1. Models of computation

jn 2004.05.03

Goals: An intuitive appreciation of the importance of the concept ‘model of computation’. Acquaintance with several interesting examples that mirror key aspects of realistic systems in the simplest possible setting.

1.0 Models of computation: purpose and types

Almost any plausible statement about computation is true in some model, and false in others!
A rigorous definition of a model of computation is essential when proving negative results: “impossible...”.
Special purpose models vs. universal models of computation (can simulate any other model).

Algorithm and *computability* are originally intuitive concepts. They can remain intuitive as long as we only want to show that some specific result can be computed by following a specific algorithm. Almost always an informal explanation suffices to convince someone with the requisite background that a given algorithm computes a specified result. Everything changes if we wish to show that a desired result is *not computable*. The question arises immediately: "What tools are we allowed to use?" Everything is computable with the help of an oracle that knows the answers to all questions. The attempt to prove negative results about the nonexistence of certain algorithms forces us to agree on a rigorous definition of *algorithm*.

Mathematics has long investigated problems of the type: what kind of objects can be constructed, what kind of results can be computed using a **limited set of primitives and operations**. For example, the question of what polynomial equations can be solved using radicals, i.e. using addition, subtraction, multiplication, division, and the extraction of roots, has kept mathematicians busy for centuries. It was solved by Niels Henrik Abel (1802 - 1829) in 1826: The roots of polynomials of degree ≤ 4 can be expressed in terms of radicals, those of polynomials of degree ≥ 5 cannot, in general. On a similar note, we briefly present the historically famous problem of geometric construction using ruler and compass, and show how “tiny” changes in the assumptions can drastically change the resulting set of objects that can be constructed.

Whenever the tools allowed are restricted to the extent that “intuitively computable” objects cannot be obtained using these tools alone, we speak of a **special-purpose model of computation**. Such models are of great practical interest because the tools allowed are tailored to the specific characteristics of the objects to be computed, and hence are efficient for this purpose. We present several examples close to hardware design.

From the theoretical point of view, however, **universal models of computation** are of prime interest. This concept arose from the natural question "What can be computed by an algorithm, and what cannot?". It was studied during the 1930s by Emil Post (1897–1954), Alonzo Church (1903-1995), Alan Turing (1912–1954), and other logicians. They defined various formal models of computation, such as production systems, recursive functions, the lambda calculus, and Turing machines, to capture the intuitive concept of "computation by the application of precise rules". All these different formal models of computation turned out to be equivalent. This fact greatly strengthens **Church's thesis** that the intuitive concept of algorithm is formalized correctly by any one of these mathematical systems. The models of computation defined by these logicians in the 1930s are **universal** in the sense that they can compute anything computable by any other model, given unbounded resources of time and memory. This concept of universal model of computation is a product of the 20-th century which lies at the center of the theory of computation.

The standard universal models of computation were designed to be conceptually simple: Their primitive operations are chosen to be as weak as possible, as long as they retain their property of being universal computing systems in the sense that they can simulate any computation performed on any other machine. It usually comes as a surprise to novices that the set of primitives of a universal computing machine can be so simple, as long as these machines possess two essential ingredients: *unbounded memory* and *unbounded time*.

In this introductory chapter we present 2 examples of universal models of computation:

- Markov algorithms, which access data and instructions in a sequential manner, and might be the architecture of choice for computers that have only sequential memory.
- A simple random access machines (RAM), based on a random access memory, whose architecture follows the von Neumann design of stored program computers.

Once one has learned to program basic data manipulation operations, such as moving and copying data and pattern matching, the claim that these primitive "computers" are universal becomes believable. Most simulations of a powerful computer on a simple one share three characteristics: It is straightforward in principle, it involves laborious coding in practice, and it explodes the space and time requirements of a computation. The weakness of the primitives, desirable from a theoretical point of view, has the consequence that as simple an operation as integer addition becomes an exercise in programming. The purpose of these examples is to support the idea that conceptually simple models of computation are equally powerful, in theory, as models that are much more complex, such as a high-level programming language. Unbounded time and memory is the key that lets the snail ultimately cover the same ground as the hare.

The theory of computability was developed in the 1930s, and greatly expanded in the 1950s and 1960s. Its basic ideas have become part of the foundation that any computer scientist is expected to know. But computability theory is not directly useful. It is based on the concept "computable in principle" but offers no concept of "feasible in practice". And feasibility, rather than "possible in principle", is the touchstone of computer science. Since the 1960s, a theory of the complexity of computation has being developed, with the goal of partitioning the range of computability into complexity classes according to time and space requirements. This theory is still in full development and breaking new ground, in particular with (as yet) exotic models of computation such as quantum computing or DNA computing.

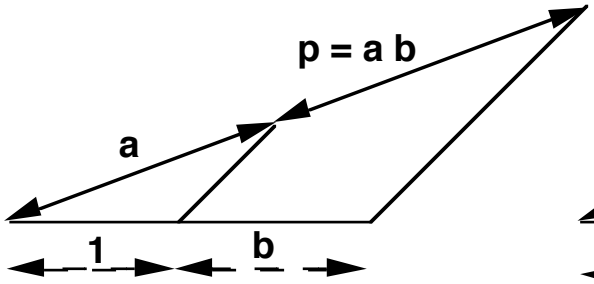
1.1 Construction with ruler and compass

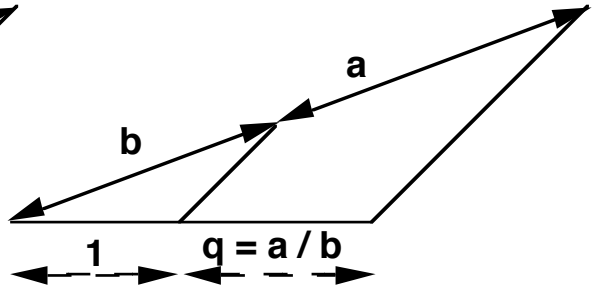
Plato (427-347 B.C.) considered ruler and compass the only suitable tools for geometric construction.

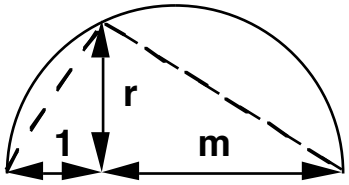
- Primitive objects: point, straight line (finite or unbounded), circle
- Primitive operations:
  given 2 points P, Q construct a line passing through P and Q
  given 2 points P, Q construct a circle with with center P that passes through Q
  given 2 lines L, L', or 2 circles C, C', or a line L and a circle C,
  construct a new point P at any intersection of such a pair.
- Compound object: any structure built from primitive objects by following a sequence of primitive operations.

