Algorithms and Machines

Massimo Di Pierro

Contents

1. Introduction

The English term *Algorithm* derives from the name of the 8th century Arab mathematician *al-Khowarizmi*[1]. He introduced to the Arab world the decimal numbering system originated in India and invented the symbol for zero (0). The English term *Algebra* comes from the title of his book *Al-jabr wa'l muqabala*.

The first clear written example of an algorithms as we conceive it today appears in the book of *Euclid*, *The Elements*, written in the 4th century B.C. In his book Euclid proposes a number of algorithms, i.e. step by step recipes, aimed to perform geometric and mathematical computations[2]. Perhaps the most clear example an algorithm is Euclid's procedure to compute the gretest common divisor of two integer numbers x and y:

1. **If** x is smaller than y then swap the values of x and y.
2. Compute the reminder of x divided by y and store its result into x
3. **If** m is not equal zero then **go** to step 1
4. **Stop**. The answer (greatest common divisor) is y

For the purpose of this book an Algorithm can be defined as a step by step decription of the procedure to perform a task or carry on a computation. The description has to be un-ambiguous and each steps must be simple enough to be executable by a machine. Ultimately by machine we mean a *Computer*.

An algorithm is an abstract concept but, once we specify a particular machine, a compehensive set of elementary steps, and a language to describe each step, the algorithm becomes a *program* (a sequence of steps written in that language). A language that can be undesrtood by a machine is called a *programming language*. Throughout this book we will use the following expressions: "a machine understand a language" meaning that each word in the language correspond to a particular *component of* or *function performed by* the machine; "a machine executes a program" meaning that the machine performs a number of functions in

the order those functions are listed in the program. The program must be written in a language understandable by the machine.

A famous quote from *E. W. Dijkstra*, one of the most prominent contemporary computer scientists is that "Computer Science is no more about computers than astronomy is about telescopes". In fact astronony is the study of the universe as a whole, its contituents (planes, starts, galaxies), and its history. The telescope is just one instrument, although the most important, used by the astronomers to look at the sky. In a similar fashion *computer science* is not about the *hardware* (the physical object Computer, the machine) but it is about *software* (programming languages, algorithms, information representation, artificial intelligence). The Computer is an instrument, certaily the most important, used by computer scientists to undesrtand, develop and execute programs.

Computer Science can be defined as the art to understand the logical thought process (or at leat some parts of it) and represent it in the form of an algorithm written in a programming language. One can think of the software as the mind and the hardware as the brain. There cannot be a mind without a brain and the brain is useless without a mind.

In this book we consider a very specific model of computation, the one propsed by *Alan Turing* in 1930. We also have in mind a specific model of machine architecture proposed by *John von Neumann* in 1940. Any of such machines will be referred to as a *classical computer*. The reader should be aware that there are alterantive models of computation such as *Celullar Automata*, *Neural Networks* and *Quantum Computers*. It is a remarkable fact that classical computers are all equivalent to a *Turing Machine* and each of them has the ability to emulate (or simulate) any of the other known models of computation (for example it is common for physicists to use classical computers to simulate phenomena that belong to the domain of quantum physics). The field of research that studies what can be or cannot be computed by machines is called *Computability Theory*.

The difference beween one model of computation and another is not in what a particular model can do or not do in respect to another model. The difference is in how the process of computation is described and in what constitutes an elementary step. As a consequence of these differences a Quantum Computer may be faster then a classical computer in solving a particular problem, while this latter will be faster in solving some other particular problem. The theory that describes the efficiency of a particular computational model (in respect to some class of problems) is called *Computational Complexity Theory*. This is a very active field of research and it is only briefly touched by the book.

This book is designed for a one semester introductory course on computing. The book's main focus is on the concept of algorithm and how algorithms can be written in programming languages understandable by a machine. [FILL HERE][3][4][5][**?**]

1.1. Book summary

This book consists of 10 main chapters including this introduction.

- In Chapter 2 we briefly review the history of algorithms and machines, how the problem of describing the logic thought process and the problem of automating algebraic calculations became a single problem partially solved by the invention of the computer.

- In Chapter 3 we introduce examples of software and of machine programming languages. We will use two languages: a visual language based on flow charts and a machine language that can be understood by a simple simulated machine. The book comes with a program called "Flow Chart Runner" that will allow readers to program by designing flow charts, compile the programs into the machine language and execute the compiled programs, step by step, within the simulated machine.

- In Chapter 4 we introduce Boolean Algebra and we present a typical hardware implementation of the modern computers, with particular reguard to the machine discussed in Chapter 3. We also discuss similarities and differences with common comericial computers.

- Chapter 5 is about high level programming languages and how they differ from low level languages such as the machine language discussed in the preceding chapters. We also discuss how high level languages are mapped into low level machine languages. The main language used in this chapter is Python. The Python programming language was originally developed for teaching without compromise on the expressive power. Python is basically an interpreted procedural language that supports some syntactic expressions typical of object orinted languages (such as classes, inheritance and operator overloading). Python also supports elements of functional programming (such as lambda expression). In Chapter 5 we review the examples presented in Chapter 3 under the light of this high level language. The Python interpreter is distributed with this book.

- In Chapter 6 we discuss partical aspects of information representation and we present some common data types used to store structured information (lists, trees and graphs). The problem of information representation is crucial in Alrtificial Intelligence and and it is one of the main aspects that differentiate high level programming languages from low level languages.

- In Chapter 7 a variety of typical computer science algorithms are discussed. Algorithms are selected according with their historic relevance and thier practical applications. All algorithms are written in the Python language. Some of this algorithms are quite complex and should be skipped at a first reading of this book. All these algorithms are implemented in the Algorithm Animator program that also accompanies this book.

- In Chapter 8 we discuss modern applcations of computer science in a number of fields such as mathematics, physics, engineering, biology, telecommunications and finance.

- In Chapter 9 we discuss the prospects for Artificial Intelligence that we see as the ultimate goal of Computer Science. We present in some detail some of the theorems of Alan Turing and Kurt Gödel and their practical consequences in respect to Artificial Intelligence. In this chapter we also discuss differences between procedural languages and functional languages, recursion and infinite loops.

- Chapter 10 is the last and final one of the book. We review a number of philosophical and ethical problems raised by the modern advances and applications of Computer Science. We discuss common misconception about the field and prospects for its future.

- The Book terminates with a Glossary of terms and three appendices, which contain tutorials for the three programs distributed with the book: the Flow Chart Runner, the Algorithm Animator and the Python Interpreter.

2. History

2.1. The Abacus and positional notation

The *Abacus* was invented around the year 3000 B.C., in China, long before numbers as we conceive them today. Abacus is a Latin word. It comes from the Greek word *abax* or *abakon* that means table or tablet. The Abacus is a mechanical device used for counting, originally made of wood and stone. The Abacus has a frame that holds rods. Mounted on the rods are sliding beads made of stones. A stone in the Abacus may represent an object or a set of objects. The operation of addition and subtraction is performed my moving the stones up and down along the rods.

The novelty in the Abacus is in the concept of positional notation: each stone may in fact represent a single object or a set of objects depending by its position. The same concept is used by the Arabic numbering system, the same system we use today. In the Arabic numbering system combinations of 10 digits $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$ represent numbers. The meaning of each digit depends upon the position of the digit in the number. For example, in the number 123 the first digit on the left represents one 1×100; the middle digit represent 2×10 and the third right-most digit represent 3×1. Although we take positional notation for granted, only 2000 years ago, this concept was not popular. The Roman numbering system made a very poor use of positional notation and, in fact, Romans did not have a way to represent the zero, negative numbers nor numbers bigger than 4000. One can easily imagine the limitation of such an ancient numbering system.

The Arabic numbering system that we use today was first introduced in Europe by Leonardo of Pisa in his book *Liber Abaci* published in 1228.

It is worth noting that the operation of the Abacus, although very simple, requires some understanding of the algorithmic process.

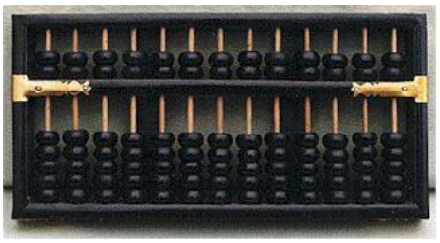

Figure 2.1:

2.2. Aristotle and formal logic

Aristotle was born in Stagirus on the Chalcidic peninsula of northen Grece in 384 B.C. He, more than any other thinker, influenced the subsequent Western intellectual culture[8]. Aristotle studies covered a wide variety of subjects including Politics, Ethics, Physics and, in a broad sense, Philosophy. He is also recognized as the father of *formal logic*.

Aristotle did not consider logic a science itself but, rather, the foundation of every scientific knowledge and, therefore, logic had to be applied to every science. Aristotle believed that all theoretical sciences had to be based on a set of axioms and a set of rules to decide if a given proposition follows from the axioms or not.

In respect to logic, the most important work of Aristotle is his doctrine of the *syllogism*. A syllogism is a simple reasoning scheme that is based on three propositions: a major premise, a minor premise and a conclusion. For example:

Every person is mortal (major premise)
Socrates is a person (minor premise)
Socrates is mortal (conclusion)

The doctrine of the syllogism represent the first known attempt to automate the process of logical reasoning, a science that today is known as formal logic.

Aristotle and his successors derived many different types of syllogisms and organized them into equivalence classes. The system, although, was not without faults and, in fact, Aristotle theory of syllogisms containes defects, including formal defects within the system itself. Moreover Aristotle overestimated the importance of the syllogism over other forms on deductive reasoning. For example Mathematics is deductive but Mathematical reasoning is not based on syllogisms. Even if it is probably possible to rewrite proofs of Mathematical theorems in terms

of syllogisms, this would make them less convincing than they are.

2.3. Blaise Pascal and the Adding Machine

BlaisePascal was a French philosopher, a mathematician and a physicist. He was born in Clermont-Ferrand on June 19, 1623, son of a tax collector.

His father decided that he was not to study mathematics until the age of 15 and all mathematical books were taken out of his home. This certainly sparkled the curiosity of Pascal who started working on the subject by itself. At the age of 15 he presented his first conference paper on geometry.

At the age of 19 Pascal invented the Adding Machine, the first commercial mechanical calculator. By 1752 he produced fifty prototypes of his machine, also called the *Pascaline*, but unfortunatey it did not have much of a commercial success.

The *Adding Machine* was made of metal and wood and was capable of performing simple additions and subtractions between a couple of numbers. Two numbers were inserted in the machine by turning two set of wheels (each representing one digit). The machine would then mechanically compute the sum and represent the result by the position of a third set of wheels.

2.4. Charles Babbage and the Analytical Engine

Pascal adding machine was very simple and was not a programmable computing device, since it was only capable to perform additions.

One other limitation of Pascal's machine and imitations is that the machine had no memory. Output information had to be written down on paper and reenter, if required, at a later time. *Charles Babbage' Analytical Engine* is universally accepted as the first example of a programmable device with memory[9].

Babbage was born in 1791 in Teignmouth, England. He was educated at the University of Cambridge and, in 1816, he became fellow of the Royal Society. Babbage was also member of the Analytical, the Royal Astronomical, and the Statistical societies.

Charles Babbage was interested the problem of computing table of logarithms. One of the major problems at the time was that of determining the *longitude* of a ship at sea. Determining the latitude was easy and it was done by looking at the azimuth of the polar star. Determining the longitude was a much bigger problem because it required comparing the local time, determined by looking at

the position of the sun, with a fixed conventional time, for example, the Greenwich Time. This was difficult. Shipmen did not have a reliable way to determine the Greenwich Time since clocks were not precise enough. The problem was eventually solved with the invention of the spring-based pocket watch. Anyway, for some time, the best technique to determine longitude was for sailors to compute the Greenwich Time by looking at stars. This required quite a lot of skills, including mathematical abilities. The British Royal Astronomical Society started to publish table of logarithms to help sailors with their computations.

To automate the process of building these tables, Charles Babbage, in 1842, designed his

Differential Engine, a mechanical device capable of performing sequences of arithmetic operations on long numbers. It was not a great success; in fact Babbage did not manage to build his machine for lack of funding (Nevertheless, in 1991, his machine was built by a group of British scientists who decided to prove the validity of Babbage's research. The machine was built according to Babbage original drawings and worked flawlessly computing with a precision of 31 digits, according to specifications).

In 1830 Babbage started working on an evolution of his first machine, that he called the Analytical Engine. The Analytical Engine is considered the first true programmable computer[**?**].

Here is Babbage's description of the Analytical Engine:

> *The Analytical Engine consists of two parts:*
>
> *1. The store in which all the variables to be operated upon, as well as all those quantities which have arisen from the results of other operations, are placed.*
>
> *2. The mill into which quantities about to be operated upon are always brought,*
>
> *Every formula which the Analytical Engine can be required to compute consists of certain algebraic operations to be performed upon given letters, and of certain other modifications depending on the numerical value assigned to those letters. There are therefore two sets of cards, the first to direct the nature of the operations to be performed - these cards are called operation cards - the other to direct the particular variables on which those cards are required to operate - these latter are called variable cards. [...] Under this arrangement, when any formula is required to be computed, a set of operation cards must be strung*

together, which contain the series of operations in the order in which they occur. Another set of cards must be strung together, to call in the varibales into the mill, the order in which they are required to be acted upon. Each operation card will require three other cards, two to represent the variables and constants and their numerical values upon which the previous operation card is to act, and one to indicate the variable on which the arithmetical result of this operation is to be placed. [...]

The Analytical Engine is therefore a machine of the most general nature.

The cards Babbage refers to in this passage are paper cards similar to the cards invented by Joseph-Marie Jacquard to program design patterns into weaving machines. These cards had holes to represent the corresponding design pattern. In a modern language we say that *punched-cards* provide the information *storage* for the Analytical Engine.

In the Analytical Engine these cards were used for storing different types of information: the description of the algorithm or program, the input data, the temporary variables, and the output of the computation. At each stage of the computation information was stores in the form of holes in these cards.

The component that Babbage refers to as the *mill* would correspond to what we call today the *processor* or *CPU*.

The Analytical Engine was never built because it was expensive and it was not practical; nevertheless it incorporated many of the idea behind modern computers. Modern computers use electronic memory instead of punch-cards to store information but the concepts of input, output, program and memory variables remained unchanged since Babbage.

A big change that occurred about 100 years later is the introduction of electronics. Babbage's Analytical Engine was a mechanical device containing moving part with all practical limitations that this implies. In particular mechanical machines tend to be bulky, break easily, require a lot of maintenece and their speed is very limited because of friction. Mechanical computiong machines never suceeded in practice.

2.5. Ada Byron and Programming Languages

Ada Byron, Countess of Lovelace was born in 1815 in Piccadilly, England, daughter of the famous poet Lord George Byron. One year after she was born her father

moved to Greece and she never saw him since.

Ada's mother decided to persuade her to stay away from poetry and give her a sound mathematical education. At the age of eighteen, during a party, Ada met Charles Babbage and saw his drawings for the Analytical Engine. She described it as follows:

> *The distinctive characteristics of the Analytical Engine, and that which has rendered it possible to endow mechanism with such extensive faculties as bid fair to make this engine the executive right-hand of abstract algebra, is the introduction into it of the principle which Jacquard devised for regulating, by means of punched cards, the most complicated patterns in fabrication of brocaded stuffs. It is in this that the distinction of between the two engines lies. Nothing of the sort exists for the Difference Engine. We may say most aptly that the Analytical Engine weaves allegorical patterns just as the Jacquard loom weaves flowers and leaves.*

Ada immediately realized the potential of the Analytical Engine and started working with Babbage on a extending the capabilities of the language used to program the machine. Ada also wrote:

> *The Analytical Engine might act upon other things besides numbers, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should also be susceptible of adaptations to the action of the operating notation and mechanism of the engine [...]. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.*

This is a profetic statement, in fact today we use computers to deal with all sorts of information including music which can be stored and processed by the processor.

The language invented by Ada to program the Analytical Engine includes two important programming statements that are common to modern programming languages:

- The *jump statement* (go)

- The *conditional statement* (goif)

Both these statements were also present in the Euclid's algorithms presented in the first chapter of this book. Ada's contribution was to realize the importance of these statements as fundamental building blocks of a programming language.

Much later in 1978, the US Department of Defence developed a new High Level Programming language that was called *ADA* to honor Lady Byron for her eary pioneering work on this field.

2.6. George Boole and Modern Logic

George Boole was born in Lincoln, England, in 1815 (the same year as Ada Byron). His interests were in languages and at the age of 12 he translated and published an ode from the poet Horace. Later he developed an interest in mathematics and algebra and this brought him fame.

Until Boole, since Aristotle, very little progress had been on *formal logic*. Boole, with his book entitled *An an Investigation of the Laws of Thought*, completely revolutionized the field[11]. Boole realized that logic can be understood and defined as a set of algebraic relations among sets of propositions. For example lets divide propositions into two classes: $true$ and $false$. Let's consider two propositions, p and q, each of them can be either $true$ or $false$. We can then derive three sets of propositions:

- $(p\ AND\ q)$ which are true if and only if both p and q are true.
- $(p\ OR\ q)$ which are true if and only if both p and q are true.
- $(NOT\ p)$ which are true only and only if p is false.

AND, OR and NOT are called *logical operators* and have algebraic relations similar to those of arithmetic operators (+, -, *, /). This analogy finally realizes the dream of Ada Byron. Research into automating reasoning (that gave birth to formal logic) and research into automating algebraic computations (culminating in the Analytical Engine) merged together and gave birth to modern Computer Science.

Augustus De Morgan, a famous mathematician contemporary of Boole, described his work with the following words:

Boole's system of logic is but one of many proofs of genius and patience combined. [...] That the symbolic process of algebra, invented as tools for numerical calculation, should be competent to express every act of thought, and to furnish the grammar and dictionary of an all-containing system of logic, would not have been believed until it was proven.

Boolean algebra constitutes the mathematical foundations behind of any modern computing device.

In modern computers logical operators are used at two levels:

- At the *software* level logical expressions are used in programs in conjunctio with **if** conditional statements. For example in the statement "**if** ($x = 1$ **and** $y = 2$) then $z = 3$", where "$x = 1$", "$y = 2$" and "$z = 3$" are assumed to be valid programming language statements that can be either *true* or *false*.

- At the *hardware* level logical operators are implemented as electric circuits and they constitue the basic electronic components used to perform all functions of the processor. In Chapter 4, we will see in some detail as all basic arithmetic operations such as addition and subtraction, can be performed by an electronic devices made out of *logical gates*. A logical gate is an electric circuit that performs the function of a logical operator.

2.7. Alan Turing and the Turing Machine

Aristotle provided one example of logical reasoning. Boole laied a the mathematical foundation of modern formal logic. In the same fashion we can say that Babbage provided us with a design of one particular computing device but it was *Alan Turing* who laied the complete mathematical foundations of modern Computer Science.

Alan Turing was born in London, England, in 1912. Some of his early interests were in the game of chess and in athletic. He was also exceedingly good in mathematics and, during his high school times, he won many mathematics prizes. In 1931 Turing entered King's College at Cambridge to study Mathematics.

Turing became interested in finding a rigorous definition of Algorithm in order to allow one to prove whether there are problem that cannot be described by

an algorithm. In order to confront this problem he invented a class of universal computing machines, now know as *Turing Machines*.

Each Turing machine is an ideal device consisting of an infinite tape (for information storage) and a processor capable of moving along the tape, reading from and writing to the tape. Each individual Turing machine is characterized by the alphabet used to store information on the tape, by the number in internal states of the processor and by its operating rules.

The initial string stored on the tape represents the input data and input program of the machine. The processor begins working by reading the first symbol from the tape. At each step the processor, depending on the value of its internal state and the last symbol read, would perform one or more of the following tasks:

- read from the tape
- move to a different location on the tape
- write to the tape
- change its internal state
- stop

When the machine stops the content of the tape represents the output of the computation. Turing proved with mathematical rigor that all non-trivial Turing Machines are equivalent to each other. Therefore if something is computable by any of the Turing Machines it is computable by all Turing Machines.

All modern computers have been proven to be equivalent to a Turing Machine, therefore all computers are equivalent in respect to what they can do or cannot do (apart for memory, speed and size).

Since the logical thought process can be described with formal logic, logic is equivalent to Boolean algebra and algebraic computations can be carried on by any Turing machine, we conclude that any Turing machine is instructed appropriately can, in some sense, reproduce the logical thought process. This is a very strong statement that deserves more explanations and we will devote an entire chapter to it. For noew we will only state that that for many computer scientists the ultimate goal of this science is, in fact, Artificial Intelligence, although different people give different meaning to it.

The Turing machine was an ideal device (for example was based on a tape of infinite length) and it was not practical to build it. Nevertheless within the next

few years a number of technological advances in the field of electronics made it possible to build of a variety of machines based on similar principles.

During World War II, Turing worked for the British Governmen Code and Cypher Shool and much of his work has been covered by the Official Secrecy Act imposed by the British government. We now know that Turing played a central role in decrypting secret German Codes during the War.

The German employed a machine called Enigma to encrypt their secret messages. It was easy to intercept the communications but it was almost impossible to decrypt them because of the enormous number of possible encryption keys used by the Enigma machine. It could generate almost 1 trillion coding schemes. Turing contributed to the development of the *COLOSSUS*, a 5 by 3 by 2.5 meters machine made out 1800 valves. The purpose of the machine was to read German encrypted messages and decrypt them at the speed of 5000 thousand characters per second. Turing and the COLOSSUS played a central role in the British intelligence that, ultimately, lead the British-US coalition forces to win World War II.

2.8. John von Neumann and Modern Computer Architecture

John von Neumann was born in Budapest, Hunagry, in 1903 and he was a contemporary of Alan Turing [13]. John von Neumann pursued research in many fields including mathematics, quantum physics and finance[14]. He received a Ph.D. in Mathematics from the University of Budapest, lectured at Berlin and Hamburg and in 1930 he moved to Princeton where, in 1933, became one of six mathematics professors at the Institute for Advanced Studies (Einstein was also one of the six).

Von Neumann is the inventor of the modern computer architecture[**?**]. Like Babbage's Analytical Engine, the Turing Machine was an ideal device, not situable for practical purposes, so von Neumann proposed a new alternative architecture that corresponds to the one of the modern computer.

He envisioned a device made of the following four components: *CPU* (=Central Processing Unit), *Memory* (or storage), *Input* and *Output*.

Von Neumann realized that there is no need to distinguish storage for variables and storage for the program; they can both be stored in different position within some *linearly addressable memory*. The concept of linearly addressable memory is that of a memory divided into cells. Each memory cell would have a unique

number known as *memory address*. The CPU would then be able to read from and write into each of the memory cells by specifying its memory address. One important idea that inspired von Neumann design is the possiblility to number all words in a programming language therefore a program would consists in a series of numbers that can be represented and stores in electrical form.

The role of the CPU is similar to that of the processor in the Turing Machine but, instead of moving around the tape, von Neumann's CPU had the ability to address a memory location by sending an electrical signal that represents the location of that memory cell. The CPU would then be able to read and or write numbers (also represented as electirc signal) into that memory cell. In the same fashion as the CPU could read from and write to memory it could address and write to the input (for example a typing device) and read from the output (for example a printer or a computer screen).

The role of the input device is to read commands and information from a user. The role of the output device is to present information to the user.

2.9. Kurt Godel and the limits of computation

Aristotle had said that any science had to based on logic on an aximatic basis. One major attempt to put the whole Mathematics on an aximatic system was devised by *Bertrand Russell* in his book *Principia Mathematica* written in 1910[15]. His attempt did not succeed for reasons we will explain below. The most famous mathematician of the time, *David Hilbert*, also envisioned the possibility of rewraiting the entire Mathematics as an exiomatic system and challenged his collegues to define a set of axioms and a set of rules (or procedures) that would allow a human being as well as a machine to take a proposition, apply mechanically the rules, and determine without ambiguity if the proposition is a true mathematical statement or not.

This school of thought gave origin to a phylosophy known as logical positivism. Positivists hoped to remove any metaphysical element from phylosphy and redefine the discipline purely based on logic.

Kurt Gödel was born in 1906 in Brünn, Austria. He entered the Univerity of Vienna to study mathematics in 1923. In 1931 he publish his proof of what is today know as *Gödel's Incompleteness Theorem*. He proved that Hilbert's dream could be realized, thus setting a limit to what automation can achieve. In 1934 Gödel emigrated to the United States, where, at Princeton he developed an interest in General Relativity and became a close collaborator and frend of Albert Einstein.

Gödel's work should not be reguarded as a negative conclusion or as shading a dark cloud over years or successes. Gödel's theorem deals with the concepts of infinity that always troubled logicitians and mathematicians. The main point of Gödel's Incompleteness theorem is that in any formal, aximatic, system that are proposition that do belong to the system and are true (in the sense that they follow from the axioms after application of the rules of the system) but the number of steps required to prove their thougth is infinity. Hence there are statement in any formal exiomatic language that cannot be proved to be true by a man or by machine simply following the rules of the language itself.

Gödel's theorem has some practical implications. For example, it is impossible to write a computer program that takes as input another computer program and tests with a success rate of 100% if the input is a valid program. By a valid program we mean a program that, once executed by a machine, reaches a stopping point within a finite time.

Another remarkable result achieved by Gödel is the proof that all axiomatic languages are equivalent to each other. One practical consequence of this statement is that given any axiomatic language we can map its statements into aritmetic statements. The proof that the original statement is true is equivalent to a mathematical proof of the equivalent aritmetic statement. Powerfull, is in it?

One question remains open after Gödel's work. Can the human mind be able to perfrom any kind of computation that cannot be performed by a machine? Even if this were true, Gödel's theorem tell us that we would not be able to prove it within a formal aximatic system and therefore the answer to the above question layes outside the domain of science.

Gödel received the US National Medal of Sience in 1974.

2.10. Grace Hopper and high level languages

Many of the scientific developments we described so far are to be considered conceptual development. At this point the concept of computer was clear in the minds of scientists but computers had to be built out of electromechanical switches (called realys) and this made them big and expensive. Some of the first such computers were the *Mark 1* (designed at Harvard Univerity and built between 1939 and 1944), the *COLOSSUS* (built by the British Intelligence in 1940), the *ENIAC* (Electronic Numerical Integrator And Calculator, developed in US in 1944 by Presper Eckert and John Mauchly) and the *EDAVC* (develped with the collaboration of von Neumann at Cambridge University in 19XX). All these

computers were huge and not at all user friendly. They were almost exclusively used for research and military purposes.

Every machine, including early microproessors, can only understand one and only one language called *machine language*. There is not a single machine language since each machine has its own machine language that depends on the characateristics of the device. In a machine language each word is represented as number (according with von Neumann design) and it correspond to one of the most simple arithmetic operations (+,-,*,/) a logical operation (AND, OR, NOT) or a typical program control flow statement (**if** and **go**).

Programming in a machine language is difficult and presents many inconveniences, for example: programs are difficult to read and to reuse, programs depend on the particular machine they are developed and cannot be used on different types of machines. Not to mention how easy it was to introduce errors (also known as *bugs*) in the programs.

In 1950 *Maurice Wilkes* had introduced the *assembly language*, a language very similar to machine language that used English words to represent instructions in the language rather tha numbers. Assembly language reduces the possibility of transcription copy of a program but does not address any of the major isses with the machine language.

What was needed was a way to automate the translation from a hight level programming language easliy understandable by humans into the machine language understanble by the computer. This translation could be done by the computer itself by excuting a translation program written itself in machine language. A program of this kind is called *compiler*.

Grace Murray Hopper created the first compiler. She was born in New York City in 1906. Hopper earned a Ph.D. in Mathemtics from Yale in 1939 and 1943 joined the U.S. Naval reserve. In 1949 she worked for the Eckert-Muchley Computer Corporation which produced the first comercial computer, the *UNIVAC*, developed by the same team that invented the ENIAC. There Hopper realized the possibility of programming computers with commands written in English rather then in some almost incomprehensible numerical notation and 1952 she wrote her first paper on compilers. Her first compiler was called the *FLOW-MATIC* and, by the end of 1956, it could understand twenty English statements.

The FLOW-MATIC and other high level languages differ from assembly languages since they are not just a literal transaltion into English of machine language statements. They have a syntax that is very different than machine language or assembler. For example they include words that correspond to data structures

more complex than a reference to an individual memory cell. They also support syntactic expression in order describe how to manipulate these data structures. The simplest type of data structure is the *string*, a sequence of characters. In the memory of the computer each characters is represented as a number and stored its own individual memory cell in the form of an electric signal. Low level language such as machine language and assemply do not have a mean to refer to a string. They can only manipulate the invidual characters in the form of numbers. High level languages support expressions able to manipulate a string as it were a fundamental object. The compiler translates operations on the string into operations on the individual characters.

The natural evolution of the FLOW-MATIC was the *COBOL* programming language, which appeared in 1959. Hopper contributed to development of COBOL and its standard manuals and documentations.

It is interesting that Hopper retired from the Naval Reserve at the age of 60 but, she was so indispensible that, in less than seven months. She was the first woman to be ever recalled to duty. She was also the first woman Distinguished Fellow of the British Computer Society. In 1983 Hopper was promoted Commodore by Presidential appointment.

Here is an example of a COBOL program that tells the computer to display the string "Hello, World."

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.     HELLOWORLD.
000300 DATE-WRITTEN.   01/16/03        00:00.
000400*       AUTHOR    MASSIMO DI PIERRO
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
000900
001000 DATA DIVISION.
001100 FILE SECTION.
001200
100000 PROCEDURE DIVISION.
100100
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100400     DISPLAY '' '' LINE 1 POSITION 1 ERASE EOS.
```

```
100500     DISPLAY ''HELLO, WORLD.'' LINE 15 POSITION 10.
100600     STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800     EXIT.
```

Any COBOL program is composed of five divisions:

- IDENTIFICATION DIVISION containing a description of the program and its author.
- ENVIRONMENT DIVISION containing a description of the machine the program is designed for
- DATA DIVISION containing a description of the data types used by the program
- FILE DIVISION containing a description of the files used by the program (a file is a collection of data stored on a disk or other storage device).
- PROCEDURE DIVISION containing the instructions, the actual program

Modern computer languages do not require programs to be composed of five divisions and consist only of a procedure division.

Moreover each line in a COBOL program is uniquely identified by a line number the order the different statements are to be read by the machine. Each statement terminates with a fill stop.

Notice the statement at line 100500: it tells the system to DISPLAY the string of character "Hello, World." at the screen coordinates $row = 15$ and $column = 10$. Despite the use of English words, all programming languages have a very strict syntax. For example replacing the word DISPLAY with SHOW would not work. DISPLAY is a *keyword* of the COBOL language, SHOW is not. Compilers refuse to compile something that is not syntactically correct according with the rules of the language. Programmers tend to make this kind of mistakes and compilers usually respond with a statement like "*syntax error*".

A compiler program translates COBOL programs into machine langauge programs, therefore COBOL programs are (almost) independent on the particular computer architecture in use. For this reason many COBOL programs survived until present day.

It is interesting to note that to transalate from COBOL or any other programming language other then machine language one needs a compiler and a compiler

has to be written in some programming language. It is natural to assume that the compiler has to be written in machine language. While this was true for the first compilers this is not true for modern compilers. Typically, when a new programming language is invented the new compiler is written in one of the already existing programming languages and then itself compiled into machine language. Eventually the new language is used to write the compiler for the language itself and the new language becomes an autonomous object capable of compiling itself.

This characteristic of programming languages has remarkable analogies with living systems. In biological terms the language that describes every life form that we know is *DNA*. DNA is translated into proteins, the building blocks of living organisms, by the ribosomes. Although one needs ribosomes to begin with, DNA contains instructions to fabricate the ribosomes. Living organisms are made of living cells and each cell contains DNA and ribosomes. In computational terms we can say that the role of the ribosomes is that of executing the program written in the language of DNA. Every function of the cell, including its own duplication process (reproduction) can be understood in computational terms[**?**].

2.11. Transistors and Microprocessors

A big technological progress was achieved in 1947 when *John Bardeen*, *Walter Brattain* and *William Shockley* at *Bell Labs* invented the *transistor*, for which they received the Nobel Prize in 1956. The transostor is a small device made of silicon or germanium that can be used for two purposes: to amplify an electrical signal and as a switch. The first purpose is tipically exploited in transostor-based radio devices and amplifiers. The second purpose is applied to digital devices as basic component for logical gates. The CPU of modern computers is made almost exclusively out of transistors.

In 1971, *Federico Faggin* at *Intel* created the first commercial *micropocessor*, called *Intel 4004*, i.e. a single piece of silicon device (*chip*), containing an entire processor and made exclusive out of logical gates, implemented with transostors. The Intel 4004 contained 2300 transostor and was able to execute about 60'000 elementary instructions per second. By instructions we mean an elementary instruction such as an addition, a subtraction, evaluate a logical expression, execute a jump in the program. At 1/8th inch wide by 1/6th inch long this small device had as much computer power as the 3,000 cubic feet *ENIAC* developed only 20 years earlier. Federico Faggin signed the chip with his own inititials.

The invention of the microprocessor marks the beginning of what we can call

the digital age. An age in which computers are cheap, small and fast. If we just consider arithmetic operations, today the computational power of machines exceeds the computational power of the human brain.

Since the Intel 4004 computer power has been doubling every 18 months. *Gordon Moore*, founder of Intel Comrporation, was the first to observe this that became known as *Moore's Law*. There are claims that Moore's law may eventually break down in the near future since the miniaturization process has reached such an high degree that effects of quantum physics are coming into play. In practice there is no evedince that Moore's low is breaking down and if new laws of physics will be required to design future generations of computing devices than so be it.

Accoring to official sources in 2003 Intel has shipped its 1 billionth computer chip.

2.12. From the punched-cards to brain implants

Early computers until the late seventies, had very poor input output devices. The first type of input/output device was the punched-card reader/writer. The idea of writing information by punching holes in a card was already in the Babbage original design of the Analytical Engine.

The keyboard was originally introduced as an evolution of the old fashion typing machine and used to the purpose of punching cards. Later it was connected directly to the computer so that the computer could read the input directly from the keyboard instead of the punched-card. In the same way the first printers were similar to ordinary typing machines but they were controlled by electric signals originated from the computer.

The computer screen was a big leap in output technology. At the beginning of the nieteen century particle physicists started to unravel the nature of matter and its structure in terms of building blocks (particles). Physicists started to built accelerators that extract particles from matter and accelerate them against a target. They also built particle detectors, i.e. materials and sensor that, when crossed by a charged particle, emit light or send an electrical impulse to a reading device.

The TV screen and the *computer screen* are two similar applications of particle accelerators. The core of a computer screen is a *particle accelerator* that uses an electric field to extract electrons from a hot *cathode* and accelerate them to form a beam. The beam points towards an array of small detectors that emit light when struck by the beam. The image is designed on the screen by moving the direction

of the beam of the electrons. In the case of an ordinary TV screen, signals received via radio or cable direct the beam. In the case of a computer screen, the computer itself according with its program sends signals to the screen and direct the been. An algorithms transforms numbers representing characters in the memory of the computer into numbers representing the image corresponding to those characters.

The first personal computer, comprised by a small CPU, a computer screen and a keyboard was developed by Steve Jobs and Steve Wozniak, founders of Apple Corporation. Subsequent models, based on similar design, were developed by *I.B.M. Corporation*. The First IBM computer, called *IBM Personal Computer*, was equipped with a programming language called BASIC that made the machine easy to use and therefore popular.

A more modern type of computer screen is the *LCD screen*. It uses polarized liquid crystals to transform electric signals into images. LCD screens are replacing ordinary screens for the good. In fact computer screens are the most energy consuming part of ordinary Personal Computers. In Japan alone in 2001 70% of Personal Computers used LCD screens with a total energy saving equivalent to two nuclear reactors.

Another very common input device is the *mouse*. The mouse if a jbox on wheels that detects the movement of its own wheels and sends electric signals to the computer representing these movements. The computer reads the signals and calculates the positions (or more precisely the shift in position) of the mouse.

Other input/output devices are available although less common, such as 3D mouse that use optical signals to transmit its position to the computer and 3D glasses that project two different computer generated images to the two human eyes thus giving the illusion of a stereoscopic vision. Many of these modern devices are not common because they are expensive and because existing computer programs do not take advantage of their capabilities.

In this book we often identify the computer with the CPU and the memory and we treat Input/Output devices as external. This is somewhat in contrast with ordinary experience. In fact, in our ordinary life, we often deal with personal computers that have a keyboard, a mouse and a screen and we identify the computer with them, rather than the CPU in the box behind the desk. Nevertheless it is very important to keep this distinction since not all computers are personal computers. Many modern supercomputers, for example a *Cray T3E* of the IBM *Blue Gene* (in figure) do not come with ordinary input output devices. They are connected to a network and communicate with the users through other computers also connected to the network. The reason is that some computers run operating

programs that allow them to manage multiple input/output devices and multiple users at the same time.

What will they think of next? Neuroscientist *Philip R. Kennedy* and neurosurgeon *Roy E. Bakay* have developed an electrode brain implant that allows speech-impaired patients to communicate through a computer. This device is still very experimental and in its early stages but, in principle, it could help patients affected by *amyotrophic lateral sclerosis* to interact with a computer and eventually, through the computer, to speak with other people. Similar implants will help paralized people to walk by controlling robotic limbs. The idea behind this kind of research is that a computer can inteact directly with the human brain by reading its electric signal. A computer program interprets those signal and reacts by moving a robotic limb or performing some other action. The most difficul part of the process is in writing a computer program that interprets electric signals in the human nervous system. In fact, this requires a deep undestaning of how this system works. Moreover the human nervous system may have characteristics that differentiate one individual from another individual and, perhaps, there is a need for a computer program that can learn the neruological pattern of each individual patient.

2.13. The modem and the Internet

In 1968 the *Advanced Research Projects Agency* (ARPA) within the U.S. Ministry of Defence initiated the first computer network, *ARPANET*. Finally computers could exchange data with each other without a physical human intervention. By 1971, 23 computers were connected to each other through ARPANET. In 1972 ARPANET went public and soon after other computer networks appeared.

The *Internet* the largest network of networks of computer. Not all computers are connected to a network but many business computers and virtually all academic computers are connected to some network. At some point the owners of many of these networks decided to join them together and allow signals from a different network to travel through their own. They agreed on a common naming scheme so that every computer connected would have a unique identifying number (called the *IP number*) and they also agreed on a set of common protocol that allow computers to automatically re-route signals and process them according with the type of information that they represent (the *TCP protocol*), for example an email message, a web page, or a less usual type of information.

This network of networks was then called the Internet, as opposed to the *In-*

tranet, term used to refer to smaller local networks. No single entity has ownership of the Internet although each computer or network that joins it is owned and managed by somebody. The Internet communication protocols and domain names are established by international no-profit organization created for such purpose.

The TCP/IP protocol was officially proposed in 1974. The idea behing the protocol was that:

- Each network should be able to work on its own and requiring no substantial modification to join the Internet.
- The Computers that manage Internet traffic would retain no information about the traffic in transit (for speed reason and for privacy of the users).
- The information in transit from a source computer to a destination computer would be split into packages and each package would be routed through the fastest route available, independently on the other packages.
- The design and the protocols used by the Internet would be freely available at no cost.

In 1977 *Dennis Haynes* invented the PC *modem*. A device that transforms computer generated digital electric signals into analog signals, of the type that can travel over phone lines, and vice versa. The modem opened the possibility for everybody with a Personal Computer and a phone line to connect to the Internet.

When we navigate the Internet and connect to a web site, what actually happens is that our Personal Computer is using a modem to transform digital signals into analog signals that travel over the phone line. At the other line of the phone line another Computer converts those signals back to digital and re-routes them through the Internet to the Computer that serves the web site we are visiting. Therefore, in some sense, we are using the console of out Personal Computer to interact with a remote server computer.

The Internet was originally used by academic, research and private institutions to connect to each other computer and exchange data. The popularity of the Internet exploded in 1991 when Physicists at *CERN*, the research lab that hosts the largest particle accelerator in the world, developed the *HTTP protocol* and the *HTML language*. These are a very simple communication protocol (HTTP) based on TCP/IP and a language to describe documents (HTML). HTTP and HTML were simple enough and free to become the standard for posting documents on the Internet.

Also in 1991 the U.S. vice President *Albert Gore* introduced the *High Performance Computing Act* (known as *Information Superhighway*) that funded much of the subsequent telecom infrastructures required by the Internet. In a speech presented at the International Telecommunications Union in Buenos Aires, in 1994, Gore quoted these profetic words of the writer *Nathaniel Hawthorne*:

> *By means of electricity, the word of matter has become a great nerve, vibrating thousand of miles in a breathless point of time [...] The round globe is a vast [...] brain, instict with intelligence!*

These words were originally written by Hawthorne in 1951 and inspired by Samuel Morse's invention of the telegraph.

Today, wherever I am, I can connect my laptop to the Internet using a wireless modem (also known as *Wi-Fi*) and order almost any product that is on sale anywhere in the world. Moreover I can connect to any of my offices and use any of their computers as if they were sitting right in from on me.

Finally Information technology has created a new economy that, despite its up and downs, has fueled a service sector that, in the United States alone, in 2002, accounts for 80% of the total Gross Domestic Product at the expense of only 2% of the total energy consumption. This new economy is an enormous potential for growth since it is not capped by the planet limited resources. In fact human ability to generate ideas and produce information has, in principle, not limit. In 2001 the Information Technology sectors plunged and it is was so relevant to the world economy that the entire economy plunged with it.

2.14. The future and Artificial Intelligence

In 1997 a computer known as *Deep Blue* (a standard IBM RS/6000 supercomputer with 32 processors) programmed by a team of five computer scientists confronted World Champion *Garry Kasparov* in a game of chess under tournament conditions. Kasparov resigned 19 moves into Game 6, handing a historic victory to Deep Blue. Apparently Kasparov commented the game by saying the "the computer is very human" and "I get scared of something beyond my comprehension".

One the one side one should not overestimate the meaning of this victory of the machine over man. The machine was executing a computer program exclusively designed to play chess[21]. The program encoded a lot of knowledge about the game gathered by the best chass players and it was written by human programmers.

On the other side, the game of chess, traditionally, for centuries, has been considered a game that requires intelligence. At the time of Pascal or Babbage, the idea of machine that could beat man in the game of chess would have been unthinkable. Nevertheless scientists have succeded in reproducing the thinking process behing the game of chess into an algorithm written in some programming language and a machine (not so advanced by modern day standard) executed that program flawlessly (as only machines can do) and won.

The computer is a wonderful machine but I believe we have seen only very little of its potential.

2.15. Chronological Summary

- 3000 BC. Invention of the Abacus in China.
- 370 BC. Aristotle thought about automate logic.
- 1642 AD. 19 years old Blaise Pascal invented and build the first mechanical adding machine.
- 1830. Charles Babbage invented first true mechanical computing machine, the Analytical Engine.
- 1842. Ada Byron invented what is first programming language for the Analytical Engine. She indetified the basic types of programming steps.
- 1847. George Boole invented modern mathematical logic, foundation of every modern computer.
- 1935. A British mathematician, Alan Turing, lied the theoretical foundations of Computability theory. He proved that all computers are equivalent.
- 1940. John von Neumann invented the modern computer architecture: CPU, Memory, Input, Output
- 1945. John Mauchley and J. Presper Eckert developed the first successful general purpose digital computer, called ENIAC.
- 1947. Bardeen, Brattain and Schockley, invented the transistor. It made possible reducing computers'size and increasing their speed.

- 1952. A mathematician and Navy Officer, Grace Murray Hopper, developed the first compiler for the first commercial computer (the UNIVAC).
- 1968. Federico Faggin designed the first commercial computer on a chip, the so called microprocessor, model Intel 4004.
- 1968. ARPA created the first computer network.
- 1976. Steve Jobs and Steve Wozniak created the first commercial Personal Computer.
- 1977. Dennis C. Haynes invented the PC modem, that allows computers to communicate over phone lines.
- 1991. The Internet started.
- 1997. A computer program beat the human World Chess Champion.

3. Software

This chapter is about software and our goal is to understand the process of developing algorithms that solve simple problems[22]. An algorithm is a sequence of steps simple enough that can be programmed into and executed by a machine. In order to define a step we must have in mind a specific machine.

We will consider an ideal machine constuted by a processor and two memory sectors, one containing the program to be executed (we call this program memory) and one containing memory cells that can be used to store temporary values produced during the execution of the program (we call this temporary memory). In our machine the processor is also connected to the the output (the screen) and the input (the keyboard). The processor or CPU of our simulated machine contains three components: a *counter*, a *stack register* and an *Arithmetic Logical Unit* (ALU).

In the next section we briefly introduce these components. In next chapter we present a possible hardware implementation of these three devices.

3.1. The counter

Each processor, or CPU, includes a counter. This is a location of memory built in the CPU itself that contains the memory address of the next instruction to be executed. When the CPU is turned on the counter is set to zero and the CPU read the instruction stored in memory cell number zero and executes it. At each stage, after each instruction is read and before it is excuted, the counter is automatically incremented on one unity, so that next instruction is read and executed from the memory. And so on and on.

We have not examined yet what constitutes a valid CPU instruction and we postpone this discussion to a later section. For now it is imporatant to realize that any machine language include one (or more) instruction to change the value of the counter. In our simulated language there are two of such instructions: `go` and `goif`. If the `go` instruction is read from the memory, the CPU changes the

value of the counter to the value stored at the top in the processor *stack register*. If the `goif` instruction is read from the memory, the CPU changes the value of the counter to the value stored at the top in the processor *stack register* if and only if the next value store into the stack register is different from zero.

These machine language commands correspond to the jump and the conditional statement first proposed by Ada Byron. We will later see how thay are used in a program.

3.2. The Stack

Every CPU contains an internal storage used to contain values to be read from or written to the memory. This storage area is called CPU registry. Our simulated CPU, in this respect, is more sophisticated than ordinary CPUs, which do not includes a stack registry. The best way to visualize a stack is a pile of papers on a desk. We can add one sheet of paper to the top of stack and we can remove a sheet from the top the stack. We call *push* the operation of adding a sheet of paper and *pop* the operation of removing a sheet of paper. A registry stack is an electronic device built into the CPU that stores numbers (in electronic form) and has similar properties to the paper stack. The CPU can push a number into the stack and we can pop a number from the stack. For example if 2,5,3,6 are pushed into the stack and one then pop four numbers one gets 6,3,5,2, i.e. the same numbers in opposite order respect to the order they were pushed in. The concept of a stack is crucial to understand functions and procedures but we postpone this discussion to a later chapter.

3.3. Arithmetic Logic Unit

The ALU is another component of our CPU. It is an electronic circuit that can pop numbers from the CPU stack registry, perform simple matematical operations on then and push the result back into the stack. For example if the CPU reads from the program the instruction `sum`, this activates the ALU that pops two numbers from the CPU stack registry, sums then and pushs the result int the stack. Our ALU is very simple and can only perform simple arithmetic instructions such as additions, subtractions, multiplications, divisions and logical operations. We will study each of them in more deatil later.

Modern computers can have very sophisticated ALUs capable of performing advanced mathematical computations such as vector rotation. At this point it

important to stress that our simulated machine, despite beeing very minimalistic, is completely equivalent to a Turing Machine and is able to compute everything that a core sohisticated computer can do. A more powerful ALU makes the machine faster but not more powerful from a computational point of view.

In one of the next examples we see how to write a computer program that computes exponentials and table of logarithms (the goal of Charles Babbage) using only the four basic operations. Actually our ALU is able to compute exponentials (since we consider this to be a simple operation) but, as we mentioned, even if it did not our simulated machine would still be a general purpose programmable machine.

3.4. Program memory

The program memory is an electronic device divided into memory cells. Each cell is indetified by a number, the memory address of the cell, and it contains a value, stored in some electronic form. The value stored in each cell is a number that represent one of the possible machine language instructions. For example we have see the `go` and `goif` instructions. Each of these two instructions corresponds to a number, let's say 1 and 2 respectively. If the program memory contains a program that uses these instructions then some of the memory cells occupied by the program instructions will contain the numbers 1 and/or 2. At this point we are not concerned of where and why. We will be able to answer these questions later in the chapter.

3.5. Temporary memory

The temporary memory is the very similar to the program memory. It an dentical electronic device divided into memory cells. Each cell is identified by a number, the memory address of the cell, and it stores a number in electronic form. The only difference between the program memory and temporary memory is that we use the first to store numbers that represent our description of the program, while we use the latter to store numbers that correspond to temporary values arising at intermediate steps in the computation.

For example let's say we want to compute 3+4+5. The CPU has to perfrom tha sum in some order. Let's say we choose the following order (3+4)+5 which means that first 3 is to added to 4, then 5 is to be added to the previous result. The result of 3+4 is a temporary value and it has to be stored somewhere. For

very simple calculation we can use the CPU stack registry as storage but, for complex calculations, we may need a bigger storage space. The temporary memory provides this space. The temporary memory has one more advantage over the CPU stack registry. The latter only allows us to push and pop numbers. We cannot retrieve a number that is stored deep into the stack (think of the problem of finding a particular sheet of paper in a big pile of papers). The temporary memory is more structured since each cell has its own address. The CPU has the ability to read from and write to each individual cell. Therefore when a program has to store a numeric value it can store it in a particular cell and it only has to remeber the address where the value is stored. At this point it may not be obvious to a reader why remembering an address is better the remebering the value storead at that address. This will be more clear in a while. In general there are three reasons why this is better:

- We may want to store more than a single item, for example a string of characters. If we store them sequentially (one after the other) in the temporary memory we only have to remember the address of the first of them and this is more compact than remenbering each invidual character.

- We can give names to those memory addresses that contain value we may want to access explicitely. Names are easier to remember than numbers. For example we can store the value of π at memory address number 7634 and, by convention, we call this memory address *pi*. A memory cell with a name is called a *variable* and its name is called *variable name*. It is much easier to remember a variable name than its value. Moreover the value of a variable may change with time because the CPU writes into it according with the program and we do not necessarity have to track these changes.

- In practice we are going to use high level languages and a compiler translates our programs into machine language. The compiler itself is a computer program. One of the jobs of the compiler is that figuring out which temporary values are produced by the program and finding space in the temporary memory to store them. The compiler uses an algorithm to figura out what is stored where and the programmer, in the majority of cases, does not need to know it. Some high level languages such as Python and Java do not even allow tha programmer to query about the memory address of a variables. Some other languages such as C and C++ do allow that.

In practice modern computers follow the von Neumann architecture and this means that program memory and termporary memory correspond to different locations into the same device called *RAM memory*. We keep the distintction only for didactic purposes.

3.6. Machine language

Our CPU is able to perform the following tasks:

- Read an instruction from the program at the location spacified by the CPU counter.
- Load and push a constant value into the CPU stack regitry (`load`, `loadc`)
- Read a number from the memory into the CPU stack registry (`read`)
- Read a number from the keyboard into the CPU stack registry (`input`, `inputc`)
- Write a number from the CPU stack registry to the memory (`write`)
- Write a number from the CPU stack registry to the screen (`print`, `printc`)
- Perform aritmetic operations on the number in the stack (`sum`, `sub`, `mul`, `div`, `mod`, `pow`, `exp`, `log`, `sin`, `cos`, `neg`, `int`, `float`, `dup`, `swap`,`dump`)
- Perform a local (Boolean) operation on the numbers in the stack (`and`, `or`, `not`, `le`, `eq`, `gt`)
- Change the value of the CPU counter (`go`)
- Change the value of the CPU counter if a given condition is true (`goif`)
- Stop program execution (`stop`)
- Do nothing an execute the next instruction (`pass`)

The first task is performed automatically by the CPU. The other tasks are performed when and only when the corresponding instruction, shown in brackets, is read from the program memory.

Some of the those instructions are required by our machine in order to be equivalent to a Turing Machine. Fifteen of these instructions are:

```
load read write sum sub mul div
mod and or not le eq go goif stop
```

Other seventeen instrutions are not required but are there for convenience:

```
loadc input inputc print printc neg pow exp
log sin cos int float dup swap dump gt pass
```

We will examine each of them in turn later in the chapter. Each of these instruction is here written for convenience using an English word. In practice each instruction is a number stored into the program memory.

One may wonder about the need for an instruction that does nothing (pass).

pass The pass instruction instruct the CPU of our MS to do nothing and execute the next instruction in the program.

This instruiction is useless but it included for convenience, in fact, we will assume that our program memory is filled with pass instructions. We can also assumed that the pass instruction corresponds to the number 0 so that the program memory is initially filled with zeros. At some point we write our program in the program memory. The convenience of the pass instructions is that those memory cells that do not contain a valid program instruction contain pass instructions and therefore they are skipped during the program execution.

We will refer to the above set of instructions as the Minimal Machine Language (MML).

3.7. The Flow Chart Runner

Programming in a machine language, including the MML, can be hard. For this purpose the book is accompanied by a compuer program called *Flow Chart Runner* (FCR). This program is composed by two parts: a *Flow Chart Designer* (FCD) and a *Machime Simulator* (MS). The FCD allow the user to write a program by visually designing an algorithm as a flow chart. A flow chart is a diagram made

of geometric shapes, in our case ellipses, and each ellipse contains an algorithmic statement. The ellipses are connected by arrows that describe the control flow of the program. So, for example, if an ellipse contains the instruction "do this" and another ellipse contains the statement "do that" and one arrow points from the first to the second statement than "do that" is executed rigt after "do this" is executed. The FCD allow us to design a program and also to translate it automatically (compile it) into the *Minimal Machine Language* (MML). The Machine Simulator (MS) than simulates the machine described in this chapter and executes the compiled program. It is also common to say that the computer runs the program, rather then executes.

The FCD has to be undesrtood as a tool for us to learn the MML and not as a programming language in itself, despite the fact that it actually is a nice programming language. In fact our ultimate goal is to learn to break the solution of a problem, in the form of an algorithm, into the simple steps that can be executed by our machine. Details of how to use the FCD and the MS are delegated to Appendinx A.

From now on we will assume the reader is familiar with details of the program is able to reproduce the programs described in the following examples.

3.8. The first program: do nothing and stop

Our first program is a "do nothing" program described by the following Flow Chart (FC): hello.fc

(3.1)

The program starts and tells the machine to stop. This program is easy to translate into machine language. It consists into a the single MML instruction stop. All instructions in the MML are to be stored at some memory address and, by convention, program execution starts at memory address number 0. therefore here is the compiled program:

```
# our first program
000000: stop
```

We adopt the following convetions:

- Everything following the pound sign (`#`) until the the end of the line if a comment; that is a comment meant for humans and not stored into the

memory of the machine. In the program above the whole sentence "# our first program" is a comment.

- We write each program instruction preceded by the memory address where it is to be stored. The memory address is written as a six digits decimal number. The program above has only one instruction, `stop`, stored at memory address 000000. The point where program execution conventionally starts.

stop The instruction stop tells the CPU to stop inceremented the CPU counter and do not execute the a instruction.

When we turn our machine on (we start the Machine Simulator) the CPU counter contains the number 0. The machine the automatically reads the instruction stored in the memory cell having the address 000000. That memory cell contains some numeric representation of the stop instruction. The CPU executes the stop instruction which means it ceases to increment the counter and does not procede further. The program excution has stopped.

The program above is useless but it teaches us how to a program is stored into the memory. For the first part of this chapter we stick to numbers and numerical computations. Later in the chapter we will see how to perform computations that involve characters and strings of characters.

3.9. The second program: store a number in the temporary memory

By clicking on the first circle below the "start" statement of the FCD we can add a new statement. We add the statement "a=4" and we produce the followoing algorithm: (a=4.fc)

(3.2)

The algorithm above performs the following operations"

- Reserve a memory cell in the temporary memory
- Call this cell with the name "a"
- Write the number 4 into the memory cell "a"
- Stop the execution of the program

We have already seen how the stop statement is compiled into the MML instruction stop. We need to see how the statement "a=4" is compiled into our MML. Here is the compiled program:

```
### a=4
000000 : load
000001 : 4
000002 : load
000003 : 1000 # addr. a
000004 : write
### stop
000032 : stop
```

The first five lines (from 000000 to 000004) correspond to the FC statement "a=4". The sixth line (number 000032) corresponds to the FC statement "stop". What happened to the memory cells in between (from 000005 to 000031)? As we mentioned before we are assuming that they contain pass instructions and therefore will be ignored by the CPU. Alterantively we could just place the stop instruction at line 000005. Anyway there is a practical convenience in using the pass instructions.

In order to undertand the above compiled program we have to undestand what the individual MML instductions do:

load The load instruction tells the CPU to read next program line and push its content into the stack.

For example, in the program above, when the load insstruction at line 000000 is executed it tells the CPU to push the number 4 into the stack. The CPU does not treat 4 as a command but as a number. The program counter is automatically increased of two unites and program executions resumes at line 000002. Analogously the load instruction at line 000002 tells the CPU to push the number 1000 into the stack and program execution resumes at line 000004.

write The write instruction tells the CPU to pop two numbers from the stack, x and y, and to store the number y at the memory cell whose address is number x.

In the program above the write instruction al line 00004 tells the CPU to pop 1000 and 4 from the stack and to store the number 4 at memory cell whose address is number 1000. Each variable in memory must be assigned to a corresponding unique memory address. In typical high level programming languages the compiler (in conjunction with the operating system of the machine) deals with this problems and the user is rarely interested in how variable names are associate to memory addresses.

The comment "# addr. of a" is just a comment for us and it has no meaning for the machine.

The FCD associates variables names to memory addresses according to a very simple rule. Only 16 variable names are allowed

$$\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}, \texttt{g}, \texttt{h}, \texttt{i}, \texttt{j}, \texttt{k}, \texttt{m}, \texttt{n}, \texttt{p[i]}, \texttt{q[i]}, \texttt{r[i]}$$

and each of them is always associated to the same memory address according to the following table:

variable name	variable address	variable type
`a`	1000	number
`b`	1001	number
`c`	1002	number
`d`	1003	number
`e`	1004	number
`f`	1005	number
`g`	1006	number
`h`	1007	number
`i`	1008	number
`j`	1009	number
`k`	1010	number
`m`	1011	number
`n`	1012	number
`p[`i`]`	2000+i	array of numbers
`q[`i`]`	3000+i	array of numbers
`r[`i`]`	4000+i	array of numbers

The last three variables (p,q and r) are of type array and their use is different than the variable "a" introduced in this section. We will discuss arrays in a later section. All other variables contain numbers.

We now have all elements required to understand the compiled program above:

- Instruction at line 000000 tells the CPU to load next program line and push its value (4) into the stack.
- Line 000001 does not contain an MML instruction but the value referred to by the load instruction at line 000000. We say that 4 is a constant because its value is explicitely written into the program by a programmer. We also say that 4 is an argument of the load instruction meaning that the couple (load, 4) for a single entity.
- Instruction at line 000002 tells the CPU to load next program line and push its value (1000), representing the memory address of the variable a, into the stack.
- Line 000003 contains the argument of the above load instruction.
- Line 000004 tells the compiler to store 4 at memory address 1000, i.e. store the avlue 4 into variable a.
- Lines 000005 through line 000031 do contain pass instructions are are ignored.
- Line 000032 tells the CPU to stop and not to increment the counter again.

The CPU starts the program execution at line 000000 and at each step increments the program counter of one unity thus executing the next program instructions. Hence, the above program instructions are excueted in the order they are stored in memory.

If we were to replace variable a with varibale b the number 1000 in the program would have been replaced by the number 1001 according to the above table. Modern high level languages associate variable names to memory addresses according to more complex rules that we will discuss later.

As another example, we may want to store two numbers, 4 and 7.123412, into the memory in two different variables called, a and b, respectively. Here is the corresponding FC:

(3.3)

Which the FCD compiles into the following code:

```
### a=4
000000 : load
000001 : 4
000002 : load
000003 : 1000 # addr. a
000004 : write
### b=7.123412
000032 : load
000033 : 7.123412
000034 : load
000035 : 1001 # addr. b
000036 : write
### stop
000064 : stop
```

We leave to the reader the task to understand the MML code. We just notice that every assignment instruction

$$variable = \text{constant}$$

is always compiled into the following five program lines:

load
constant
load
address of the variable according to table
write

3.10. The third program: output the value of a variable

We have seen how to write an MML program that stores a constant into a variable. We now want to output the value of a variable to the screen (or other output device). In the end we want to write long programs that perform complex computations. Eventually the program will produce a result that is stored into one or more variables and we will want to output this result.

The MML command the instruct the CPU to send a numeric value of the output is print.

Let's insert a new statement into program 3.2 that prints prints the variable a. Here is the corresponding FC:

(3.4)

And here is the compiled program:

```
### a=4
000000 : load
000001 : 4
000002 : load
000003 : 1000 # addr. a
000004 : write
### print a
000032 : load
000033 : 1000 # addr. a
000034 : read
000035 : print
### stop
000064 : stop
```

We now assume the reader understand the compilation of the a=4 statement and the the stop statement and we focus on the print a statement. We define two more MML instructions:

read The read instructions pops a number from the stack, reads the content of the variable stored at that memory address and pushes the value read into the stack.

print The print statement pops a number from the stack and send it to the output device (the screen or, in the case of the MS, to the output window).

The "print a" FC statement is compiled into them MML instructions at lines 000032 thourgh 000035. The operations corresponding operations performed by the CPU are the following:

- Instruction at line 000032 tells the CPU to push the address of a (1000) into the stack.
- Instruction at line 000034 tells the CPU to pop the address from the stack and read the numeric value stored at that memory location. The number read is then pushed into the stack.
- Instruction at line 000035 tells the CPU to pop a number from the stack and send it to the output device.

Notice that the compiler automatically renumber lines to keep then in the correct order as they appers in the FC.

Although in the MS there are two different instructions to write to the memory (write) and write to the output device (print), from the hardware point of view these two operations are very similar. The only reason to distinguish them is didactic. In practice, from the point of view of the CPU they both correspond to writing a number to some location which can be the variable memory or the output device.

To the CPU an output device looks exectely like a memory with only one difference, it can write to it but not read from it. I fact any output device is consituted of three parts, a memory connected to the CPU, the human hardware interface (for example a computer screen) and electronic circuits that copy the contect of the memory into the human interface. In the case of a computer screen, the CPU prints by writing in the memory associated to the screen, called the video

memory, and the corresponding electronic circuit, called video processor, copies the contect of the video memory into the screen.

In modern Personal Computers it is common to have video memory and video processor assembled together on a board called video cards. The computer screen has become such an important device that the video processor can be even more complex than the main processor of the computer, the CPU. A single CPU can also control multiplescreens by writing into different video memory banks, each connected to its own video processor and compuer screen.

3.11. The fourth program: input from the keyboard

All programs we have written so far do not interact with the user during execution. This is a limitation. Think for example of a program that performs some computation and we want the same computation to be performed in different sets of input data without having to rewrite the program for each set. The idea is that the program should be able to instruct the CPU to read data from an input device, for example a keyboard, and store the data read, in the form of numbers, into variables.

The reader is probably familiar with programs such as word processors and speadsheets. These programs continuously interact with the user by reading the keyboard and movements of the mouse and respond accordingly.

The MML instruction to read a number from the kayboard is input. Let's consider the following FC:

(program04)

This program instructs the CPU to read a number from the keyboard, store into variable a and then print variable a. This type of program is called an echo program since its only task is to copy to the output the numbers inputted by

the user types. Echo programs can be very important. Think for example of a routing computer within a network. The computer receives a signal from an input, interpret the signal and determines is the signal is addressed to it or not. If not the computer re-routes the signal to some other computer thorugh by echoing the input to a different output. Echo programs also run in personal computers and allow different programs running in parallel on the same computer to communicate with each other.

Program **??** is compiled into the following MML program:

```
### input a
000000 : input
000001 : load
000002 : 1000 # addr. a
000003 : write
### print a
000032 : load
000033 : 1000 # addr. a
000034 : read
000035 : print
### stop
000064 : stop
```

The program is very similar to program 3.4 but instead of storing a constant, 4, into a, it stores a value typed by the user during program execution. The new MML instruction is input.

input The input instruction tells the computer to query the user for a number. The CPU reads the keyboard from the computer and pushes the number typed by the user into the stack.

In the above program the statement "input a" is compiled into the following steps:

- Instruction at line 000000 tells the CPU to read a number from the keyboard and push it into the stack.
- Instruction at line 000001 tells the CPU to push the address of a into the stack.

- Instruction at line 000003 tells the CPU to pop the address of a from the stack and the value inserted by the user and store the latter into a.

The rest of the program performs as before, reads the value stored in a and prints it. Then it stops.

3.12. The fifth program: Pascal's adding machine

At this point we almost have the ability to write an MML program that simulates Pascal's adding machine. The task of our program is that of adding two numbers from the keyboard, adding them and printing the result of the addition. Here is the FC that corresponds to Pascal's adding machine:

(program05)

The program performs the following steps:

- reads a number from the keyboard and stores it into a
- reads another number from the keyboard and stores it into c
- sums a and b and stores the result into c
- prints c
- stops

The MML command to add two numbers is sum:

sum The instruction sum instructs the CPU to pop two numbers from the stack, send them to the ALU. The ALU sums the two numbers and sends the result back. The result is then pushed by the CPU into the stack.

Here is the compiled program:

```
# input a
000000 input
000001 load
000002 1000 # addr. a
000003 write
# input b
000032 input
000033 load
000034 1001 # addr. b
000035 write
# c=a+b
000064 load
000065 1000 # addr. a
000066 read
000067 load
000068 1001 # addr. b
000069 read
000070 sum
000071 load
000072 1002 # addr. c
000073 write
### print c
000096 : load
000097 : 1002 # addr. c
000098 : read
000099 : print
### stop
000128 : stop
```

The FC statement "c=a+b" is compiled into lines 000064 through 000073:

- Instruction at line 000064 tells the CPU to push the address of a into the stack
- Instruction at line 000066 tells the CPU to read the value of a from the memory and push it into the stack
- Instruction at line 000067 tells the CPU to push the address of b into the stack
- Instruction at line 000069 tells the CPU to read the value of b from the memory and push it into the stack
- Instruction at line 000070 tells the CPU to pop two numbers from the stack, the AL:U to add them, and push the sum back into the stack.
- Instruction at line 000071 tells the CPU to push the address of c into the stack
- Instruction at line 000073 tells the CPU to pop the address of c and pop the sum from the stack and store the sum into c.

Any other line of the MML program should be obvious to the reader.

The reader is encouraged to simulated the running of the program above on pen and paper in order to learn to keep track of the stack content and how this is used.

An MML instruction that pops a single number from the stack and pushes one number into it is called a unary operator (read and input are examples of unary operators).

An MML instruction that pops two numbers from the stack and pushes one number into is is called a binary operator (write and sum are examples of binary operators).

The MML includes the following additional arithmetic unary operators:

neg It pops x from the stack pushes $-x$

exp It pops x from the stack and pushes $\exp(x)$

log It pops x from the stack and pushes $\log_e(x)$

sin It pops x from the stack and pushes $\sin(x)$, x is assumed to be in radiants.

cos It pops x from the stack and pushes $\cos(x)$, x is assumed to be in radiants.

int It pops x from the stack and returns the integer part of x (for example is $x = 21.345$ the integer part of x is 21).

The MML includes the following additional arithmetic binary operators:

sub It pops x and y from the stack and pushes $y - x$

mul It pops x and y from the stack and pushes $y \cdot x$

div It pops x and y from the stack and pushes y/x

mod It pops x and y from the stack and pushes the reminder of y/x

pow It pops x and y from the stack and pushes the reminder of y^x

lt It pops x and y from the stack and pushes 1 if y is less than x, 0 otherwise.

eq It pops x and y from the stack and pushes 1 if y is equal to x, 0 otherwise.

gt It pops x and y from the stack and pushes 1 if y is greater than x, 0 otherwise.

Each of the above arithmetic operators has a corresponding statement in terms of variables in the FCD.

It is worth noticing that many of the above operators are redundant since for example we we could write a program that perform the same operation as neg only using sub in fact:

$$-x = 0 - x$$

In the same fashion we could write programs that perform the same operation as exp, log, sin, cos and pow in terms of sum,sub,mul and div using the *Taylor's expansions* below[1]:

$$\begin{aligned} \exp(x) &= 1 + \frac{x}{1} + \frac{x^2}{1 \cdot 2} + \frac{x^3}{1 \cdot 2 \cdot 3} + ... \frac{x^n}{... \cdot n} + ... \\ \log(1 + x) &= ... \end{aligned}$$

[1]The Taylor's expansion for $\log(1 + x)$ only works for $|x| < 0$. In any case for any $y > 0$ we can find an integer n and a real number $x < 0$ such that $y = e^n(1 + x)$. Therefore $\log(y) = n + \log(1 + x)$ and we can use the Taylor expansion.

$$\begin{aligned}
\sin(x) &= \frac{x}{1} - \frac{x^3}{1 \cdot 2 \cdot 3} + ... + \frac{(-1)^n x^{2n+1}}{... \cdot (2n+1)} + ... \\
\sin(x) &= 1 - \frac{x^2}{1 \cdot 2} + ... \frac{(-1)^n x^{2n}}{... \cdot (2n)} + ... \\
y^x &= \exp(x \log_e(y))
\end{aligned}$$

Nevertheless it is convenient to have these arithmetic implemented at the hardware level into the ALU rather then the sofware level since this this make our MS much more efficient.

3.13. The sixth program: Babbage's loops

Out machine has already one charateristic in common with Babbage's Analytical Engine. That is the ability to store information. Where Babbage's machine was using punched-cards out machine uses the Ram memory.

Babbage designed his machine to perform repeatedly the same computation in order to produce tables of matematical functions and eventually compute tables of logarithms.

The concept of performing the same operation over and over requires a new MML instruction, equivalent to the jump already present in the language of Babbage's original Differential Engine, the go instruction.

go The go instruction tells the CPU to pop a number form the stack and set the CPU counter to this value, so that program execution jumps to the line corresponding to such a number.

Before we produce a program capable of computing tables we start with a simple program prints the number 4 over and over. Here is the program FC:

The FCD compiled the above FC into the following MML program:

```
### a=4
000000: load
000001: 4
000002: load
000003: 1000 # addr. a
000004: write
### print a
000032: load
000033: 1000 # addr. a
000034: read
000035: print
### go
000064: load
000065: 32
000066: go
### stop
000096: stop
```

The go statement is compile into lines 000064 to 000066. They perform the following operations:

- Instruction at line 000064 instruct the CPU to push into stack the line number where control flow has to jump to (32).

- Instruction at line 000066 tells the CPU to pops the number 32 from the stack and reset the CPU counter to 32. After instruction at line 000066 is executed the program resumes from line number 000032.

The program above stores 4 into variable a and than prints a over and over. The program does not stop. The user has to stop program execution.

The concept of repeating the same set of instructions over and over is called a loop. The ability to loop is common to all programming languages.

Ideally we want to instruct the CPU to perform some operation, for example "print a" a finite number of times, for example 10 times. This can be done but we need to ingredients:

- A variable to be used as counter. At the beginning of the program the counter is initialized (i=0) and its value is incremented at each loop (i=i+1).
- A conditional jump such that the program loops only and only the condition i<10.

The conditional jump is discussed in the next section.

3.14. The seventh program: Ada's conditional statement

It was Adab Byron who first realized the importance of the the conditional jump or conditional go. The MML instruction for the conditional jump is goif.

goif The goif instruction tells the CPU to pop two numbers x and y from the stack and, if y is different from zero, to set the CPU counter to x.

Here the a program FC that stores 4 into a, print a and loops over the print instruction 10 times. The program uses another variable i as counter for the number of loops.

The FCD compiles it into the following MML program:

```
### a=4
000000 : load
000001 : 4
000002 : load
000003 : 1000 # addr. a
000004 : write
### i=0
000032 : load
000033 : 0
000034 : load
000035 : 1008 # addr. i
000036 : write
### print(a)
000064 : load
000065 : 1000 # addr. a
000066 : read
```

```
000067 : print
### i=(i + 1)
000096 : load
000097 : 1008 # addr. i
000098 : read
000099 : load
000100 : 1
000101 : sum
000102 : load
000103 : 1008 # addr. i
000104 : write
### if i<10 go
000128 : load
000129 : 1008 # addr. i
000130 : read
000131 : load
000132 : 10
000133 : lt
000134 : load
000135 : 64
000136 : goif
### stop
000160 : stop
```

By now the reader should be able to undertand every step of this program excpet for lines 000128 thought 000136:

- Instruction at line 000128 tells the CPU to push the address of i into the stack
- Instruction at line 000130 tells the CPU to push into the stack the value of the variable i
- Instruction at line 000131 tells the CPU to push the number 10 into the stack
- Instruction at line 000133 is pops the value of i and 10 from the stack and if i is less the 10 pushes 1 into the stack, otherwise it pushed 0 into the stack. This number 1 or 0 represents the condition (true or false respectively).

- Instruction at line 000134 tells the CPU to pushes line number 64 into the stack
- Instruction at line 000126 tells the CPU to pop the line number 64 and the condition from the stack and, if the condition is true (1), to store the number 64 into the CPU counter.

The jump is therefore conditional to i being less then i. The condition is true the first nine times its is executed, therefore the program loops nine times and executes the "print a" statement 10 times (the first it is executed before ever looping). The program therefore executes the statement "i=i+1" repeatedly. The tenth time the statement "if i<10 go" is executed the condition is false and the jump is not executed. Control flow proceeded to the next valid program instruction (at line 000130) that indicate the CPU to stop.

3.15. The eigth program: Boolean operators

Our machine, according with common conventions, uses the number 1 to indicate a true condition and the number 0 to indicate a false condition. The same is true for almost any modern programming language. According with Boole's findings, in fact, we can use numbers to represent the logical value of a proposition which can be either true or false.

Boole proposed three logical operators that can be used to construct more complex logical expressions: AND, OR and NOT. Boole's logical operators are implemented in the following MML instructions:

and The and instruction tells the CPU to pop two numbers, x and y, from the stack and, if they are both different than 0 (true), to push 1 into the stack, 0 otherwise.

or The and instruction tells the CPU to pop two numbers, x and y, from the stack and, if any of them is different than 0 (true), to push 1 into the stack, 0 otherwise.

not The and instruction tells the CPU to pop one number, x, from the stack and, if x is equal to 0 (true) to push 1 into the stack, 0 otherwise.

For arithmetic expression these logical operators are to be used in conjunction with the comparison operators (lt,eq,gt) and the goif statement.

Here is a program FC that asks the user to input a number and if the number is less then 10 or bigger than 20 loops and asks for another number. The program terminates when the user inputs a number within the desired range.

The compiled file is:

```
### a=input()
000000 : input
000001 : load
000002 : 1000 # addr. a
000003 : write
### f=(a < 10)
000032 : load
000033 : 1000 # addr. a
000034 : read
000035 : load
000036 : 10
000037 : lt
000038 : load
000039 : 1005 # addr. f
000040 : write
### g=(a > 20)
```

```
000064 : load
000065 : 1000 # addr. a
000066 : read
000067 : load
000068 : 20
000069 : gt
000070 : load
000071 : 1006 # addr. g
000072 : write
### if f or g go
000096 : load
000097 : 1005 # addr. f
000098 : read
000099 : load
000100 : 1006 # addr. g
000101 : read
000102 : or
000103 : load
000104 : 0 # prg. step
000105 : goif
### stop
000128 : stop
```

[FILL HERE]

This type of program is a validation program. Validation programs are used to check the input of other programs in order to make sure that the input is within some expected range of values. Think for example of a program that implements a diving machine. The machine inputs two numbers, x and y, and computes the ratio of x over y. A good dividing machine would check if y is zero before trying to perform the division since the result of the operation can not be predicted from the rules of arithmetics and actually depends on the internal details of the CPU.

Many modern computers follow strict specifications set by the International Electronic Engineers E(FILL HERE) about proper CPU behavior in cases like the one discussed above. Still, it is good practive to write programs that check for incorrect input and behave appropriately, eventually stopping and notifying the user of the wrong input.

At this point we have almost completely exhausted the list of instructions supported by the CPU of the MS. All instructions we have introduces so far

are sufficiant for machine our machine equivalent to a Turing Machine and some of them are even reduntant. Never the less we want to introduce a few more instructions that will help us writing more efficient MML code (dup and swap) and deal with characters other then numbers (inputc, printc). We will introduce them later in the chapter.

3.16. Max of a set

```
### m=0
000000 : load
000001 : 0
000002 : load
000003 : 1012 # addr. m
```

```
000004 : write
### a=input()
000032 : input
000033 : load
000034 : 1000 # addr. a
000035 : write
### if a == 0 go
000064 : load
000065 : 1000 # addr. a
000066 : read
000067 : load
000068 : 0
000069 : eq
000070 : load
000071 : 192
000072 : goif
### if not a > m go
000096 : load
000097 : 1000 # addr. a
000098 : read
000099 : load
000100 : 1012 # addr. m
000101 : read
000102 : gt
000103 : not
000104 : load
000105 : 32
000106 : goif
### m=a
000128 : load
000129 : 1000 # addr. a
000130 : read
000131 : load
000132 : 1012 # addr. m
000133 : write
### go
000160 : load
```

000161 : 32 # prg. step
000162 : go

3.17. Average of a set

```
### b=0
000000 : load
000001 : 0
```

```
000002 : load
000003 : 1001 # addr. b
000004 : write

### i=0
000032 : load
000033 : 0
000034 : load
000035 : 1008 # addr. i
000036 : write

### a=input()
000064 : input
000065 : load
000066 : 1000 # addr. a
000067 : write

### if a == 0 go
000096 : load
000097 : 1000 # addr. a
000098 : read
000099 : load
000100 : 0
000101 : eq
000102 : load
000103 : 224
000104 : goif

### i=(i + 1)
000128 : load
000129 : 1008 # addr. i
000130 : read
000131 : load
000132 : 1
000133 : sum
000134 : load
000135 : 1008 # addr. i
```

```
000136 : write

### b=(b + a)
000160 : load
000161 : 1001 # addr. b
000162 : read
000163 : load
000164 : 1000 # addr. a
000165 : read
000166 : sum
000167 : load
000168 : 1001 # addr. b
000169 : write

### go
000192 : load
000193 : 64 # prg. step
000194 : go

### b=(b / i)
000224 : load
000225 : 1001 # addr. b
000226 : read
000227 : load
000228 : 1008 # addr. i
000229 : read
000230 : div
000231 : load
000232 : 1001 # addr. b
000233 : write

### print(b)
000256 : load
000257 : 1001 # addr. b
000258 : read
000259 : print
```

```
### stop
000288 : stop
```

3.18. Tables of logarithms

a=input()

```
000000 : input
000001 : load
000002 : 1000 # addr. a
000003 : write
### b=0
000032 : load
000033 : 0
000034 : load
000035 : 1001 # addr. b
000036 : write
### c=1
000064 : load
000065 : 1
000066 : load
000067 : 1002 # addr. c
000068 : write
### i=1
000096 : load
000097 : 1
000098 : load
000099 : 1008 # addr. i
000100 : write
### b=(b + c)
000128 : load
000129 : 1001 # addr. b
000130 : read
000131 : load
000132 : 1002 # addr. c
000133 : read
000134 : sum
000135 : load
000136 : 1001 # addr. b
000137 : write
### c=(c * a)
000160 : load
000161 : 1002 # addr. c
000162 : read
```

```
000163 : load
000164 : 1000 # addr. a
000165 : read
000166 : mul
000167 : load
000168 : 1002 # addr. c
000169 : write
### c=(c / i)
000192 : load
000193 : 1002 # addr. c
000194 : read
000195 : load
000196 : 1008 # addr. i
000197 : read
000198 : div
000199 : load
000200 : 1002 # addr. c
000201 : write
### print(b)
000224 : load
000225 : 1001 # addr. b
000226 : read
000227 : print
### i=(i + 1)
000256 : load
000257 : 1008 # addr. i
000258 : read
000259 : load
000260 : 1
000261 : sum
000262 : load
000263 : 1008 # addr. i
000264 : write
### if i < 20 go
000288 : load
000289 : 1008 # addr. i
000290 : read
```

000291 : load
000292 : 20
000293 : lt
000294 : load
000295 : 128
000296 : goif
stop
000320 : stop

```
### a=input()
000000 : input
000001 : load
000002 : 1000 # addr. a
000003 : write

### b=exp(a)
000032 : load
000033 : 1000 # addr. a
000034 : read
000035 : exp
000036 : load
000037 : 1001 # addr. b
000038 : write
```

```
### print(b)
000064 : load
000065 : 1001 # addr. b
000066 : read
000067 : print

### stop
000096 : stop
```

3.19. Random numbers

Computing machines are deterministic. They execute the program blindly in a predictable way. If two machines execute the same program they must produce the same result and, if they do not, at leat one them is defective. A computing machine does not add content to our program. The meaning of the program is in the program itself and in the machine language implemented in the machine. Although for the programmer the program may mean something or represent something that exists outside the program itself, this is not the case from the point of view of the machine.

Nevertheless it would be wrong to conclude that machines cannot teach us anything about randomness and chaos. Machines teach us that order can generate chaos and randomness and the latter can generate ordered structures of extream complexity and beauty.

Consider a simple algorithm that performs the following steps:

The above algorithm generates an infinite sequence of numbers. On the one side this sequence is perfectly deterministic but, on the other side, the sequence of numbers has property that we typically associate with randomness:

- For each interval $[x, y)$ subset of $[0, 1)$, the probability of finding a number of the sequence in the interval is equal to $y - x$.
- There is no simple way to predict the nth number of the sequence without going through the computation of the sequence.
- There is no apparent correlation between numbers in the sequence that are separated by many loops of the algorithm.
- Different initial values for a and b produce a different sequence of numbers.

If we initialize the sequence with a value that is time dependent, such as the number of seconds since [FILL HERE] a programmer cannot predict the output of the above algorithm since there is no way to tell the exact time when the program will be executed.

Order has produced randomness or, more precisely, pseudo-randomness.

There a difference between real randomness and pseudo-randomness. In a real random generator each number (within some interval) has the same probability to appear in the sequence. In a pseuso-random generator, such as the algorithm above, it is known that only a finite set of numbers may appear in the sequence and the sequence itself, in fact, is periodic. The period, in terms of number of loops, can be very big and for a pseudo-random generator is exceeds the number of atoms in the universe. For practical purposes a good pseudo-random generator is equivalent to a real random generator.

One way wonder if it is at all possible to build a physical device that is a real random generator. The answer is yes because all funadamental physics law are based on quantum mechanics. Qunatum mechanics asserts that if we prepare that state of an elementary particle (for example we place an electron in a determined position in space) and we measure its state at some later time the result is random and non deterministic. It is not possible to write an algorithm that predicts the result of the experiment. Only the probability of obtaining any and each result is predictable and can be computed algorithmically. It is therefore conceivable to construct a phsyical device based on quantum mechanical laws that generates a sequence of random results which cannot be reproduced by any algorithm. There are many examples of such devices but they are not exploited in modern day computers.

As order generates chaos so chaos can generate order. At the hardware level the transistor, the basic component of any modern electronic computer, is a quantum mechanical device. The transistor would not exist in a classical and deterministic universe. It works because of the laws of quantum mechanics that govern the behavior of atoms and electrons. Nevertheless, at a macroscopic level, one single transistor is made of billions of atoms and in typical computing devices it is attraversed by corrents of billions of electrons. We do not observe the interactions of one electron with one atom but, rather, the collective interaction of the current with the device. Quantum mechanical effects averages out and we observe a collective phenomenon that appears classical and deterministic (this may change as electronic components get smaller and smaller). The computer is made of transistors and it is also a classical and deterministic device.

Everything that we observe in the physical world can be described algorithmically and simulated by a compuer therefore it come with no surprice that also at the software level chaos can generate order. For example it is possible to write compuer programs that use sequences of random numbers to perform computations. We will see in Chapter [FILL HERE] how to compute π using random numbers with a technique called Monte Carlo. Pseudo-random sequences can also be used to generate complex structures known as fractals, a property that is used from data compression to video-games.

The MML language has one instruction to generate random numbers.

rand The rand instruction tells the CPU to produce (somehow) a random number in the interval $[0, 1)$ with uniform distrubution and push it into the stack.

The follwoing FC shows a program that stores a random number into the variable a and prints it.

Here is the compiler program:

```
### a=rand()
000000 : rand
000001 : load
000002 : 1000 # addr. a
000003 : write
### print(a)
000032 : load
000033 : 1000 # addr. a
000034 : read
000035 : print
```

```
### stop
000064 : stop
```

3.20. Introduction to arrays

The FCD supports two types of variables, a numeric type and an array type. While a numeric type of variable is associated to a single memory cell, a array type is associated to a multiple memory cells. The variable name, for example `p`, does not refer to the address of a single cell but to the collective set of memory cells. The notation `p[`i`]` refers to the i-th cell in the array. The index i can go from 0 (the first cell in the array) to an arbitrary number. Any array element, `p[`i`]` is a veriable.

The FCD maps the the variable `p[`i`]` in 2000+i, according to the table **??**. 2000+i indicates a number equal to 2000 (the starting position of the array in memory) plus the value stored into variable i.

Arrays are a very useful programming tool. At the MML level there is no such a this as a variable or an array since the memory is just a set of cells labelled by an address.

Arrays allow progremmers to use the fact that memory is linear. For example a program may have to store a large set of numbers and there is no need to store them in different variables and give a different name to each of them. This would make the program unnecessarility long and tedious. By using an array the program can store the set of numbers in contiguous memory cells and give a collective name to the set (the array name). Each item in the set can be accessed with the syntax p[i].

The following program FC uses a loop to fill an array of 10 elements with random numbers. Another loop goes through the array elements and prints each of them in the order they were generated.

FILL HERE

Here is the compiled program.

```
### i=0
000000 : load
000001 : 0
000002 : load
000003 : 1008 # addr. i
```

```
000004 : write

### p[i]=rand()
000032 : rand
000033 : load
000034 : 2000 # addr. p[0]
000035 : load
000036 : 1008 # addr. i
000037 : read
000038 : sum
000039 : write

### i=(i + 1)
000064 : load
000065 : 1008 # addr. i
000066 : read
000067 : load
000068 : 1
000069 : sum
000070 : load
000071 : 1008 # addr. i
000072 : write

### if i < 10 go
000096 : load
000097 : 1008 # addr. i
000098 : read
000099 : load
000100 : 10
000101 : lt
000102 : load
000103 : 32
000104 : goif

### i=0
000128 : load
000129 : 0
```

```
000130 : load
000131 : 1008 # addr. i
000132 : write

### print(p[i])
000160 : load
000161 : 2000 # addr. p[0]
000162 : load
000163 : 1008 # addr. i
000164 : read
000165 : sum
000166 : read
000167 : print

### i=(i + 1)
000192 : load
000193 : 1008 # addr. i
000194 : read
000195 : load
000196 : 1
000197 : sum
000198 : load
000199 : 1008 # addr. i
000200 : write

### if i < 10 go
000224 : load
000225 : 1008 # addr. i
000226 : read
000227 : load
000228 : 10
000229 : lt
000230 : load
000231 : 160
000232 : goif

### stop
```

```
000256 : stop
```

There are a few things to note:

- The program known that the array containes 10 elements because this number is a constant in the program. It is not possible to query an MML array for its size. This is very similar to C and C++ arrays. Java and Python arrays have a more complex memory representation and store information about their size.
- Since an array does not store information about its end an array can be outofbounds. For example, the FCD stores `p[1000]` at the same memory location as `q[0]`, therefore the programmer has to be careful to avoid conflics in the use of the two variables. C and C++ present the same problem. Java and Python check for array bounds and throw and error message if array bounds are violated.
- Although, at first site, a reader may think that the FCD is limited by the fact that it supports only a few variables (a-k,m,n,p[],q[] and r[]), it is worth noticing that, in principle the FCD can address a theoretically infinite memory space by suing the array r[]. Such array can, in principle, extend indefinitely. This observation is crucial when comparing our machine with a Turing machine. One critical characteristics of Turing mechines is their infinite tape. Our machine has an infinite memory and both the FCD and the MML can access it.

3.21. Max of an array

The follwong program FC fills an array with 10 random numbers and then loops over the array elements and determines its maximum.

FILL HERE

Here is the compiled program.

```
### i=0
000000 : load
000001 : 0
000002 : load
```

```
000003 : 1008 # addr. i
000004 : write

### p[i]=rand()
000032 : rand
000033 : load
000034 : 2000 # addr. p[0]
000035 : load
000036 : 1008 # addr. i
000037 : read
000038 : sum
000039 : write

### i=(i + 1)
000064 : load
000065 : 1008 # addr. i
000066 : read
000067 : load
000068 : 1
000069 : sum
000070 : load
000071 : 1008 # addr. i
000072 : write

### if i < 10 go
000096 : load
000097 : 1008 # addr. i
000098 : read
000099 : load
000100 : 10
000101 : lt
000102 : load
000103 : 32
000104 : goif

### i=0
000128 : load
```

```
000129 : 0
000130 : load
000131 : 1008 # addr. i
000132 : write

### m=p[i]
000160 : load
000161 : 2000 # addr. p[0]
000162 : load
000163 : 1008 # addr. i
000164 : read
000165 : sum
000166 : read
000167 : load
000168 : 1012 # addr. m
000169 : write

### if not p[i] > m go
000192 : load
000193 : 2000 # addr. p[0]
000194 : load
000195 : 1008 # addr. i
000196 : read
000197 : sum
000198 : read
000199 : load
000200 : 1012 # addr. m
000201 : read
000202 : gt
000203 : not
000204 : load
000205 : 256
000206 : goif

### m=p[i]
000224 : load
000225 : 2000 # addr. p[0]
```

```
000226 : load
000227 : 1008 # addr. i
000228 : read
000229 : sum
000230 : read
000231 : load
000232 : 1012 # addr. m
000233 : write

### i=(i + 1)
000256 : load
000257 : 1008 # addr. i
000258 : read
000259 : load
000260 : 1
000261 : sum
000262 : load
000263 : 1008 # addr. i
000264 : write

### if i < 10 go
000288 : load
000289 : 1008 # addr. i
000290 : read
000291 : load
000292 : 10
000293 : lt
000294 : load
000295 : 192
000296 : goif

### print(m)
000320 : load
000321 : 1012 # addr. m
000322 : read
000323 : print
```

```
### stop
000352 : stop
```

The possibility of looping over the elements of an array is its most imporant characteristics. We can use arrays to store tables in memory and perform a variety of operations on the table, including searching and sorting. Databases are based on this principle. Modern databases use complex algorithms to perform these operations with high efficiency but, at the bottom, a database is a collection of tables and programs that act of those tabels.

3.22. Linear search an array

FILL HERE

```
### i=0
000000 : load
000001 : 0
000002 : load
000003 : 1008 # addr. i
000004 : write

### p[i]=input()
000032 : input
000033 : load
000034 : 2000 # addr. p[0]
000035 : load
000036 : 1008 # addr. i
000037 : read
000038 : sum
000039 : write

### i=(i + 1)
000064 : load
000065 : 1008 # addr. i
000066 : read
000067 : load
000068 : 1
```

```
000069 : sum
000070 : load
000071 : 1008 # addr. i
000072 : write

### if i < 10 go
000096 : load
000097 : 1008 # addr. i
000098 : read
000099 : load
000100 : 10
000101 : lt
000102 : load
000103 : 32
000104 : goif

### i=0
000128 : load
000129 : 0
000130 : load
000131 : 1008 # addr. i
000132 : write

### a=input()
000160 : input
000161 : load
000162 : 1000 # addr. a
000163 : write

### if a == p[i] go
000192 : load
000193 : 1000 # addr. a
000194 : read
000195 : load
000196 : 2000 # addr. p[0]
000197 : load
000198 : 1008 # addr. i
```

```
000199 : read
000200 : sum
000201 : read
000202 : eq
000203 : load
000204 : 320 # prg. step
000205 : goif

### i=(i + 1)
000224 : load
000225 : 1008 # addr. i
000226 : read
000227 : load
000228 : 1
000229 : sum
000230 : load
000231 : 1008 # addr. i
000232 : write

### if i < 10 go
000256 : load
000257 : 1008 # addr. i
000258 : read
000259 : load
000260 : 10
000261 : lt
000262 : load
000263 : 192
000264 : goif

### go
000288 : load
000289 : 352 # prg. step
000290 : go

### print(m)
000320 : load
```

```
000321 : 1012 # addr. m
000322 : read
000323 : print

### stop
000352 : stop
```

3.23. Dealing with characters

Until this point we have explored how our machine can manage numbers and computations that involve numbers. In this section we want to investigate how to extend the functionality of our mahcine to deal with characters and strings.

The main idea is that of listing all possible characters that the machine should be able to manipulate and number them so that internally each chacarter is represented by the machine as a number. The machine does not need to "understand" the concept of a character or include a new set of funcitonalities to handle chacarters. The machine only need to be able to transform an output sequence of characters (for example typed by the user on the keyboard) and transform them into a sequence of characters. Conversely the machine need to be able to transform numbers into characters when sending them to the output (for example displaying them on a screen).

The procedure of listing all symbols in a language and associating integer numbers to it is sometime referred to as Gödelization becasue the power of this apparently simple procedure was first applied by Gödel is his proof that all formal languages (whatever the symbols, the axioms and the rules) are equivalent to aritmetic. We have already implicitely see one example of Gödelization at the beginning of the chapter when we assumed that it were possible to associate each MML instruction with a number and our machine could store the program as a sequence of numbers, each corresponding to an individual instruction.

Now we want to do it again for the symbols used by the humans (our alphabet) rather the the symbols used by the machine (the MML instructions) and, again, this is a matter of convention.

There are three used conventions to map charaters into number and vice versa:

- EBCDIC. This is a very old convention not used any more by modern computers.

- ASCII. This is the standard convention used by almost every computer in use today and it includes a mapping for 256 symbols including the English alphabet and some additional symbols common in European languages (é,è,ö,etc.). The ASCII table is reported in Appendix A, in table **??**.
- UNICODE. This is an extension of the ASCII to include some exhotic symbols and orintal alphabets. While the ASCII mapping is implemented in hardware and recognized by all programming languages, the UNICODE mapping is implemented in software and only modern languages such as C++, Java and Python support UNICODE characters.

Our machine simulator follows the ASCII convention. The MML language provides two instruciton to deal with characters:

inputc The inputc instruction tells the CPU to read a character from the keyboard and push the corresponding ASCII number into the stack.

printc The printc instruction tells the CPU to pop a number from the stack and send the corresponding ASCII character to the output.

It is important to notice that the convertions

$$\begin{aligned} \mathit{number} &\rightarrow \mathit{character} \\ \mathit{character} &\rightarrow \mathit{number} \end{aligned}$$

are performed by the output device rather then by the CPU itself. From the point of view of the CPU everything is a number.

Here is a simple FC that describes how to read a single charater from the keyboard and to output its corresponding ASCII character:

And the corresponding compiled program:

```
### a=inputc()
000000: inputc
000001: load
000002: 1000 # addr. a
000003: write
### print(a)
000032: load
000033: 1000 # addr. a
000034: read
000035: print
#### stop
000064: stop
```

The program above can be used to build the table **??** since the table is already built-in the MS and used by the inputc instruction.

Conversely we can build a FC that describes the opposite operations (input a number from the keyboard and output the corresponding ASCII chacarter:

Here is the compiled program:

```
### a=input()
000000: input
000001: load
000002: 1000 # addr. a
000003: write
### printc(a)
```

```
000032: load
000033: 1000 # addr. a
000034: read
000035: printc
#### stop
000064: stop
```

It is interesting it interesting to obeserve that the ASCII table includes characters that cannot be visualized by can be used to control the output. These characters are called control charaters. For example the ASCII number 13 corresponds to "go to a new line" or ASCII number [FILL HERE] corresponds to moving the output cursor 6 spaces on the left. The ASCII number [FILL HERE] corresponds a beep and it is typically used to notify the user that an error occurred during a computation.

In also important to observe that a digit, for example "4", can be read as an ASCII character (by inputc), [FILL HERE], or as a primitive number (by input), 4. Analogously a number stored in memory, for example 70, can be printed as the corresponding ASCII characters (by printc), "F", or as a primitive number (by print), 70.

A value that does not require to be interpreted (converted) is called a primitive type. A value that does require conversion is not a primitive type. Our machine recognizes three types a primitive types: integer numbers with sign, floating point numbers and characters. The character is a primitive type because the converion is done by the machine and not by the program.

3.24. Strings as arrays of characters

A string can be defined as an ordered set of characters. A string is not a primitive type of the MML and, actually, there is not such a structure as the string in the MML or in any other machine language. Nevertheless it is possible to write a program that implements the concept of string as a derived type. High level languages, such as C++, Java and Python do have keywords to represend and handle strings as primitive objects. Internally these languages deals with the string in the same way as our machine simulator does, using arrays.

A string in fact can be thought of an array of characters. Each single character can be represented by the corresponding ASCII number. A string is nothing else then a set of charaters, hence it can be represented by an arrays of integer numbers.

The only open issue is how to store information about the length of the string. For example is we store a string into an array and we want to print the string we can write our a program that loops over the array elements and, for each of them, prints the corresponding ASCII character. The problem is how to write the stopping condition for the loop.

There are two common solutions to this problem:

- Store the length of the string into an auxiliary variable and use a counter into the loop. After printing each character increase the counter of one unit. When the counter equals the length of the string then all characters in the string have been printed therefore break the loop.
- Do not store the length of the array but, instead, store a zero in the array after the last character of the string. Before printing each character of the string check if the array element is zero. If it is break the loop.

Each of these two solutions has advantages and disadvantages. The first of them is implemented in languages such as Java and Python. The second is adopted in the C and C++ languages.

Here is a program FC that shows how to store the string "Hello" into the array p and how to print the string by looping over the array p.

FILLE HERE

And here is corresponding MML code:
[FILL HERE]

3.25. Functions

A crucial issue in programming is the ability to reuse code. For examle we have seen programs that perform complex functions such as searching, sorting, printing strings, etc.

It is not desirable to rewrite these pieces of codes every time we need them but we want to be able to reuse them, as components, in other programs. It would be ideal to have a mechanism to store pieces of MML code so that we can reuse them. Functions and function calls provide this tipe of mechanism.

Consider the following piece of code:

```
[...]
100000: swap
100001: dup
100002: read
100003: dup
100004: not
100005: load
100006: 1000012
100007: goif
100008: printc
100009: load
100010: 1000000
100011: go
100012: dump
100013: dump
100014: go
```

This piece of code is not autonomous and it must be part of some bigger program. Line 100000 assumes the stack contains two numbers, one representing a memory address and one representing a program line. The subsequent lines assume that the memory address corresponds to the beginning of a null terminated string and print each character in the string (lines 100009 through 100011 implement the loop over the characters).

When the program encounters the terminating zero in the string the control flow jumps to line 100012 (line 100004 through 100007 implement the conditional jump) where the stack is cleaned of leftover stuff and control flow jumps back to the program line that had been stored in the stack before line 100000 was executed.

The program uses three new instructions:

dup (duplicate) Pop x from the stack and push x twice into the stack.

swap Pop x and y from the stack and push x and y into the stack (in reverse order).

dump Pop x from the stack and discard (dump) it.

We leave details about the internal working of the piece of code to the reader. What is imporant is that it can be reused many times within the same program

and can also be reused by different programs. This piece of code performs the task of printing a string. In order to call this piece of code a program must do the following:

- push into the stack the address of the string that has to be printed
- push into the stack the program line number where program execution has to resume after the print is printed
- jump to the line where the piece of code is stored, in the example line 1000000.

A piece of code written to be reused is called a procedure or a function. In old languages the term procedure was reserved for a piece of code that does not leave new data into the stack and, conversely, the term function was used when the piece of code does leave new data into the stack. We do not make this distinction in this book.

The operation of pushing data into the stack in order to pass them to a function is called parameter passing. The numbers pushed into the stack are called calling parameters. When referring to the calling parameters inside the function itself they are called arguments. Any description of the data expected by a function is called signature of the function. The data pushed into the stack by the function upon returning are colled return values of the function. When a function completes its task we say the function returns.

One more issue with functions needs to be discussed variables. If a function uses variable created by some other function or some other piece of the program, those variables are called global variables. If the function only uses variables not used by any other function of other piece of the program then they are called local variables.

Writing functions in MML is very tedious and error prone. Almost all high level languages have a keyword to identify pieces of code that are to be used as functions. Functions have names and they can be called by name. The compiler determines where to store a function in memory and how to pass parameters to the function.

Some languages such as C and C++ allow the programmer to query about memory address where a function is stored (line 100000 in the example above). Other languages such as Java or Python do not allow it and the only way to refer to a function is its name.

Some languages such as C++ and Java allow different functions to have the same name as long as their signatures are different. When two functions have the same name but different signatures thay are compiled independently and stores in different memory locations. This language feature is called function overloading.

Moreover high level languages allow the programmer to create local variables of one function that have the same name as local functions of another function. The compiler recognizes that variables with the same name used within different functions mkust be stored in at different memory address so that there is no intereference. When the programmer wants two local variables to be stored at the same address, he or she cay say so by using the appropriate language syntax.

The concepts of procedure and function were first introduced by Grace Hopper in the FLOW-MATIC language and then the COBOL language. We will make an extensive use of functions in the following chapters.

We hope we have convinced our reader that the MS described in this chapter is a very general device capable of performing any computation.

4. Hardware

4.1. The von Neuman architecture

In the preceding chapter we have focused on the concept of Algorithm and in the order to explain the concept of elementary step we proposed an ideal machine. We also described the funcional chacateristics of such machine. In this chapter we descibe how such functionalities can be implemented in hardware using standardized electronic components. The reader should be aware that since this is not an electronics we will just skim the surface of this complex subjects called electronics engineering.

4.2. Analog and Digital

4.3. Binary representation of integers

4.4. Binary representation of floating point

4.5. Boolean gates

4.6. Adding with Boolean gates

4.7. Bubtracting with Boolean gates

4.8. Flip-Flops and memory

4.9. Counting with Boolean gates

4.10. Input/Output and multiplexing/demultiplexing

4.11. A big picture

5. Structured Programming

6. Data Structures

6.1. Information rapresentation

6.2. Arrays

6.3. Lists

6.4. Trees

6.5. Graphs

6.6. Serialization

6.7. Data compression

7. Algorithms

8. Applications

9. Artificial Intelligence and Limits of Computation

10. Future Prospective and Ethical Considerations

11. Glossary of Terms

12. Appendix A. Flow Chart Runner

13. Appendix B. Algorithms Animator

14. Appendix C. Python Reference

15. Bibliography

Bibliography

[1] Muhammad ibn Musá Khuwarizmi, Robert of Chester's Latin translation of al-Khwarizmi's al-Jabr : a new critical edition, F. Steiner Verlag Wiesbaden.

[2] Euclid, The Elements (CHECK)

[3] Douglas R. Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid, Basic Books (1979)

[4] John von Neumann, The Computer and the Brain, Yale Univ. Press (2nd Ed.) (2000)

[5] Marvin L. Minsky, Society of Mind, Touchstone Books (1988)

[6] Steve J. Heims, John Von Neumann and Norbert Wiener: From Mathematics to the Technologies of Life and Death, MIT Press (1980)

[7] Ioan James, Remarkable Mathematicians: From Euler to non Neumann, Cambridge University Press (2003)

[8] Bertrand Russell, The History of Western Philosophy, Touchstone Books

[9] Bruce Collins and James MacLachlan, Charles Babbage and the Engines of Perfection, Oxforn University Press (1999)

[10] Betty A. Toole (editor), Ada Byron, Ada, The Enchantress of Numbers, A Selection from the Letters of Lord Byron's Daughter and Her Decsription of the First Computer, CarTech, Inc. (1998)

[11] George Boole, An an Investigation of the Laws of Thought, Dover Publications (1073)

[12] D.C. Ince (Editor), Alan Mathison Turing, Collected Works of A.M. Turing : Mechanical Intelligence, North-Holland; (1992)

[13] William Aspray, John Von Neumann and the Origins of Modern Computing, The MIT Press (1990)

[14] John von Neumann and Oskar Morgenstern, