encoded legal Unicode characters.
- None of the special syntax characters such as "<" and "&" appear except when performing their markup-delineation roles.
- The begin, end, and empty-element tags which delimit the elements are correctly nested, with none missing and none overlapping.

- The element tags are case-sensitive; the beginning and end tags must match exactly.
- There is a single "root" element which contains all the other elements.

The definition of an *XML document* excludes texts which contain violations of well-formedness rules; they are simply not XML. An XML processor which encounters such a violation is required to report such errors and to cease normal processing. This policy, occasionally referred to as draconian, stands in notable contrast to the behavior of programs which process HTML, which are designed to produce a reasonable result even in the presence of severe markup errors. XML's policy in this area has been criticized as a violation of Postel's law.

Schemas and validation

In addition to being well-formed, an XML document may be *valid*. This means that it contains a reference to a Document Type Definition (DTD), and that its elements and attributes are declared in that DTD and follow the grammatical rules for them that the DTD specifies.

XML processors are classified as *validating* or *non-validating* depending on whether or not they check XML documents for validity. A processor which discovers a validity error must be able to report it, but may continue normal processing.

A DTD is an example of a *schema* or *grammar*. Since the initial publication of XML 1.0, there has been substantial work in the area of schema languages for XML. Such schema languages typically constrain the set of elements that may be used in a document, which attributes may be applied to them, the order in which they may appear, and the allowable parent/child relationships.

DTD

The oldest schema language for XML is the Document Type Definition (DTD), inherited from SGML.

DTDs have the following benefits:

- DTD support is ubiquitous due to its inclusion in the XML 1.0 standard.
- DTDs are terse compared to element-based schema languages and consequently present more information in a single screen.
- DTDs allow the declaration of standard public entity sets for publishing characters.
- DTDs define a *document type* rather than the types used by a namespace, thus grouping all constraints for a document in a single collection.

DTDs have the following limitations:

- They have no explicit support for newer features of XML, most importantly namespaces.
- They lack expressiveness. XML DTDs are simpler than SGML DTDs and there are certain structures that cannot be expressed with regular grammars. DTDs only support rudimentary datatypes.
- They lack readability. DTD designers typically make heavy use of parameter entities (which behave essentially as textual macros), which make it easier to define complex grammars, but at the expense of clarity.
- They use a syntax based on regular expression syntax, inherited from SGML, to describe the schema. Typical XML APIs such as SAX do not attempt to offer applications a structured representation of the syntax, so it is less accessible to programmers than an element-based syntax may be.

Two peculiar features that distinguish DTDs from other schema types are the syntactic support for embedding a DTD within XML documents and for defining *entities*, which are arbitrary fragments of text and/or markup that the XML processor inserts in the DTD itself and in the XML document wherever they are referenced, like character escapes.

DTD technology is still used in many applications because of its ubiquity.

XML Schema

A newer schema language, described by the W3C as the successor of DTDs, is XML Schema, often referred to by the initialism for XML Schema instances, XSD (XML Schema Definition). XSDs are far more powerful than DTDs in describing XML languages. They use a rich datatyping system and allow for more detailed constraints on an XML document's logical structure. XSDs also use an XML-based format, which makes it possible to use ordinary XML tools to help process them.

RELAX NG

RELAX NG was initially specified by OASIS and is now also an ISO international standard (as part of DSDL). RELAX NG schemas may be written in either an XML based syntax or a more compact non-XML syntax; the two syntaxes are isomorphic and James Clark's Trang conversion tool can convert between them without loss of information. RELAX NG has a simpler definition and validation framework than XML Schema, making it easier to use and implement. It also has the ability to use datatype framework plug-ins; a RELAX NG schema author, for example, can require values in an XML document to conform to definitions in XML Schema Datatypes.

Schematron

Schematron is a language for making assertions about the presence or absence of patterns in an XML document. It typically uses XPath expressions.

ISO DSDL and other schema languages

The ISO DSDL (Document Schema Description Languages) standard brings together a comprehensive set of small schema languages, each targeted at specific problems. DSDL includes RELAX NG full and compact syntax, Schematron assertion language, and languages for defining datatypes, character repertoire constraints, renaming and entity expansion, and namespace-based routing of document fragments to different validators. DSDL schema languages do not have the vendor support of XML Schemas yet, and are to some extent a grassroots reaction of industrial publishers to the lack of utility of XML Schemas for publishing.

Some schema languages not only describe the structure of a particular XML format but also offer limited facilities to influence processing of individual XML files that conform to this format. DTDs and XSDs both have this ability; they can for instance provide the infoset augmentation facility and attribute defaults. RELAX NG and Schematron intentionally do not provide these.

Related specifications

A cluster of specifications closely related to XML have been developed, starting soon after the initial publication of XML 1.0. It is frequently the case that the term "XML" is used to refer to XML together with one or more of these other technologies which have come to be seen as part of the XML core.

- XML Namespaces enable the same document to contain XML elements and attributes taken from different vocabularies, without any naming collisions occurring. Although XML Namespaces are not part of the XML specification itself, virtually all XML software also supports XML Namespaces.
- XML Base defines the `xml:base` attribute, which may be used to set the base for resolution of relative URI references within the scope of a single XML element.
- The XML Information Set or *XML infoset* describes an abstract data model for XML documents in terms of *information items*. The infoset is commonly used in the specifications of XML languages, for convenience in describing constraints on the XML constructs those languages allow.
- xml:id Version 1.0 asserts that an attribute named `xml:id` functions as an "ID attribute" in the sense used in a DTD.
- XPath defines a syntax named *XPath expressions* which identifies one or more of the internal components (elements, attributes, and so on) included in an XML document. XPath is widely used in other core-XML specifications and in programming libraries for accessing XML-encoded data.
- XSLT is a language with an XML-based syntax that is used to transform XML documents into other XML documents, HTML, or other, unstructured formats such as plain text or RTF. XSLT is very tightly coupled with XPath, which it uses to address components of the input XML document, mainly elements and attributes.

- XSL Formatting Objects, or XSL-FO, is a markup language for XML document formatting which is most often used to generate PDFs.
- XQuery is an XML-oriented query language strongly rooted in XPath and XML Schema. It provides methods to access, manipulate and return XML.
- XML Signature defines syntax and processing rules for creating digital signatures on XML content.
- XML Encryption defines syntax and processing rules for encrypting XML content.

Some other specifications conceived as part of the "XML Core" have failed to find wide adoption, including XInclude, XLink, and XPointer.

Use on the Internet

It is common for XML to be used in interchanging data over the Internet. RFC 3023 gives rules for the construction of Internet Media Types for use when sending XML. It also defines the types "application/xml" and "text/xml", which say only that the data is in XML, and nothing about its semantics. The use of "text/xml" has been criticized as a potential source of encoding problems and is now in the process of being deprecated. RFC 3023 also recommends that XML-based languages be given media types beginning in "application/" and ending in "+xml"; for example "application/svg+xml" for SVG.

Further guidelines for the use of XML in a networked context may be found in RFC 3470, also known as IETF BCP 70; this document is very wide-ranging and covers many aspects of designing and deploying an XML-based language.

Programming interfaces

The design goals of XML include "It shall be easy to write programs which process XML documents." Despite this fact, the XML specification contains almost no information about how programmers might go about doing such processing. The XML Infoset provides a vocabulary to refer to the constructs within an XML document, but once again does not provide any guidance on how to access this information. A variety of APIs for accessing XML have been developed and used, and some have been standardized.

Existing APIs for XML processing tend to fall into these categories:

- Stream-oriented APIs accessible from a programming language, for example SAX and StAX.
- Tree-traversal APIs accessible from a programming language, for example DOM.
- XML data binding, which provides an automated translation between an XML document and programming-language objects.
- Declarative transformation languages such as XSLT and XQuery.

Stream-oriented facilities require less memory and, for certain tasks which are based on a linear traversal of an XML document, are faster and simpler than other alternatives. Tree-

traversal and data-binding APIs typically require the use of much more memory, but are often found more convenient for use by programmers; some include declarative retrieval of document components via the use of XPath expressions.

XSLT is designed for declarative description of XML document transformations, and has been widely implemented both in server-side packages and Web browsers. XQuery overlaps XSLT in its functionality, but is designed more for searching of large XML databases.

Simple API for XML (SAX)

SAX is a lexical, event-driven interface in which a document is read serially and its contents are reported as callbacks to various methods on a handler object of the user's design. SAX is fast and efficient to implement, but difficult to use for extracting information at random from the XML, since it tends to burden the application author with keeping track of what part of the document is being processed. It is better suited to situations in which certain types of information are always handled the same way, no matter where they occur in the document.

Pull parsing

Pull parsing treats the document as a series of items which are read in sequence using the Iterator design pattern. This allows for writing of recursive-descent parsers in which the structure of the code performing the parsing mirrors the structure of the XML being parsed, and intermediate parsed results can be used and accessed as local variables within the methods performing the parsing, or passed down (as method parameters) into lower-level methods, or returned (as method return values) to higher-level methods. Examples of pull parsers include StAX in the Java programming language, XMLReader in PHP and System.Xml.XmlReader in the .NET Framework.

A pull parser creates an iterator that sequentially visits the various elements, attributes, and data in an XML document. Code which uses this iterator can test the current item (to tell, for example, whether it is a start or end element, or text), and inspect its attributes (local name, namespace, values of XML attributes, value of text, etc.), and can also move the iterator to the next item. The code can thus extract information from the document as it traverses it. The recursive-descent approach tends to lend itself to keeping data as typed local variables in the code doing the parsing, while SAX, for instance, typically requires a parser to manually maintain intermediate data within a stack of elements which are parent elements of the element being parsed. Pull-parsing code can be more straightforward to understand and maintain than SAX parsing code..

Document Object Model (DOM)

DOM (Document Object Model) is an interface-oriented Application Programming Interface that allows for navigation of the entire document as if it were a tree of "Node" objects representing the document's contents. A DOM document can be created by a

parser, or can be generated manually by users (with limitations). Data types in DOM Nodes are abstract; implementations provide their own programming language-specific bindings. DOM implementations tend to be memory intensive, as they generally require the entire document to be loaded into memory and constructed as a tree of objects before access is allowed.

Data binding

Another form of XML processing API is XML data binding, where XML data is made available as a hierarchy of custom, strongly typed classes, in contrast to the generic objects created by a Document Object Model parser. This approach simplifies code development, and in many cases allows problems to be identified at compile time rather than run-time. Example data binding systems include the Java Architecture for XML Binding (JAXB), XML Serialization in .NET, and CodeSynthesis XSD for C++.

XML as data type

XML is beginning to appear as a first-class data type in other languages. The ECMAScript for XML (E4X) extension to the ECMAScript/JavaScript language explicitly defines two specific objects (XML and XMLList) for JavaScript, which support XML document nodes and XML node lists as distinct objects and use a dot-notation specifying parent-child relationships. E4X is supported by the Mozilla 2.5+ browsers and Adobe Actionscript, but has not been adopted more universally. Similar notations are used in Microsoft's LINQ implementation for Microsoft .NET 3.5 and above, and in Scala (which uses the Java VM). The open-source xmlsh application, which provides a Linux-like shell with special features for XML manipulation, similarly treats XML as a data type, using the <[ ]> notation. The Resource Description Framework defines a data type `rdf:XMLLiteral` to hold wrapped, canonical XML.

History

XML is an application profile of SGML (ISO 8879).

The versatility of SGML for dynamic information display was understood by early digital media publishers in the late 1980s prior to the rise of the Internet. By the mid-1990s some practitioners of SGML had gained experience with the then-new World Wide Web, and believed that SGML offered solutions to some of the problems the Web was likely to face as it grew. Dan Connolly added SGML to the list of W3C's activities when he joined the staff in 1995; work began in mid-1996 when Sun Microsystems engineer Jon Bosak developed a charter and recruited collaborators. Bosak was well connected in the small community of people who had experience both in SGML and the Web.

XML was compiled by a working group of eleven members, supported by an (approximately) 150-member Interest Group. Technical debate took place on the Interest Group mailing list and issues were resolved by consensus or, when that failed, majority vote of the Working Group. A record of design decisions and their rationales was

compiled by Michael Sperberg-McQueen on December 4, 1997. James Clark served as Technical Lead of the Working Group, notably contributing the empty-element "<empty />" syntax and the name "XML". Other names that had been put forward for consideration included "MAGMA" (Minimal Architecture for Generalized Markup Applications), "SLIM" (Structured Language for Internet Markup) and "MGML" (Minimal Generalized Markup Language). The co-editors of the specification were originally Tim Bray and Michael Sperberg-McQueen. Halfway through the project Bray accepted a consulting engagement with Netscape, provoking vociferous protests from Microsoft. Bray was temporarily asked to resign the editorship. This led to intense dispute in the Working Group, eventually solved by the appointment of Microsoft's Jean Paoli as a third co-editor.

The XML Working Group never met face-to-face; the design was accomplished using a combination of email and weekly teleconferences. The major design decisions were reached in twenty weeks of intense work between July and November 1996, when the first Working Draft of an XML specification was published. Further design work continued through 1997, and XML 1.0 became a W3C Recommendation on February 10, 1998.

Sources

XML is a profile of an ISO standard SGML, and most of XML comes from SGML unchanged. From SGML comes the separation of logical and physical structures (elements and entities), the availability of grammar-based validation (DTDs), the separation of data and metadata (elements and attributes), mixed content, the separation of processing from representation (processing instructions), and the default angle-bracket syntax. Removed were the SGML Declaration (XML has a fixed delimiter set and adopts Unicode as the document character set).

Other sources of technology for XML were the Text Encoding Initiative (TEI), which defined a profile of SGML for use as a "transfer syntax"; and HTML, in which elements were synchronous with their resource, document character sets were separate from resource encoding, the xml:lang attribute was invented, and (like HTTP) metadata accompanied the resource rather than being needed at the declaration of a link. The Extended Reference Concrete Syntax (ERCS) project of the SPREAD (Standardization Project Regarding East Asian Documents) project of the ISO-related China/Japan/Korea Document Processing expert group was the basis of XML 1.0's naming rules; SPREAD also introduced hexadecimal numeric character references and the concept of references to make available all Unicode characters. To support ERCS, XML and HTML better, the SGML standard IS 8879 was revised in 1996 and 1998 with WebSGML Adaptations. The XML header followed that of ISO HyTime.

Ideas that developed during discussion which were novel in XML included the algorithm for encoding detection and the encoding header, the processing instruction target, the xml:space attribute, and the new close delimiter for empty-element tags. The notion of well-formedness as opposed to validity (which enables parsing without a schema) was

first formalized in XML, although it had been implemented successfully in the Electronic Book Technology "Dynatext" software; the software from the University of Waterloo New Oxford English Dictionary Project; the RISP LISP SGML text processor at Uniscope, Tokyo; the US Army Missile Command IADS hypertext system; Mentor Graphics Context; Interleaf and Xerox Publishing System.

Versions

There are two current versions of XML. The first (*XML 1.0*) was initially defined in 1998. It has undergone minor revisions since then, without being given a new version number, and is currently in its fifth edition, as published on November 26, 2008. It is widely implemented and still recommended for general use.

The second (*XML 1.1*) was initially published on February 4, 2004, the same day as XML 1.0 Third Edition, and is currently in its second edition, as published on August 16, 2006. It contains features (some contentious) that are intended to make XML easier to use in certain cases. The main changes are to enable the use of line-ending characters used on EBCDIC platforms, and the use of scripts and characters absent from Unicode 3.2. XML 1.1 is not very widely implemented and is recommended for use only by those who need its unique features.

Prior to its fifth edition release, XML 1.0 differed from XML 1.1 in having stricter requirements for characters available for use in element and attribute names and unique identifiers: in the first four editions of XML 1.0 the characters were exclusively enumerated using a specific version of the Unicode standard (Unicode 2.0 to Unicode 3.2.) The fifth edition substitutes the mechanism of XML 1.1, which is more future-proof but reduces redundancy. The approach taken in the fifth edition of XML 1.0 and in all editions of XML 1.1 is that only certain characters are forbidden in names, and everything else is allowed, in order to accommodate the use of suitable name characters in future versions of Unicode. In the fifth edition, XML names may contain characters in the Balinese, Cham, or Phoenician scripts among many others which have been added to Unicode since Unicode 3.2.

Almost any Unicode code point can be used in the character data and attribute values of an XML 1.0 or 1.1 document, even if the character corresponding to the code point is not defined in the current version of Unicode. In character data and attribute values, XML 1.1 allows the use of more control characters than XML 1.0, but, for "robustness", most of the control characters introduced in XML 1.1 must be expressed as numeric character references (and #x7F through #x9F, which had been allowed in XML 1.0, are in XML 1.1 even required to be expressed as numeric character references). Among the supported control characters in XML 1.1 are two line break codes that must be treated as whitespace. Whitespace characters are the only control codes that can be written directly.

There has been discussion of an XML 2.0, although no organization has announced plans for work on such a project. XML-SW (SW for skunkworks), written by one of the original developers of XML, contains some proposals for what an XML 2.0 might look

like: elimination of DTDs from syntax, integration of namespaces, XML Base and XML Information Set (*infoset*) into the base standard.

The World Wide Web Consortium also has an XML Binary Characterization Working Group doing preliminary research into use cases and properties for a binary encoding of the XML infoset. The working group is not chartered to produce any official standards. Since XML is by definition text-based, ITU-T and ISO are using the name *Fast Infoset* for their own binary infoset to avoid confusion.

Criticism

XML and its extensions have regularly been criticized for verbosity and complexity. Mapping the basic tree model of XML to type systems of programming languages or databases can be difficult, especially when XML is used for exchanging highly structured data between applications, which was not its primary design goal. Other criticisms attempt to refute the claim that XML is a self-describing language (though the XML specification itself makes no such claim). The most proposed alternatives include JSON and YAML — both focusing on representing structured data, rather than narrative documents.

Chapter 13

Base64

Base64 is a group of similar encoding schemes that represent binary data in an ASCII string format by translating it into a radix-64 representation. The Base64 term originates from a specific MIME content transfer encoding.

Base64 encoding schemes are commonly used when there is a need to encode binary data that needs be stored and transferred over media that are designed to deal with textual data. This is to ensure that the data remains intact without modification during transport. Base64 is used commonly in a number of applications including email via MIME, and storing complex data in XML.

Design

The particular choice of character set selected for the 64 characters required for the base varies between implementations. The general rule is to choose a set of 64 characters that is both part of a subset common to most encodings, and also printable. This combination leaves the data unlikely to be modified in transit through information systems, such as email, that were traditionally not 8-bit clean. For example, MIME's Base64 implementation uses A–Z, a–z, and 0–9 for the first 62 values. Other variations, usually derived from Base64, share this property but differ in the symbols chosen for the last two values; an example is UTF-7.

The earliest instances of this type of encoding were created for dialup communication between systems running the same OS - e.g. uuencode for UNIX, BinHex for the TRS-80 (later adapted for the Macintosh) - and could therefore make more assumptions about what characters were safe to use. For instance, uuencode uses uppercase letters, digits, and many punctuation characters, but no lowercase, since UNIX was sometimes used with terminals that did not support distinct letter case.

Examples

A quote from Thomas Hobbes' *Leviathan*:

Man is distinguished, not only by his reason, but by this singular passion from other animals, which is a lust of the mind, that by a perseverance of delight in the continued and indefatigable generation of knowledge, exceeds the short vehemence of any carnal pleasure.

represented as a byte sequence of 8-bit-padded ASCII characters is encoded in MIME's Base64 scheme as follows:

```
TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1d
CBieSB0aGlz
IHNpbmd1bGFyIHBhc3Npb24gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaCBpcyBhI
Gx1c3Qgb2Yg
dGhlIG1pbmQsIHRoYXQgYnkgYSBwZXJzZXZlcmFuY2Ugb2YgZGVsaWdodCBpbiB0a
GUgY29udGlu
dWVkIGFuZCBpbmRlZmF0aWdhYmxlIGdlbmVyYXRpb24gb2Yga25vd2xlZGdlLCBle
GNlZWRzIHRo
ZSBzaG9ydCB2ZWhlbWVuY2Ugb2YgYW55IGNhcm5hbCBwbGVhc3VyZS4=
```

In the above quote the encoded value of *Man* is *TWFu*. Encoded in ASCII, *M*, *a*, *n* are stored as the bytes `77`, `97`, `110`, which are, in 8-bit quantities, `01001101`, `01100001`, `01101110` in base 2. These three bytes are joined together into a 24 bit buffer producing `010011010110000101101110`. Packs of 6 bits (6 bits have a maximum of 64 different binary values) are converted into numbers (in this case, there are 4 numbers in this 24-bit string), which are then converted to their corresponding values in Base64.

Text content	**M**	**a**	**n**
ASCII	77	97	110
Bit pattern	0 1 0 0 1 1 0 1	0 1 1 0 0 0 0 1	0 1 1 0 1 1 1 0
Index	19	22	5	46
Base64-encoded	**T**	**W**	**F**	**u**

As this example illustrates, Base64 encoding converts 3 octets into 4 encoded characters.

The Base64 index table:

Value	**Char**	**Value**	**Char**	**Value**	**Char**	**Value**	**Char**
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3

8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

When the last group contains one octet the four least significant bits of the final 6-bit block are set to zero, and when the last group contains two octets the two least significant bits of the final 6-bit block are set to zero.

Padding

The '==' and '=' sequence indicate that the last group contained only 8 or 16 bits, respectively. The example below illustrates how truncating the input of the above quote changes the output padding:

```
Input ends with: carnal pleasure.  Output ends with: c3VyZS4=
Input ends with: carnal pleasure   Output ends with: c3VyZQ==
Input ends with: carnal pleasur    Output ends with: c3Vy
Input ends with: carnal pleasu     Output ends with: c3U=
```

The same characters will be encoded differently depending on their position within the three-octet group which is encoded to produce the four characters. For example

```
The Input: leasure.   Encodes to bGVhc3VyZS4=
The Input:  easure.   Encodes to ZWFzdXJlLg==
The Input:   asure.   Encodes to     YXN1cmUu
The Input:    sure.   Encodes to     c3VyZS4=
```

From a theoretical point of view the padding character is not needed, since the number of missing bytes can be calculated from the number of base64 digits. In some implementations the padding character is mandatory to use, for others it is not used.

The number of output bytes per input byte is approximately 4 / 3 and converges to that value for large number of bytes. More specifically, given an input of *n* bytes, the output will be `(n + 2 - ((n + 2) % 3)) / 3 * 4` bytes long, including padding characters.

Implementations and history

Variants summary table

Implementations may have some constraints on the alphabet used for representing some bit patterns. This notably concerns the last two characters used in the index table for

index 62 and 63, and the character used for padding (which may be mandatory in some protocols, or removed in others). The table below summarizes these known variants, and link to the subsections below.

Variant	Char for index 62	Char for index 63	*pad* char	Fixed encoded line-length	Maximum encoded line length	Line separators	Characters outside alphabet	Line checksum
Original Base64 for Privacy-Enhanced Mail (PEM) (RFC 1421, deprecated)	+	/	= *(mandatory)*	Yes (except last line)	64	CR+LF	Forbidden	*(none)*
Base64 transfer encoding for MIME (RFC 2045)	+	/	= *(mandatory)*	No (variable)	76	CR+LF	Accepted (discarded)	*(none)*
Standard 'Base64' encoding for RFC 3548 or RFC 4648	+	/	= *(mandatory)*	Yes (except last line)	64 or 76 *(only if line separators are specified and needed)*	CR+LF *(only if specified and needed)*	Forbidden	*(none)*
'Radix-64' encoding for OpenPGP (RFC 4880)	+	/	= *(mandatory)*	No (variable)	76	CR+LF	Forbidden	24-bit CRC (Radix-64-encoded, including one *pad* character)
Modified Base64 encoding for UTF-7 (RFC 1642, obsoleted)	+	/	*(none)*	No (variable)	*(none)*	*(none)*	Forbidden	*(none)*
Modified Base64 for filenames (non standard)	+	-	*(none)*	No (variable)	*(filesystem limit, generally 255)*	*(none)*	Forbidden	*(none)*
Modified Base64 for URL applications ('base64url' encoding)	-	_	*(none)*	No (variable)	*(application-dependent)*	*(none)*	Forbidden	*(none)*
Modified Base64 for XML name tokens (*Nmtoken*)	.	-	*(none)*	No (variable)	*(XML parser-dependent)*	*(none)*	Forbidden	*(none)*
Modified Base64 for XML identifiers (*Name*)	–	:	*(none)*	No (variable)	*(XML parser-dependent)*	*(none)*	Forbidden	*(none)*
Modified Base64 for Program identifiers (variant 1, non standard)	_	-	*(none)*	No (variable)	*(language/system-dependent)*	*(none)*	Forbidden	*(none)*
Modified Base64 for Program identifiers (variant 2, non standard)	.	_	*(none)*	No (variable)	*(language/system-dependent)*	*(none)*	Forbidden	*(none)*
Modified Base64 for Regular expressions (non standard)	!	-	*(none)*	No (variable)	*(application-dependent)*	*(none)*	Forbidden	*(none)*

Privacy-enhanced mail

The first known standardized use of the encoding now called MIME Base64 was in the Privacy-enhanced Electronic Mail (PEM) protocol, proposed by RFC 989 in 1987. PEM defines a "printable encoding" scheme that uses Base64 encoding to transform an arbitrary sequence of octets to a format that can be expressed in short lines of 6-bit characters, as required by transfer protocols such as SMTP.

The current version of PEM (specified in RFC 1421) uses a 64-character alphabet consisting of upper- and lower-case Roman alphabet characters (A–Z, a–z), the numerals (0–9), and the "+" and "/" symbols. The "=" symbol is also used as a special suffix code. The original specification, RFC 989, additionally used the "*" symbol to delimit encoded but unencrypted data within the output stream.

To convert data to PEM printable encoding, the first byte is placed in the most significant eight bits of a 24-bit buffer, the next in the middle eight, and the third in the least significant eight bits. If there are fewer than three bytes left to encode (or in total), the remaining buffer bits will be zero. The buffer is then used, six bits at a time, most significant first, as indices into the string: `"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"`, and the indicated character is output.

The process is repeated on the remaining data until fewer than four octets remain. If three octets remain, they are processed normally. If fewer than three octets (24 bits) are remaining to encode, the input data is right-padded with zero bits to form an integral multiple of six bits.

After encoding the non-padded data, if two octets of the 24-bit buffer are padded-zeros, two "=" characters are appended to the output; if one octet of the 24-bit buffer is filled with padded-zeros, one "=" character is appended. This signals the decoder that the zero bits added due to padding should be excluded from the reconstructed data. This also guarantees that the encoded output length is a multiple of 4 bytes.

PEM requires that all encoded lines consist of exactly 64 printable characters, with the exception of the last line, which may contain fewer printable characters. Lines are delimited by whitespace characters according to local (platform-specific) conventions.

MIME

The MIME (Multipurpose Internet Mail Extensions) specification, lists base64 as one of two binary-to-text encoding schemes (the other being quoted-printable). MIME's Base64 encoding is based on that of the RFC 1421 version of PEM: it uses the same 64-character alphabet and encoding mechanism as PEM, and uses the "=" symbol for output padding in the same way, as described at RFC 1521.

MIME does not specify a fixed length for Base64-encoded lines, but it does specify a maximum line length of 76 characters. Additionally it specifies that any extra-alphabetic characters must be ignored by a compliant decoder, although most implementations use a `CR/LF` newline pair to delimit encoded lines.

Thus, the actual length of MIME-compliant Base64-encoded binary data is usually about 137% of the original data length, though for very short messages the overhead can be a lot higher because of the overhead of the headers. Very roughly, the final size of Base64-encoded binary data is equal to 1.37 times the original data size + 814 bytes (for headers). In other words, you can approximate the size of the decoded data with this formula: bytes = (string_length(encoded_string) - 814) / 1.37

UTF-7

UTF-7, described first in RFC 1642, which was later superseded by RFC 2152, introduced a system called *modified Base64*. This data encoding scheme is used to encode UTF-16 as ASCII characters for use in 7-bit transports such as SMTP. It is a variant of the base64 encoding used in MIME.

The "Modified Base64" alphabet consists of the MIME Base64 alphabet, but does not use the "=" padding character. UTF-7 is intended for use in mail headers (defined in RFC 2047), and the "=" character is reserved in that context as the escape character for "quoted-printable" encoding. Modified base64 simply omits the padding and ends immediately after the last Base64 digit containing useful bits (leaving 0-3 unused bits in the last Base64 digit).

OpenPGP

OpenPGP, described in RFC 4880, describes **Radix-64** encoding, also known as "ASCII Armor". Radix-64 is identical to the "Base64" encoding described from MIME, with the addition of an optional 24-bit CRC checksum. The checksum is calculated on the input data before encoding; the checksum is then encoded with the same base64 algorithm and, using an additional "=" symbol as separator, appended to the encoded output data.

RFC 3548

RFC 3548, entitled *The Base16, Base32, and Base64 Data Encodings*, is an informational (non-normative) memo that attempts to unify the RFC 1421 and RFC 2045 specifications of Base64 encodings, alternative-alphabet encodings, and the seldom-used Base32 and Base16 encodings.

RFC 3548 forbids implementations from generating messages containing characters outside the encoding alphabet or without padding, unless they are written to a specification that refers to RFC 3548 and specifically requires otherwise; it also declares that decoder implementations must reject data that contains characters outside the

encoding alphabet, unless they are written to a specification that refers to RFC 3548 and specifically requires otherwise.

RFC 4648

This RFC obsoletes RFC 3548 and focuses on Base64/32/16:

This document describes the commonly used Base64, Base32, and Base16 encoding schemes. It also discusses the use of line-feeds in encoded data, use of padding in encoded data, use of non-alphabet characters in encoded data, use of different encoding alphabets, and canonical encodings.

Filenames

Another variant called **modified Base64 for filename** uses '-' instead of '/', because Unix and Windows filenames cannot contain '/'.

URL applications

Base64 encoding can be helpful when fairly lengthy identifying information is used in an HTTP environment. For example, a database persistence framework for Java objects might use Base64 encoding to encode a relatively large unique id (generally 128-bit UUIDs) into a string for use as an HTTP parameter in HTTP forms or HTTP GET URLs. Also, many applications need to encode binary data in a way that is convenient for inclusion in URLs, including in hidden web form fields, and Base64 is a convenient encoding to render them in not only a compact way, but in a relatively unreadable one when trying to obscure the nature of data from a casual human observer.

Using standard Base64 in URL requires encoding of '+' and '/' characters into special percent-encoded hexadecimal sequences ('+' = '%2B' and '/' = '%2F'), which makes the string unnecessarily longer.

For this reason, a **modified Base64 for URL** variant exists, where *no* padding '=' will be used, and the '+' and '/' characters of standard Base64 are respectively replaced by '-' and '_', so that using URL encoders/decoders are no longer necessary and have no impact on the length of the encoded value, leaving the same encoded form intact for use in relational databases, web forms, and object identifiers in general.

Program identifiers

There are other variants that use '_-' or '._' when the Base64 variant string must be used within valid identifiers for programs.

XML

XML identifiers and name tokens are encoded using two variants:

- '.-' for use in XML name tokens (*Nmtoken*), or even
- '_:' for use in more restricted XML identifiers (*Name*).

Regular expressions

Another variant called **modified Base64 for regexps** uses '!-' instead of '*-' to replace the standard Base64 '+/', because both '+' and '*' may be reserved for regular expressions (note that '[]' used in the IRCu variant above would not work in that context).

HTML

The `atob()` and `btoa()` JavaScript methods, defined in the HTML5 draft specification, provide base64 encoding and decoding functionality to web pages. The *atob* method is unusual in that it does not ignore whitespace or new lines, throwing an INVALID_CHARACTER_ERR instead. The *btoa* method outputs padding characters, but these are optional in the input of the *atob* method.

Other applications

Base64 can be used in a variety of contexts:

- Evolution and Thunderbird use Base64 to obfuscate e-mail passwords
- Base64 can be used to transmit and store text that might otherwise cause delimiter collision
- Base64 is often used as a quick but insecure shortcut to obscure secrets without incurring the overhead of cryptographic key management
- Base64 is used to store a password hash computed with crypt in the `/etc/passwd`
- Spammers use Base64 to evade basic anti-spamming tools, which often do not decode Base64 and therefore cannot detect keywords in encoded messages.
- Base64 is used to encode character strings in LDIF files
- Base64 is often used to embed binary data in an XML file, using a syntax similar to `<data encoding="base64">…</data>` e.g. favicons in Firefox's `bookmarks.html`.
- Base64 is used to encode binary files such as images within scripts, to avoid depending on external files.
- The data URI scheme can use Base64 to represent file contents. For instance, background images can be specified in a CSS stylesheet file as `data:` URIs, instead of being supplied in separate image files.

Chapter 14

Comma-Separated Values

Comma-separated values
Comma separated list

Filename extension	`.csv or .txt`
Internet media type	`text/csv`
Type of format	multiplatform, serial data streams
Container for	database information organized as field separated lists
Standard(s)	RFC 4180

The **comma-separated values** file format is a set of file formats used to store tabular data in which numbers and text are stored in plain textual form that can be read in a text editor. Lines in the text file represent rows of a table, and commas in a line separate what are fields in the tables row. Different implementations of CSV arise as the format is modified to handle richer table content such as allowing a different field separator character (necessary if numeric fields are written with a comma instead of a decimal

point) or extensions to allow numbers, the separator character, or newline characters in text fields.

CSV is a simple file format that is widely supported, so it is often used to move tabular data between different computer programs that support compatible CSV formats. For example: a CSV file might be used to transfer information from a database program to a spreadsheet.

Example of a USA/UK CSV file (where the decimal separator is a period/full stop and the value separator is a comma):

```
Year,Make,Model,Length
1997,Ford,E350,2.34
2000,Mercury,Cougar,2.38
```

Example of a German and Dutch CSV/SSV file (where the decimal separator is a comma and the value separator is a semicolon):

```
Year;Make;Model;Length
1997;Ford;E350;2,34
2000;Mercury;Cougar;2,38
```

Technical background

The format dates back to the early days of business computing and is widely used to pass data between computers with different internal word sizes, data formatting needs, and so forth. For this reason, CSV files are common on all computer platforms.

CSV is a delimited text file that uses a comma to separate values (many implementations of CSV import/export tools allow other separators to be used). Simple CSV implementations will not allow field values that contain a comma or other special characters such as newlines. More sophisticated CSV implementations permit commas and other special characters in a field value. Many implementations use " (double quote) characters around values that contain reserved characters (such as commas, double quotes, or newlines); embedded double quote characters may be represented by a pair of consecutive double quotes. (Creativyst 2010) Some CSV implementations may use an escape character such as a backslash to encode reserved characters as an escape sequence.

In computer science terms, a CSV file is a "flat file".

History

Comma-separated values are old technology and pre-date personal computers by more than a decade: the IBM Fortran (level G) compiler under OS/360 supported them in 1967. Comma-separated value lists were often easier to type into punched cards than fixed-column-aligned data, and were less prone to producing incorrect results if a value was punched one column off from its intended location.

The **comma separated list (CSL)** is a data format originally known as **comma-separated values (CSV)** in the oldest days of simple computers. In the industry of personal computers (then more commonly known as "Home Computers"), the most common use was small businesses generating solicitations using boilerplate form letters and mailing lists.

Some early software applications, such as word processors, allowed a stream of "variable data" to be merged between two files: a form letter, and a CSL of names, addresses, and other data fields. Many applications still do, simply because tasks requiring human input (such as constructing a list) are natural and easy using comma delimiters. CSL/CSVs were also used for simple databases.

Specification

Background

Comma separated lists date from before the earliest personal computers, but were widely used in the earliest pre-IBM PC era personal computers for tape storage backup and interchange of database information from machines of two different architectures. In that day, affordable hard drives did not exist, and many small businesses tried to achieve the benefits of computing using floppy disk based software.

No general standard specification for CSV exists. Variations between CSV implementations in different programs are quite common and can lead to interoperation difficulties. For Internet communication of CSV files, an Informational IETF document (RFC 4180 from October 2005) describes the format for the "text/csv" MIME type registered with the IANA. (Shafranovich 2005) Another relevant specification is provided by Fielded Text which also covers the CSV format.

Many informal documents exist that describe the CSV format. Creativyst (2010) provides an overview of the CSV format in the most widely used applications and explains how it can best be used and supported.

Basic rules

The basic rules from a lot of these specifications are as follows:

CSV is a delimited data format that has fields/columns separated by the comma character and records/rows terminated by newlines. Fields that contain a special character (comma, newline, or double quote), must be enclosed in double quotes. If a line contains a single entry which is the empty string, it may be enclosed in double quotes. If a field's value contains a double quote character it is escaped by placing another double quote character next to it. The CSV file format does not require a specific character encoding, byte order, or line terminator format.

Note: While binary data is not prohibited, it is especially problematic to incorporate as reserved CSV characters (comma, newline, double-quote) are often present in binary data, and are not typically 'escaped' or otherwise correctly preprocessed. The tradition has been that CSV file data is **humanly readable as text**, so that binary numbers are **converted** to ASCII string format before collation in the file. Example: binary (as hexadecimal) 0x3FFF (two bytes, one of value 63 followed by another of value 255) would be represented in ASCII as 16383.

- Each record is one line terminated by a line feed (ASCII/LF=0x0A) or a carriage return and line feed pair (ASCII/CRLF=0x0D 0x0A), however, line-breaks can be embedded.
- Fields are separated by commas (although in locales where the comma is used as a decimal separator, the semicolon is used instead as a delimiter, inducing some drawbacks when CSV files are exchanged e.g. between France and USA)

```
1997,Ford,E350
```

- In some CSV implementations, leading and trailing spaces or tabs, adjacent to commas, are trimmed. *This practice is contentious and in fact is specifically prohibited by RFC 4180, which states, "Spaces are considered part of a field and should not be ignored."*

```
1997, Ford , E350
not same as
1997,Ford,E350
```

- Fields with embedded commas must be enclosed within double-quote characters.

```
1997,Ford,E350,"Super, luxurious truck"
```

- Fields with embedded double-quote characters must be enclosed within double-quote characters, and each of the embedded double-quote characters must be represented by a pair of double-quote characters.

```
1997,Ford,E350,"Super, ""luxurious"" truck"
```

- Fields with embedded line breaks must be enclosed within double-quote characters.

```
1997,Ford,E350,"Go get one now
they are going fast"
```

- In CSV implementations that trim leading or trailing spaces, fields with such spaces must be enclosed within double-quote characters.

```
1997,Ford,E350," Super luxurious truck "
```

- Fields may always be enclosed within double-quote characters, whether necessary or not.

```
"1997","Ford","E350"
```

- The first record in a csv file may contain column names in each of the fields.

```
Year,Make,Model
1997,Ford,E350
2000,Mercury,Cougar
```

Example

Year	Make	Model	Description	Price
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture "Extended Edition"		4900.00
1999	Chevy	Venture "Extended Edition, Very Large"		5000.00
1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

The above table of data may be represented in CSV format as follows:

```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""","",5000.00
1996,Jeep,Grand Cherokee,"MUST SELL!
air, moon roof, loaded",4799.00
```

This CSV example illustrates that:

- fields that contain commas, double-quotes, or line-breaks must be quoted.
- a quote within a field must be escaped with an additional quote immediately preceding the literal quote.
- space before and after delimiter commas **may not** be trimmed. This is required by RFC 4180.
- a line break within an element must be preserved.

Application support

The CSV file format is very simple and supported by almost all spreadsheets and database management systems. Many programming languages have libraries available that support CSV files. Even modern software applications support CSV imports and/or exports because the format is so widely recognized. In fact, many applications allow .csv-named files to use any delimiter character.

Microsoft Excel will open .csv files, but depending on the system's regional settings, it may expect a semicolon as a separator instead of a comma, since in some languages the

comma is used as the decimal separator. Also, many regional versions of Excel will not be able to deal with Unicode in CSV. One simple solution when encountering such difficulties is to change the filename extension from .csv to .txt; then opening the file from an already running Excel with the "Open" command.

When pasting text data into Excel, the tab character is used as a separator: If you copy "hello<tab>goodbye" into the clipboard and paste it into Excel, it goes into two cells. "hello,goodbye" pasted into Excel goes into one cell, including the comma.

Chapter 15

JSON

	JSON
Filename extension	`.json`
Internet media type	`application/json`
Type of format	Data interchange
Extended from	JavaScript
Standard(s)	RFC 4627

JSON is a lightweight text-based open standard designed for human-readable data interchange. It is derived from the JavaScript programming language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for most programming languages.

The JSON format was originally specified by Douglas Crockford, and is described in RFC 4627. The official Internet media type for JSON is `application/json`. The JSON filename extension is `.json`.

The JSON format is often used for serializing and transmitting structured data over a network connection. It is primarily used to transmit data between a server and web application, serving as an alternative to XML.

History

Douglas Crockford was the first to specify and popularize the JSON format.

JSON was used at State Software, a company co-founded by Crockford, starting around 2001. The JSON.org website was launched in 2002. In December 2005, Yahoo! began offering some of its web services in JSON. Google started offering JSON feeds for its GData web protocol in December 2006.

Although JSON was based on a subset of the JavaScript programming language (specifically, Standard ECMA-262 3rd Edition—December 1999) and is commonly used with that language, it is considered to be a language-independent data format. Code for parsing and generating JSON data is readily available for a large variety of programming languages. json.org provides a comprehensive listing of existing JSON libraries, organized by language.

Data types, syntax and example

JSON's basic types are:

- Number (double precision floating-point format)
- String (double-quoted Unicode with backslash escaping)
- Boolean (`true` or `false`)
- Array (an ordered sequence of values, comma-separated and enclosed in square brackets)
- Object (a collection of key:value pairs, comma-separated and enclosed in curly braces; the key must be a string)
- `null`

The following example shows the JSON representation of an object that describes a person. The object has string fields for first name and last name, a number field for age, contains an object representing the person's address, and contains a list (an array) of phone number objects.

```
{
     "firstName": "John",
     "lastName": "Smith",
     "age": 25,
     "address":
     {
         "streetAddress": "21 2nd Street",
         "city": "New York",
         "state": "NY",
         "postalCode": "10021"
     },
     "phoneNumber":
     [
         {
           "type": "home",
           "number": "212 555-1234"
         },
         {
           "type": "fax",
           "number": "646 555-4567"
         }
     ]
 }
```

Since JSON is a subset of JavaScript it is possible (but not recommended) to parse the JSON text into an object by invoking JavaScript's `eval()` function. For example, if the above JSON data is contained within a JavaScript string variable *contact*, one could use it to create the JavaScript object *p* like so:

```
var p = eval("(" + contact + ")");
```

The `contact` variable must be wrapped in parentheses to avoid an ambiguity in JavaScript's syntax.

The recommended way, however, is to use a JSON parser. Unless a client absolutely trusts the source of the text, or must parse and accept text which is not strictly JSON-compliant, one should avoid `eval()`. A correctly implemented JSON parser will accept only valid JSON, preventing potentially malicious code from running.

Modern browsers, such as Firefox 3.5 and Internet Explorer 8, include special features for parsing JSON. As native browser support is more efficient and secure than `eval()`, it is expected that native JSON support will be included in the next ECMAScript standard.

Schema

There are several ways to verify the structure and data types inside a JSON object, much like an XML schema.

JSON Schema is a specification for a JSON-based format for defining the structure of JSON data. JSON Schema provides a contract for what JSON data is required for a given application and how it can be modified, much like what XML Schema provides for XML. JSON Schema is intended to provide validation, documentation, and interaction control of JSON data. JSON Schema is based on the concepts from XML Schema, RelaxNG, and Kwalify, but is intended to be JSON-based, so that JSON data in the form of a schema can be used to validate JSON data, the same serialization/deserialization tools can be used for the schema and data, and it can be self descriptive.

JSON Schema is still an IETF draft, but there are several validators currently available for different programming languages, each with varying levels of conformance. Currently the most complete and compliant JSON Schema validator available is JSV.

Example JSON Schema:

```
{
        "name":"Product",
        "properties":
        {
                "id":
                {
                        "type":"number",
                        "description":"Product identifier",
                        "required":true
```

```
                },
                "name":
                {
                        "description":"Name of the product",
                        "type":"string",
                        "required":true
                },
                "price":
                {
                        "type":"number",
                        "minimum":0,
                        "required":true
                },
                "tags":
                {
                        "type":"array",
                        "items":
                        {
                                "type":"string"
                        }
                }
        }
}
```

The JSON Schema above can be used to test the validity of the JSON code below:

```
{
        "id": 1,
        "name": "Foo",
        "price": 123,
        "tags": ["Bar","Eek"]
}
```

MIME type

The MIME type for JSON text is "`application/json`".

Use in Ajax

The following JavaScript code shows how the client can use an XMLHttpRequest to request an object in JSON format from the server. (The server-side programming is omitted; it has to be set up to respond to requests at `url` with a JSON-formatted string.)

```
var my_JSON_object = {};
var http_request = new XMLHttpRequest();
http_request.open( "GET", url, true );
http_request.onreadystatechange = function () {
  if (http_request.readyState == 4 && http_request.status == 200){
       my_JSON_object = JSON.parse( http_request.responseText );
  }
};
http_request.send(null);
```

Note that the use of XMLHttpRequest in this example is not cross-browser compatible; syntactic variations are available for Internet Explorer, Opera, Safari, and Mozilla-based browsers. The usefulness of XMLHttpRequest is limited by the same origin policy: the URL replying to the request must reside within the same DNS domain as the server that hosts the page containing the request. Alternatively, the JSONP approach incorporates the use of an encoded callback function passed between the client and server to allow the client to load JSON-encoded data from third-party domains and to notify the caller function upon completion, although this imposes some security risks and additional requirements upon the server.

Browsers can also use `<iframe>` elements to asynchronously request JSON data in a cross-browser fashion, or use simple `<form action="url_to_cgi_script" target="name_of_hidden_iframe">` submissions. These approaches were prevalent prior to the advent of widespread support for XMLHttpRequest.

Dynamic `<script>` tags can also be used to transport JSON data. With this technique it is possible to get around the same origin policy but it is insecure. JSONRequest has been proposed as a safer alternative.

Security issues

Although JSON is intended as a data serialization format, its design as a subset of the JavaScript programming language poses several security concerns. These concerns center on the use of a JavaScript interpreter to execute JSON text dynamically as JavaScript, thus exposing a program to errant or malicious script contained therein—often a chief concern when dealing with data retrieved from the Internet. While not the only way to process JSON, it is an easy and popular technique, stemming from JSON's compatibility with JavaScript's eval() function, and illustrated by the following code examples.

JavaScript `eval()`

Because all JSON-formatted text is also syntactically legal JavaScript code, an easy way for a JavaScript program to parse JSON-formatted data is to use the built-in JavaScript `eval()` function, which was designed to evaluate JavaScript expressions. Rather than using a JSON-specific parser, the JavaScript interpreter itself is used to *execute* the JSON data to produce native JavaScript objects.

Unless precautions are taken to validate the data first, the eval technique is subject to security vulnerabilities if the data and the entire JavaScript environment is not within the control of a single trusted source. If the data is itself not trusted, for example, it may be subject to malicious JavaScript code injection attacks. Also, such breaches of trust may create vulnerabilities for data theft, authentication forgery, and other potential misuse of data and resources. Regular expressions can be used to validate the data prior to invoking `eval()`. For example, the RFC that defines JSON (RFC 4627) suggests using the following code to validate JSON before eval'ing it (the variable 'text' is the input JSON):

```
var my_JSON_object = !(/[^,:{}\[\]0-9.\-+Eaeflnr-u \n\r\t]/.test(
       text.replace(/"(\\.|[^"\\])*"/g, ''))) &&
   eval('(' + text + ')');
```

A new function, `JSON.parse()`, was developed as a safer alternative to `eval`. It is specifically intended to process JSON data and not JavaScript. It was originally planned for inclusion in the Fourth Edition of the ECMAScript standard, but this did not occur. It was first added to the Fifth Edition, and is now supported by the major browsers given below. For older ones, a compatible JavaScript library is available at JSON.org.

Native encoding and decoding in browsers

Recent Web browsers now either have or are working on native JSON encoding/decoding. This removes the `eval()` security problem above and also makes it faster because it doesn't parse functions. Native JSON is generally faster compared to the JavaScript libraries commonly used before. As of June 2009 the following browsers have or will have native JSON support:

- Mozilla Firefox 3.5+
- Microsoft Internet Explorer 8
- Opera 10.5+
- Webkit-based browsers (e.g. Google Chrome, Apple Safari)

At least 5 popular JavaScript libraries have committed to use native JSON if available:

- Yahoo! UI Library
- Prototype
- jQuery
- Dojo Toolkit
- mootools

Comparison with other formats

There are other lightweight markup languages that could be used to carry or store the same information payloads as JSON commonly does. Apart from XML, examples could include OGDL, YAML, CSV and HTML. JSON is promoted as a low-overhead alternative to XML as both of these formats have widespread support for creation, reading and decoding in the real-world situations where they are commonly used.

XML

XML is often used to describe structured data and to serialize objects. Various XML-based protocols exist to represent the same kind of data structures as JSON for the same kind of data interchange purposes. When data is encoded in XML, the result is typically larger in size than an equivalent encoding in JSON, mainly because of XML's closing tags. However, there are alternative ways to encode the same information. Each of the

following XML examples carry the same information as the JSON example above in different ways.

```
<Object>
  <Property><Key>firstName</Key>     <String>John</String></Property>
  <Property><Key>lastName</Key>      <String>Smith</String></Property>
  <Property><Key>age</Key>           <Number>25</Number></Property>
  <Property><Key>address</Key>
    <Object>
      <Property><Key>streetAddress</Key> <String>21 2nd
Street</String></Property>
      <Property><Key>city</Key>          <String>New
York</String></Property>
      <Property><Key>state</Key>         <String>NY</String></Property>
      <Property><Key>postalCode</Key>
<String>10021</String></Property>
    </Object>
  </Property>
  <Property><Key>phoneNumber</Key>
    <Array>
      <Object>
        <Property><Key>type</Key>
<String>home</String></Property>
        <Property><Key>number</Key>        <String>212 555-
1234</String></Property>
      </Object>
      <Object>
        <Property><Key>type</Key>
<String>fax</String></Property>
        <Property><Key>number</Key>        <String>646 555-
4567</String></Property>
      </Object>
    </Array>
  </Property>
</Object>
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumber type="home">212 555-1234</phoneNumber>
  <phoneNumber type="fax">646 555-4567</phoneNumber>
</person>
<person firstName="John" lastName="Smith" age="25">
  <address streetAddress="21 2nd Street" city="New York" state="NY"
postalCode="10021" />
  <phoneNumber type="home" number="212 555-1234"/>
  <phoneNumber type="fax"  number="646 555-4567"/>
</person>
```

The XML encoding *may* therefore be shorter than the equivalent JSON encoding. A wide range of XML processing technologies exist, from the Document Object Model to XPath and XSLT. XML can also be styled for immediate display using CSS. XHTML is a form of XML so that elements can be passed in this form ready for direct insertion into webpages using client-side scripting.

Efficiency

JSON is primarily used for communicating data over the Internet, but has certain inherent characteristics that may limit its efficiency for this purpose. Most of the limitations are general limitations of textual data formats and also apply to XML and YAML. For example, despite typically being generated by an algorithm (by machine), parsing must be accomplished on a character-by-character basis. Additionally, the standard has no provision for data compression, interning of strings, or object references. Compression can, of course, be applied to the JSON formatted data (but the decompressed output typically still requires further full parsing for recognizable keywords, tags and delimiters).

Object references

The JSON standard does not support object references, but the Dojo Toolkit illustrates how conventions can be adopted to support such references using standard JSON. Specifically, the dojox.json.ref module provides support for several forms of referencing including circular, multiple, inter-message, and lazy referencing.

Chapter 16

JsonML

	JSON Markup Language
Internet media type	`application/jsonml+json (unofficial)`
Type of format	Markup language and Web template system
Extended from	XML, JSON and JavaScript

JsonML, the **JSON Markup Language** is a lightweight markup language used to map between XML (Extensible Markup Language) and JSON (JavaScript Object Notation). It converts an XML document or fragment into a JSON data structure for ease of use within JavaScript environments such as a web browser, allowing manipulation of XML data without the overhead of an XML parser.

JsonML has greatest applicability in Ajax (Asynchronous JavaScript and XML) web applications. It is used to transport XHTML (eXtensible HyperText Markup Language) down to the client where it can be deterministically reconstructed into DOM (Document Object Model) elements. Progressive enhancement strategy can be employed during construction to bind dynamic behaviors to otherwise static elements.

JsonML can also be used as the underlying structure for creating intricate client-side templates called JBST (JsonML+Browser-Side Templates). Syntactically JBST looks like JSP (JavaServer Pages) or ASP.NET (Active Server Pages .NET) user controls. Interactive examples are available on the jsonml.org website.

Syntax

There are two forms of JsonML, *array form* and *object form*. Conversion from XML to JsonML is fully reversible. XML Namespaces are handled by prepending the element

name with the namespace prefix, e.g., `<myns:myElement/>` becomes `["myns:myElement"]`.

JsonML array form

The array form is the original form, and allows any XML document to be represented uniquely as a JSON string. JsonML array form uses:

- JSON arrays to represent XML elements;
- JSON objects to represent attributes;
- JSON strings to represent text nodes.

JsonML (array form)	Original XML

```
["person",                <!-- XML representation of a person
  {"created":"2006-11-  record -->
11T19:23",                <person created="2006-11-11T19:23"
   "modified":"2006-12- modified="2006-12-31T23:59">
31T23:59"},                   <firstName>Robert</firstName>
  ["firstName",               <lastName>Smith</lastName>
"Robert"],                    <address type="home">
  ["lastName",                    <street>12345 Sixth Ave</street>
"Smith"],                       <city>Anytown</city>
  ["address",                   <state>CA</state>
{"type":"home"},                <postalCode>98765-
    ["street", "12345   4321</postalCode>
Sixth Ave"],                  </address>
    ["city",              </person>
"Anytown"],
    ["state", "CA"],
    ["postalCode",
"98765-4321"]
  ]
]
```

JsonML object form

The object form assumes that there are no XML attributes named *tagName* or *childNodes*. JsonML Object Form uses:

- JSON arrays to represent child node lists;
- JSON objects to represent XML elements *and* their attributes.

The JsonML object form representation of the above coding is:

JsonML (object form)	Original XML

```
{"tagName": "person",       <!-- XML representation of a person
 "created": "2006-11-       record -->
11T19:23",                  <person created="2006-11-11T19:23"
 "modified": "2006-12-      modified="2006-12-31T23:59">
31T23:59",                      <firstName>Robert</firstName>
```

```
 "childNodes": [
    {"tagName":
"firstName", "childNodes"
: ["Robert"]},
    {"tagName":
"lastName", "childNodes" :
["Smith"]},
    {"tagName": "address",
"type":"home",
"childNodes" : [
        {"tagName":
"street", "childNodes" :
["12345 Sixth Ave"]},
        {"tagName":
"city", "childNodes" :
["Anytown"]},
        {"tagName":
"state", "childNodes" :
["CA"]},
        {"tagName":
"postalCode", "childNodes"
: ["98765-4321"]},
    ]}
 ]}
```

```
<lastName>Smith</lastName>
<address type="home">
    <street>12345 Sixth
Ave</street>
    <city>Anytown</city>
    <state>CA</state>
    <postalCode>98765-
4321</postalCode>
</address>
</person>
```

A “regular” JSON transformation produces a more compact representation, but loses some of the document structural information:

```
{"person": {
    "address": {
        "city": "Anytown",
        "postalCode": "98765-4321",
        "state": "CA",
        "street": "12345 Sixth Ave",
        "type": "home"
    },
    "created": "2006-11-11T19:23",
    "firstName": "Robert",
    "lastName": "Smith",
    "modified": "2006-12-31T23:59"
}}
```

Comparison to similar technologies

XML/XSLT

XML and XSLT (Extensible Stylesheet Language Transformations) can also produce client-side templating, and both allow caching of the template separate from the data. Many programmers however find the syntax of JBST is easier to manage due to its familiarity. JBST uses JavaScript natively in the template, rather than requiring mixing of different types of control language.

InnerHTML

While seemingly used to perform similar tasks, JsonML and innerHTML are quite different. InnerHTML requires all the markup in an exact form, meaning that either the server is rendering the markup, or the programmer is performing expensive string concatenations in JavaScript.

JsonML uses client-side templating through JBST, which means that HTML is converted into a JavaScript template at build time. At runtime, the data is supplied and DOM elements are the result. The resulting DOM elements can be inserted or replace an existing element, which innerHTML cannot easily do without creating excess DOM elements. Rebinding only requires requesting additional data, which is smaller than fully-expanded markup. As a result, large performance gains are often made, since the markup is requested or cached separately from the data.

HTML message pattern/Browser-side templating

For simplicity, innerHTML has been the preferred method for the HTML-Message pattern style of Ajax. However, tools like JsonFx aim to simplify JsonML and JBST implementation while still providing a full browser-side templating Ajax pattern.