Examples:
1. Given a line L and a point P on L, construct a line L' passing through P and orthogonal to L.
2. Construct a pair of Lines L, L' that defines angles of 90º, 60º, 45º, 30º, respectively
3. A segment is a triple (L, P, Q) with points P and Q on L. A segment of length 1 may be given, or it may be arbitrarily constructed to define a unit of measurement,

Angles and lengths that can be constructed:
- Starting with given angles, construct additional angles by bisection, addition and subtraction.
- Starting with a segment of unit length, construct additional lengths by bisection, addition, subtraction, **multiplication, division and square root.**







$$(1 + r^2) + (r^2 + m^2) = (1 + m)^2$$

$$r^2 = m$$

Thus, construction by ruler and compass is reduced to the question whether some desired quantity can be expressed in terms of rational operations and square roots.

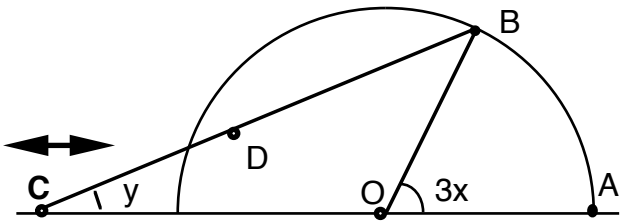
Thm: A proposed construction is possible by ruler and compass **iff** the numbers which define analytically the desired geometric elements can be derived from those defining the given elements by a finite number of rational operations and extractions of real square roots.

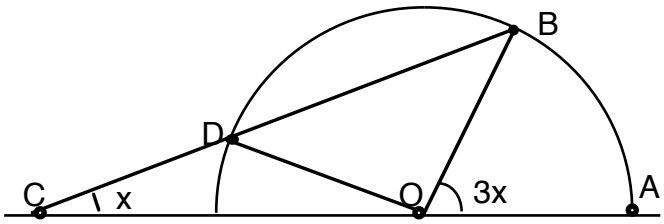
Possible construction: Equilateral polygon of 17 sides, "17-gon" (Carl Friedrich Gauss 1777-1855)
Impossible constructions: Equilateral heptagon , "7-gon" (Gauss). Doubling a unit cube (solve $x^3 = 2$).

Thm: There is no algorithm to trisect an arbitrary angle with ruler and compass.

... and yet, Archimedes (~287 B.C. - ~ 212 B.C) found a procedure to trisect an arbitrary angle:

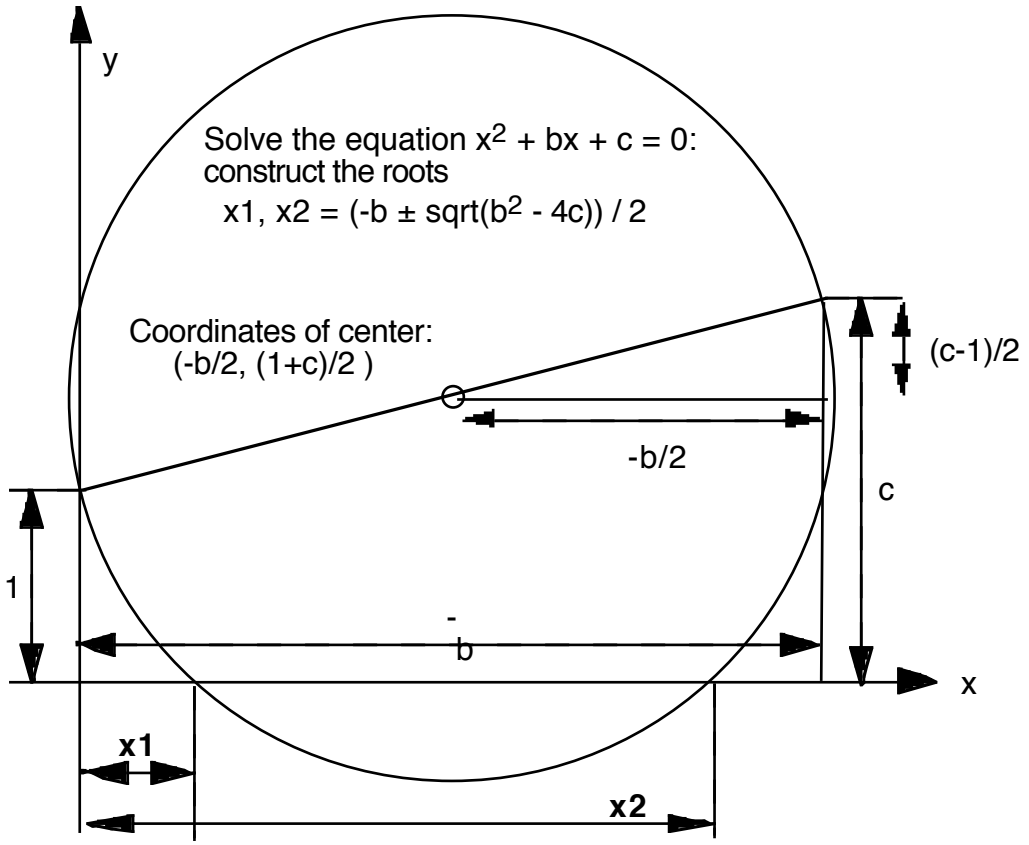
Given angle AOB = 3x in a unit circle. Ruler CB has a mark D at distance CD = radius = 1.
Slide C along the x-axis until D lies on the circle. Then, angle ACB = x.





Msg: "minor" changes in a model of computation may have drastic consequences - precise definition needed!

Hw 1.1: The quadratic equation $x^2 + bx + c = 0$ has roots x1, x2 = $(-b \pm \sqrt{b^2 - 4c}) / 2$. Prove that in the ruler and compass construction shown below, the segments x1 and x2 are the solutions of this equation.

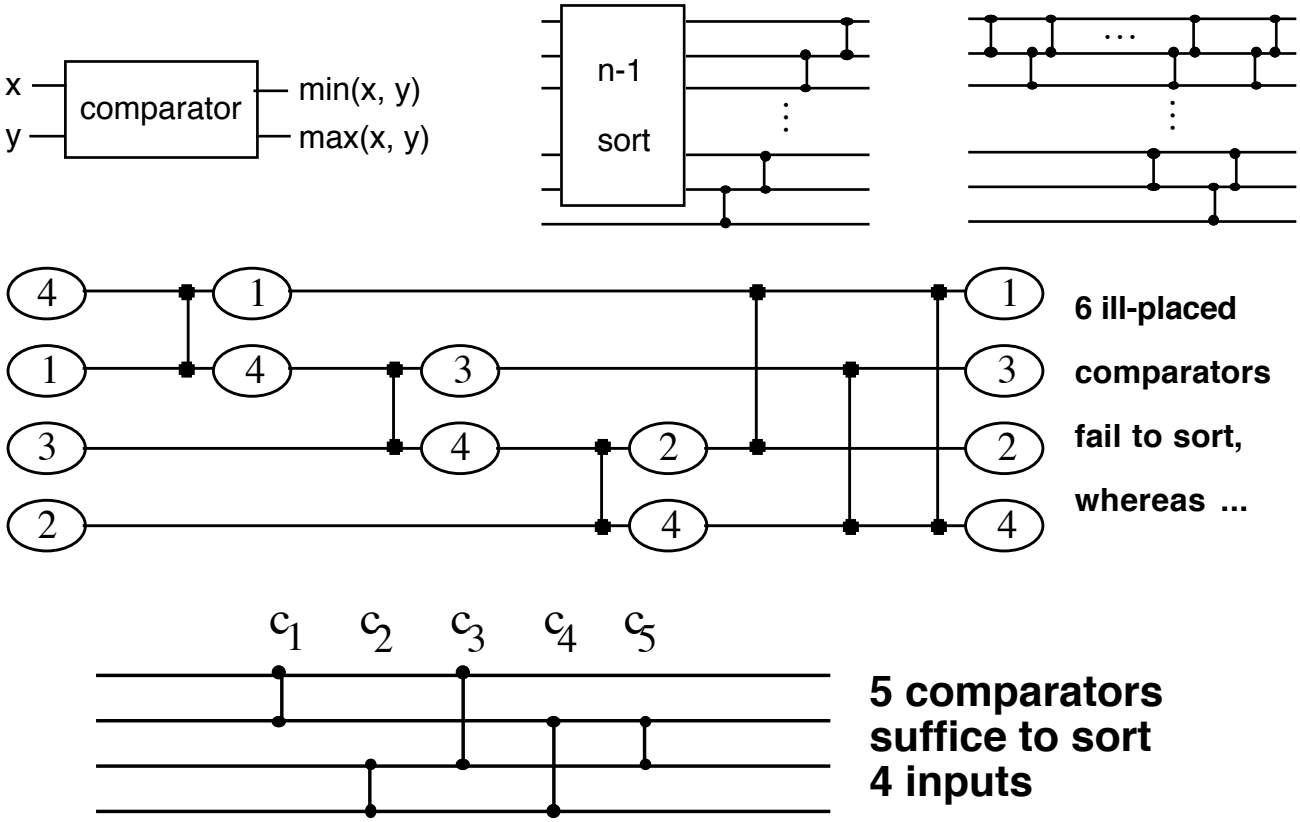


Solution 1.1: Substitute $x_c = -b/2$ and $y_c = (1 + c) / 2$ in the circle equation $(x - x_c)^2 + (y - y_c)^2$, set $y = 0$ to obtain the quadratic equation $x^2 + bx + c = 0$ whose roots are the desired values x_1 and x_2 .

1.2 Systolic algorithms, e.g. sorting networks

See D.E. Knuth: The art of computer programming, Vol.3 Sorting and searching, Addison-Wesley, 1973.

We consider a simple model of **parallel computation** - i.e. many operations are performed simultatneously. A sorting network is made up of parallel wires on which numbers travel from left to right. At places indicated by a vertical line, a comparator gate guides the smaller (lighter) number to the upper output wire, and the larger (heavier) number to the lower output wire. Clearly, enough comparators in the right places will cause the network to sort correctly - but how many gates does it take, where should they be placed?



Lemma: Given f monotonic, i.e. $x \le y \Rightarrow f(x) \le f(y)$.
If a **network of comparators** transforms $\mathbf{x} = x_1, .., x_n$ into $\mathbf{y} = y_1, .., y_n$, then it transforms f(**x**) into f(**y**).

Thm (0-1 principle): If a network S with n input lines sorts all 2^n vectors of 0s and 1s into non-decreasing order, S sorts any vector of n arbitrary numbers correctly.

Proof by contraposition: If S fails to sort some vector $\mathbf{x} = x_1, .., x_n$ of arbitrary numbers, it also fails to sort some binary vector f(**x**). Let S transform **x** into **y** with a "sorting error", i.e. $y_i > y_{i+1}$ for some index i. Construct a binary-valued monotonic function f: $f(x) = 0$ for $x < y_i$, $f(x) = 1$ for $x \ge y_i$. S transforms $f(\mathbf{x}) = f(x_1), .., f(x_n)$ into $f(\mathbf{y}) = f(y_1), .., f(y_n)$. Since $f(y_i) = 1 > f(y_{i+1}) = 0$, S fails to sort the binary vector **x**. QED.

Thm (testing proves correctness!): If a sorting network S that uses **adjacent comparisons only** sorts the "inverse vector" $x_1 > x_2 > ... > x_n$, it sorts any arbitrary vector.

Msg: Serial and parallel computation lead to entirely different resource characteristics.

Hw1.2: Prove: a sorting network using only **adjacent comparisons** must have ≥ n-choose-2 comparators.

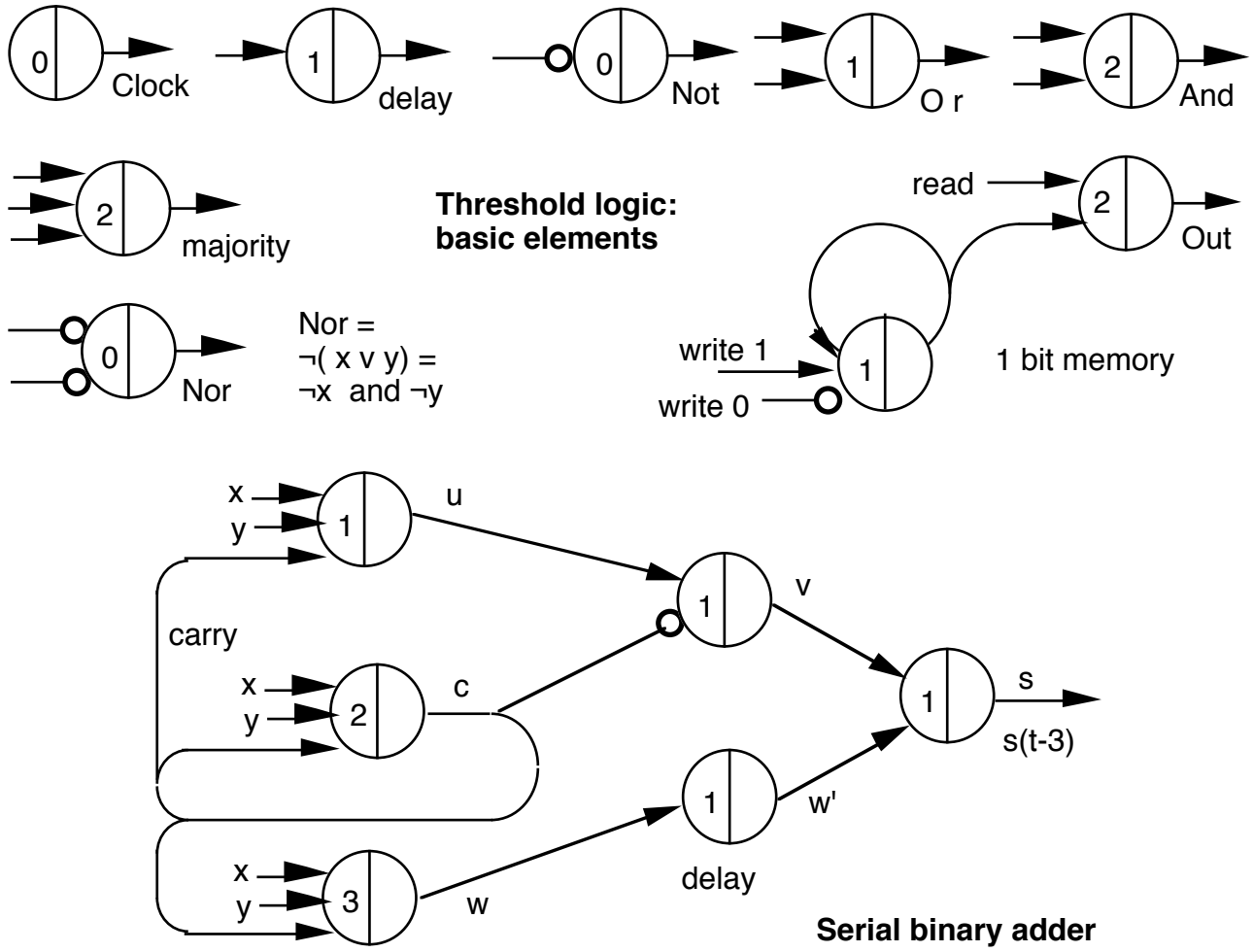
Solution 1.2: A **transposition** in a sequence $x_1, x_2, ..., x_n$ is a pair (i, j) with $i < j$ and $x_i > x_j$. The inverse vector has n-choose-2 transpositions. Every comparator reduces the number of transpositions by at most 1.

1.3 Threshold logic, perceptrons, artificial neural networks

W.S. McCulloch, W. Pitts: A logical calculus of the ideas immanent in nervous activity,
Bull. Math. Biophysics, Vol 5, 115-137, 1943.

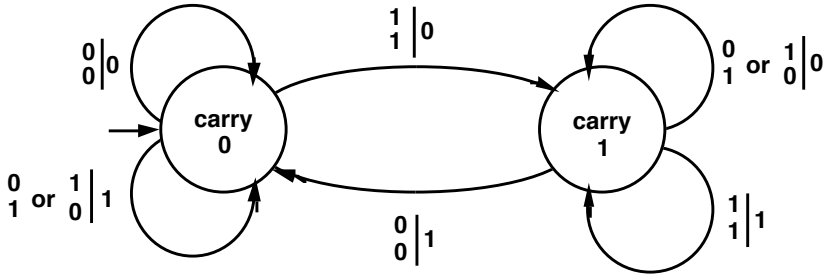
A threshold gate or "cell" is a logic element with multiple inputs that is characterized by its threshold value t. It counts, or adds up, the current input values and "fires", i.e. produces an output of 1, iff the sum equals or exceeds t. Threshold gates are more useful if they are generalized to include two kinds of inputs, excitatory and inhibitory inputs. A cell's output fires iff at least t excitatory inputs are on, and no inhibitor is on. In this model, which is used in the following examples, each inhibitor has veto power! (A variant called "subtractive inhibition" adds input values with positive or negative sign.)

We assume synchronous operation, with a gate delay of 1 clock tick. A threshold gate with inhibitory inputs is a universal logic element: the following figure shows how to implement and, or, not gates; delay and memory elements.



Msg: Logic design: how to assemble primitive elements into systems with a desired behavior. The theory of computation deals primarily with "black box behavior". But it insists that these black boxes **can** be built, at least in principle, from the simplest possible building blocks. Thus, the theory of computation involves an intricate interplay between abstract postulates and detailed "microprogramming".

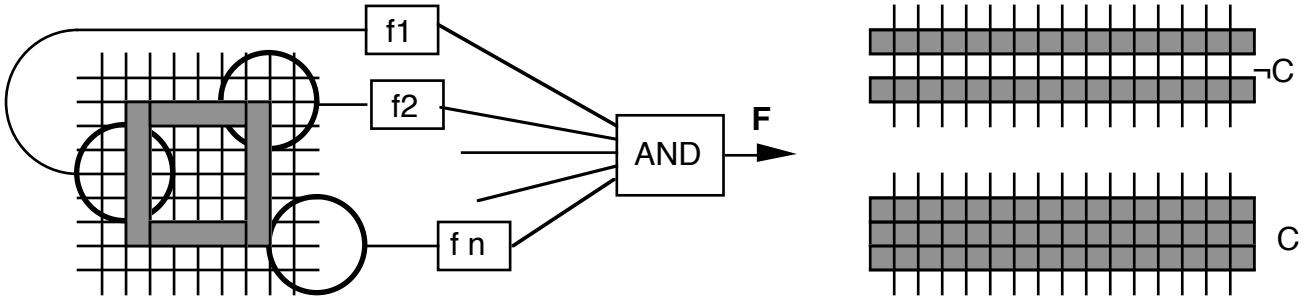
Example: the black-box behavior of the circuit above is specified by this finite state machine (output is s):



Artificial neural nets applied to picture recognition

F. Rosenblatt: Principles of neurodynamics, Spartan Books, NY 1962. "Model of the eye's retina".
M. Minsky, S. Papert: Perceptrons, MIT Press, 1969.

We consider digital pictures, i.e. 2-d arrays of black and white pixels. We define devices, called perceptrons, that recognize certain classes of pictures that share some specific property, such as "convex" or "connected". Perceptrons are an example of parallel computation based on combining local predicates. What geometric properties can be recognized by such devices? A simple variant of "perceptrons", called 'conjunctively local', is illustrated in the figure below. Let f1, ..., fn be predicates with a bounded fanout, i.e. each fi looks at ≤ k pixels. The perceptron uses AND as the threshold operator, which means that the output F = AND (f1, ..., fn).



Convexity can be recognized. Introduce an f for each triple a, b, c of points on a straight line, with b between a and c. Define f = 0 iff a and c are black, b white; f = 1 otherwise.

Connectedness cannot be recognized by any perceptron of bounded fanout k. Pf by contradiction: Consider the 2 pictures above labeled C and ¬C that extend for > k pixels. If perceptron P can distinguish them, there must be an f* that yields f* = 0 on ¬C, and f* = 1 on C. Since f* looks at ≤ k pixels, there is a pixel p in the center row not looked at by P. By blackening this pixel p, we change ¬C into a connected picture C'. But f* "does not notice" this change, keeps voting f* = 0, and causes F to output the wrong value on C'.

Conclusion: convexity can be determined as a cumulation of local features, whereas connectedness cannot.

Msg: Combinational logic alone is weak. A universal model of computation needs unbounded memory, and something like feedback loops or recursion.

Hw 1.3: Define an interesting model suitable for computing pictures on an infinite 2-d array of pixels. Study its properties. Is your model "universal", in the sense that it is able to compute any "computable picture"?

Ex: Let N = {1, 2, ..} be the natural numbers. Consider the infinite pixel array N x N, and pictures P over this array, defined as functions P : N x N -> {0, 1}. Consider 4 primitive operations, each of which takes an arbitrary row or column, and colors it with a single color, either 0 or 1: Set [row, column] k to [0, 1]. Characterize the class C of pictures generated by finite programs, i.e. finite sequences of operations of the type above. Give a decision procedure that decides whether a given picture P is in C; and if it is, provides lower and upper bounds on the complexity of P, i.e. on the length of its shortest program that generates P.

1.4 Grammars and "languages": Chomsky's hierarchy

N. Chomsky: Three models for the description of language, IRE Trans. Information Th. 2, 113-124, 1956.

Motivating example:
Sentence -> Noun Verb Noun, e.g.: Bob loves Alice
Sentence -> Sentence Conjunction Sentence, e.g.: Bob loves Alice and Rome fights Carthage

Grammar G(V, A, P, S). V: alphabet of non-terminal symbols, "variables, "grammatical types";
A: alphabet of terminal symbols, $S \in V$: start symbol, "sentence";
P: unordered set of productions of the form $L \to R$, where $L, R \in (V \cup A)^*$

Rewriting step: for $x, y, y', z \in (V \cup A)^*$, $u \to v$ iff $u = xyz$, $v = xy'z$ and $y \to y' \in P$
Derivation: "->*" is the transitive, reflexive closure of "->", i.e.
$u \to^* v$ iff $\exists\, w0, w1, \ldots wj$, with $j \geq 0$, $u = w0$, $w(i-1) \to wi$, $wj = v$
Language defined by G: $L(G) = \{ w \in A^* \mid S \to^* w \}$

Various restrictions on the productions define different types of grammars and corresponding languages:
Type 0, **phrase structure grammar**: No restrictions
Type 1, **context sensitive**: $|L| \leq |R|$, (exception: $S \to \varepsilon$ is allowed if S never occurs on any right-hand side)
Type 2, **context free**: $L \in V$
Type 3, **regular**: $L \in V$, $R = a$ or $R = aX$, where $a \in A$, $X \in V$

Exercise: Define a grammar for some tiny subset of natural language, and illustrate it with examples. Try this for a subset of English and a semantically similar subset of another language, and show that different natural languages have different grammars.

1.5 Markov algorithms: A universal model of computation

A.A. Markov (1903-1979) developed his "Theory of algorithms" around 1951. Markov algorithms can be interpreted as computer architecture for memory with sequential access only: Computation is modeled as a **deterministic** transformation of an input string into an output string, e.g. '1+2' => '3', according to rewrite rules that are scanned sequentially. The most comprehensive treatment in English is:
A.A. Markov, N.M. Nagorny: The Theory of Algorithms, (English transl), Kluwer Academic Publishers, 1988.

Alphabet A = {0, 1, ..}, our examples use A = {0, 1}. Marker alphabet M = { α, β, ..}.
Sequence (ordered) P = P1, P2, .. of **rewriting rules** (productions), which are of 2 types:
$Pi = x \to y$ (continue) or $Pi = x \neg y$ (terminate), where $x, y \in (A \cup M)^*$.

A rule $P = x \to y$ applies to the current data string D if x occurs as a contiguous substring in D, whereafter this occurrence of x is replaced by y. A terminal rule stops the process.
A Markov algorithm computes a partial function $f: A^* \to A^*$ by transforming a data string D, step-by-step. Initially, D is an input string $s \in A^*$, and if the algorithm terminates, D = f(s). Between initial and final value, D is tranformed by applying a rewrite rule $Pi = x \to y$ or $Pi = x \neg y$.
The tranformation D <- Pi (D) is chosen according to the following execution rule:
use the **first rule** that applies to the data string D, apply it at the **leftmost pattern match**.

Examples

1) Change all 0s to 1s. P1: $0 \to 1$. No terminal rule is needed, algorithm stops when no rule applies.

2) Generate 0s forever: P1: $\varepsilon \to 0$. "ε" is the nullstring, it always matches to the left of any input string.

3) Append prefix '101': P1: $\varepsilon \neg 101$. Terminal rule stops the rewriting process.

4) Append suffix '101'. Need marker, M = {α}. Careful when sequencing the rewrite rules!
P1: $\alpha 0 \to 0 \alpha$, P2: $\alpha 1 \to 1 \alpha$, P3: $\alpha \neg 101$, P4: $\varepsilon \to \alpha$
Rule P4 is executed first and generates the marker α. P4 appears last in the production sequence P, where it is protected by P3 from ever being executed again. The top priority rules P1 and P2 move α from the beginning to the end of the data string. When α reaches its destination, the terminal rule P3 converts it to the suffix 101 and stops the process.

Notation

Usually we sequence the rules implicitly by writing them on separate lines. In example 4, the order in which the two rules P1: $\alpha\ 0 \rightarrow 0\ \alpha$, P2: $\alpha 1 \rightarrow 1\ \alpha$ appear, i.e. P1 P2 or P2 P1, is irrelevant. In order to emphasize that a subset of the rules may be permuted arbitrarily among themselves, we may write these on the same line:

4a) $\alpha\ 0 \rightarrow 0\ \alpha,\ \alpha 1 \rightarrow 1\ \alpha$
$\alpha \neg 101$
$\varepsilon \rightarrow \alpha$

Moreover, P1 and P2 have a similar structure. With a large alphabet A = { 0, 1, 2, ..} we need |A| rules of the type $\alpha\ B \rightarrow B\ \alpha$, where the variable B ranges over A. In order to abbrievate the text, we write the algorithm as :

4b) $\alpha\ B \rightarrow B\ \alpha$
$\alpha \neg 101$
$\varepsilon \rightarrow \alpha$

This is called a **production schema**, a meta notation which implies or generates the actual rule sequence 4a). A schema rule $\alpha\ B \rightarrow B\ \alpha$ implies that the relative order of the individual rules generated is irrelevant.

Algorithm design

How do you invent Markov algorithms for specific tasks? Is there a "software engineering discipline" for Markov algorithms that leads you to compose a complex algorithm from standard building blocks by following general rules? Yes, the diligent "Markov programmer" soon discovers recurrent ideas.

Recurrent issues and problems in Markov programming:

a) Since the data string can only be accessed sequentially, i.e. you can't assign an "address" to some data of interest, you must identify the places in the string "where things happen" by placing markers. Thus, the data string is used as a content accessed memory. This initial placement of markers is "the initialization phase".

b) After the initial placement of markers, each operation on the data string is usually preceded by a scan from the beginning of the string to a specific marker. Let's call the substring consisting only of symbols from the alphabet A, with markers removed, the "restricted data string". We call the repeated scans for markers, followed by operations on the restricted data string, the "operating phase".

c) At various times during execution, at the latest when the operations on the data string (restricted to the alphabet A) are done, now useless or harmful markers clutter the data string. "Cleaning up", i.e. removing unneeded markers, is non-trivial: if you place $\alpha \rightarrow \varepsilon$ too early in the production sequence, marker α may be wiped out as soon as it is created!

d) Many simple Markov algorithms can be designed as executing in three phases: initialization, operating phase, clean-up phase. More complex algorithms, of course, may involve repeated creation and removal of markers, with various operating phases in between.

e) Once you have a plan for a Markov algorithm, it is often easy to see what type of productions are needed. Determining and verifying the correct order of these productions, however, is usually difficult. The presence or absence of specific markers in the data string helps to prove that certain productions cannot apply at a particular moment; thus, they may be ignored when reasoning about what happens during certain time intervals.

f) The crucial step is to invent invariants that are maintained at each iteration. This is by far the most important technique for understanding how an algorithm works, and proving it to be correct!

g) If you see a Markov algorithm without explanation of its logic, it is very difficult to "reverse engineer" the designer's ideas. Therefore we will only look at algorithms that are supported by arguments about how and why they work.

The following **examples** illustrate frequently useful techniques.

5) Generate a palindrome, $f(s) = s\ s^{reverse}$, A = {0, 1}

We postulate the following invariant for the operating phase. Let $s = s_L\ s_R$ be a partition of the data string s into a prefix s_L and a suffix s_R . Ex: s = a b c d e , s_L = a b, s_R = c d e. At all times, the current string s has the form $s = s_L\ \gamma\ s_L^{reverse}\ \alpha\ s_R$. In the example above: s = a b γ b a α c d e. The marker γ marks the center of the palindrome built so far from the prefix s_L; α marks the boundary between the palindrome already built and the suffix s_R yet to be processed. If we now remove the first letter of s_R and copy it twice, to the left and right of γ , the invariant is reestablished, with s_R shrunk and s_L expanded: a b c γ c b a α d e .

$0\ \beta_0 \rightarrow \beta_0\ 0,\ 1\ \beta_0 \rightarrow \beta_0\ 1, 0\ \beta_1 \rightarrow \beta_1\ 0,\ 1\ \beta_1 \rightarrow \beta_1\ 1$	the carry β_0 or β_1 moves left
$\gamma\ \beta_0 \rightarrow 0\ \gamma\ 0,\ \gamma\ \beta_1 \rightarrow 1\ \gamma\ 1$	carry has reached the center, produces a bit and its mirror image
$\alpha\ 0 \rightarrow \beta_0\ \alpha\ ,\ \alpha\ 1 \rightarrow \beta_1\ \alpha$	remove the leftmost bit of s_R and save its value in markers β_0 or β_1
$\alpha \rightarrow \varepsilon\ ,\ \gamma \neg \varepsilon$	at the end, remove the markers; notice terminating production!
$\varepsilon \rightarrow \gamma\ \alpha$	at the beginning, create the markers at the far left

Reversing a string is the key part of constructing a palindrome. We can easily modify Algorithm 5 to merely reverse a string by changing the invariant $s = s_L\ \gamma\ s_L^{reverse}\ \alpha\ s_R$ to $s = \gamma\ s_L^{reverse}\ \alpha\ s_R$, and the rules $\gamma\ \beta_0 \rightarrow 0\ \gamma\ 0,\ \gamma\ \beta_1 \rightarrow 1\ \gamma\ 1$ to $\gamma\ \beta_0 \rightarrow \gamma\ 0,\ \gamma\ \beta_1 \rightarrow \gamma\ 1$.
The following example illustrates a technique that uses a single marker for different purposes at different times, and thus, is be a bit hard to fathom.

6) Reverse the input string s: f(s) = $s^{reverse}$

P1	$\alpha\ \alpha\ \alpha \rightarrow \alpha\ \alpha$	a double α that merges with a single α gets trimmed back
P2	$\alpha\ \alpha\ B \rightarrow B\ \alpha\ \alpha$	double α's function as a pseudo marker to wipe out single α's !
P3	$\alpha\ \alpha \neg\ \varepsilon$	
P4	α B' B" -> B" α B'	this schema stands for 4 rules of the type α 1 0 -> 0 α 1
P5	ε -> α	gets executed repeatedly, but P2 and P3 stop P5 from creating more than 2 consecutive α's

This algorithm is best understood as executing in two phases.
Phase 1, when productions P4 and P5 are active, mixes repeated initial placements of the marker α with operations that reverse the restricted data string. Invariant of Phase 1: there are never 2 consecutive α's in the data string, hence productions P1, P2, P3 do not apply.
After phase 1, there is a brief interlude where production P5 creates 2 α's at the head of the string.
Phase 2: the cleanup phase removes all markers by activating productions P1 and P2. Invariant of Phase 2: the data string always contains a pair of consecutive α's. Thanks to the "pseudo marker" αα, one of P1, P2, P3 always applies, and P4 and P5 no longer get executed. At the very end, the terminating production P3 wipes out the last markers.

7) Double the input string s: f(s) = ss

Introduce markers α, γ to delimit parts of the current string with distinct roles. Let $s = s_L\ s_R$ be a partition of s into a prefix s_L and a suffix s_R . In general, the current string has the form $s_L\ \alpha\ s_R\ \gamma\ s_R$. By removing the rightmost bit of s_L and copying it twice at the head of each string s_R , we maintain this invariant with s_L shrunk and s_R expanded. Some stages of the current string: s, $s\ \alpha\ \gamma$, $s_L\ \alpha\ s_R\ \gamma\ s_R$, $\alpha\ s\ \gamma\ s$, ss.

$\beta_0\ \gamma \rightarrow \gamma\ 0\ ,\ \beta_1\ \gamma \rightarrow \gamma\ 1$	the carry β_0 or β_1 converts back to 0 or 1
$\beta_0\ 0 \rightarrow 0\ \beta_0,\ \beta_0\ 1 \rightarrow 1\ \beta_0\ ,\beta_1\ 0 \rightarrow 0\ \beta_1,\ \beta_1\ 1 \rightarrow 1\ \beta_1$	the carry β_0 or β_1 moves right
$\alpha\ \gamma\ 0 \rightarrow 0\ \alpha\ \gamma\ ,\quad \alpha\ \gamma\ 1 \rightarrow 1\ \alpha\ \gamma$	the markers travel to the far right
$0\ \alpha \rightarrow \alpha\ 0\ \beta_0\ ,\ 1\ \alpha \rightarrow \alpha\ 1\ \beta_1$	move one bit of s_L to expand the first s_R and generate a carry β
$\alpha \rightarrow \varepsilon\ ,\ \gamma \neg \varepsilon$	at the end, remove the markers; notice terminating production!
$\varepsilon \rightarrow \alpha\ \gamma$	at the beginning, create the markers at the far left

8) Serial binary adder

Given 2 n-bit binary integers $x = x_n .. x_1 x_0$, $y = y_n .. y_1 y_0$, $n \geq 0$, compute their sum $z = z_{n+1} z_n .. z_1 z_0$,
$A = \{0, 1, +, =\}$, $M = \{ \beta_0, \beta_1 \}$. β_0, β_1 store and transport a single bit.
Coding: Input string: x+y=0 , Output string: =z.
Idea. The input string x+y=0 gets transformed into intermediate strings of the form
$x_L + y_L = z_R$, where x_L is $x_n .. x_{i+1} x_i$, y_L is $y_n .. y_{i+1} y_i$, z_R is $z_i z_{i-1} .. z_1 z_0$.
As the bit pair x_i, y_i is being cut off from the tail of x_L and y_L, a sum bit z_i is appended to the front of z_R .
Invariant I: $z_R = x_R + y_R$, where x_R , y_R are the tails cut off from x, y, respectively. I holds initially.
The algorithm is built from the following components:

Full adder: The addition logic is represented by 8 productions that can be written in any order:

$0\beta_0=0$ -> =00	$0\beta_1=0$ -> =01	$1\beta_0=0$ -> =01	$1\beta_1=0$ -> =10
$0\beta_0=1$ -> =01	$0\beta_1=1$ -> =10	$1\beta_0=1$ -> =10	$1\beta_1=1$ -> =11

Save the least significant bit of x_L: 0+ -> $+\beta_0$ 1+ -> $+\beta_1$

Transport this bit β next to the least significant bit of y_L:

$\beta_0 0$ -> $0\beta_0$	$\beta_0 1$ -> $1\beta_0$	$\beta_1 0$ -> $0\beta_1$	$\beta_1 1$ -> $1\beta_1$

These building blocks are sequenced such that, at any time during execution, the string contains at most one β:

Full adder
Transport β
Save the least significant bit of x_L
Remove "+": += -> =

Markov algorithms are one of several universal models of computation: you can program an algorithm to compute anything "computable". E.g.. there is a Markov algorithm that simulates a universal Turing machine.
We have seen simple copy and data movement operations usefle for string matching and copying. In order to make this assertion of universality plausible, we now program simple control structures.

9) Control logic: f(s) ≡ if s = x then y else z; s, x, y, z ∈ A*.

Because x, y, and z are parameters that may assume arbitrary string values, we must expand our notation. We introduce an informal meta-language consisting of new symbols. For example, the "production" x ¬ y is really a **production schema** that stands for an arbitrary specific instance, e.g for 00 ¬ 111. Production schemata also let us write an algorithm more concisely (particularly when the alphabet A is large). Let B be a symbol that stands for 0 or 1.The schema $B\alpha$ -> α stands for the pair 0α -> α, 1α -> α.
The following algorithm distinguishes 3 cases:
1) s = x, 2) x is a proper substring of s, 3) s ≠ x and x isn't a substring of s.

$B\alpha$ -> α	"eraser" α wipes out 0s and 1s to the left
αB -> α	α wipes out 0s and 1s to the right
α ¬ z	if s ≠ x then z. How do we know s ≠ x? That was checked when generating α
xB -> α	if s contains x as a proper substring ..
Bx -> α	.. generate α
x ¬ y	if s = x then y
ε -> α	if s ≠ x then generate α

Hw: Consider the context-free grammar G: E -> x | E¬ | E E ∧ | E E ∨ that generates the language Ls of Boolean suffix expressions over a single literal x and three Boolean operators Not, And, Or. Write a Markov algorithm that distinguishes syntactically correct expressions from incorrect ones.

Hw: Learn to use the software package Exorciser demonstrated in class, and solve the exercises on Markov algorithms contained therein.

1.6 Random access machines (RAM), the ultimate RISC

The model of computation called **random access machine,** RAM, is used most often in algorithm analysis. It is significantly more "powerful", in the sense of efficiency, than either Markov algorithms or Turing machine because its memory is not a tape, but an array with random access. Provided the programmer knows where an item currently needed is stored, a RAM can access it in a single memory reference, avoiding the sequential access tape searching that a Turing machine or a Markov algorithm need.

A RAM is essentially a *random access memory*, also abbreviated as RAM, of unbounded capacity, as suggested in the Fig. below. The memory consists of an infinite array of cells, addressed 0, 1, 2, … . To make things simple we assume that each cell can hold a number, say an integer, of arbitrary size, as the arrow pointing to the right suggests. A further assumption is that an arithmetic operation (+ , – , · , /) takes unit time, regardless of the size of the numbers involved. This assumption is unrealistic in a computation where numbers may grow very large, but is often useful. As is the case with all models, the responsibility for using them properly lies with the user.

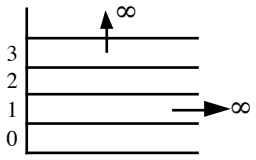


Fig.: RAM
with unbounded address space
and unbounded cell size

The ultimate RISC.

RISC stands for *Reduced Instruction Set Computer*, a machine that has only a few types of instructions built into hardware. What is the minimum number of instructions a computer needs in order to be universal? One! Consider a stored-program computer of the "von Neumann type" where data and program are stored in the same memory (John von Neumann, 1903 – 1957). Let the random access memory (RAM) be "doubly infinite": There is a *countable infinity* of memory cells addressed 0, 1, … , each of which can hold an integer of arbitrary size, or an instruction. We assume that the constant 1 is hardwired into memory cell 1; from 1 any other integer can be constructed. There is a single type of "three-address instruction" which we call "subtract, test and jump", abbreviated as

STJ x, y, z

where x, y, and z are addresses. Its semantics is equivalent to

STJ x, y, z ⇔ x := x – y; if x ≤ 0 then goto z;

x, y, and z refer to cells Cx, Cy, and Cz. The contents of Cx and Cy are treated as data (an integer); the contents of Cz, as an instruction.

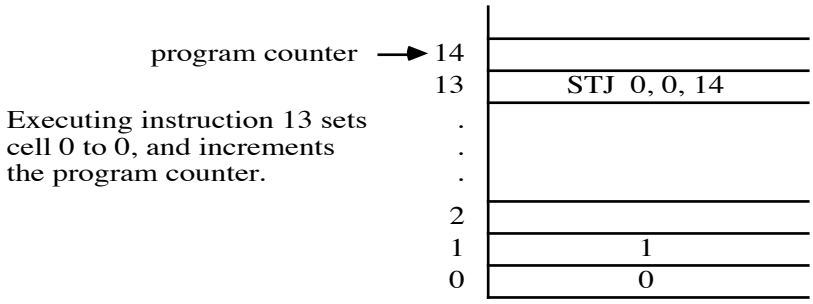


Fig.: Stored program computer: Data and instructions share the memory.

Since this RISC has just one type of instruction, we waste no space on an op-code field. But an instruction contains three addresses, each of which is an unbounded integer. In theory, three unbounded integers can be packed into the same space required for a single unbounded integer. This simple idea leads to a well-known technique introduced into mathematical logic by Kurt Gödel (1906 – 1978).

Exercise: Gödel numbering

a) Motel Infinity has a countable infinity of rooms numbered 0, 1, 2, … . Every room is occupied, so the sign claims "No Vacancy". Convince the manager that there is room for one more person.

b) Assume that a memory cell in our RISC stores an integer as a sign bit followed by a sequence d0, d1, d2, … of decimal digits, least significant first. Devise a scheme for storing three addresses in one cell.

c) Show how a sequence of positive integers i1, i2, , in of arbitrary length n can be encoded in a single natural number j: Given j, the sequence can be uniquely reconstructed. Gödel's solution: $2^{i1}\ 3^{i2}\ 5^{i3}$ ...(n-th prime)^{in} .

Basic RISC program fragments

To convince ourselves that this computer is universal we begin by writing program fragments to implement simple operations, such as arithmetic and assignment operator. Programming these fragments naturally leads us to introduce basic concepts of assembly language, in particular symbolic and relative addressing.

Set the content of cell 0 to 0: STJ 0, 0, .+1

Whatever the current content of cell 0, subtract it from itself to obtain the integer 0. This instruction resides at some address in memory, which we abbreviate as '.', read as "the current value of the program counter". '.+1' is the next address, so regardless of the outcome of the test, control flows to the next instruction in memory.
a := b, where a and b are symbolic addresses. Use a temporary variable t:

```
STJ  t, t, .+1        { t := 0 }
STJ  t, b, .+1        { t := –b }
STJ  a, a, .+1        { a := 0 }
STJ  a, t, .+1        { a := –t, so now a = b }
```

Exercise: A program library

(a) Write RISC programs for
a := b + c, a := b · c, a := b div c, a := b mod c, a := |b|, a : = min(b, c), a := gcd(b, c).
(b) Show how this RISC can compute with rational numbers represented by a pair [a, b] of integers denoting numerator and denominator.
(c) (Advanced) Show that this RISC is universal, in the sense that it can simulate any computation done by any other computer.

The exercise of building up a RISC program library for elementary functions provides the same experience as the equivalent exercise for Turing machines, but leads to the goal much faster, since the primitive STJ does a lot more work in a single operation than the primitives of a Turing machine.

1.7 Programming languages, [un-]decidability

Having learned to program an extremely primitive computer, it is obvious that any of the conventional programming languages is a universal model of computation. The power of universal computation comes at a conceptual cost: many reasonable questions about the behavior of a universal computing model are **undecidable**: that is, they cannot be answered by following any effective decision process. The **halting problem** is the standard example. We present it here in the words of Christopher Strachey, as written in a letter to the editor of The Computer Journal ( ):

To the Editor, The Computer Journal

An impossible program

Sir,
A well-known piece of folk-lore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run. I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL, but not in any essential way.
Suppose T[R] is a Boolean function taking a routine (or program) R with no formal or free variables as its argument and that for all R, T[R] = True if R terminates if run and that T[R] = False if R does not terminate. Consider the routine P defined as follows

```
rec routine P
    §L:  if T[P] go to L
         Return §
```

If T[P] = True the routine P will loop, and it will only terminate if T[P] = False. In each case T[P] has exactly thew wrong value, and this contradiction shows that the function T cannot exist.

Yours faithfully, C. Strachey
Churchill College, Cambridge

End of Ch1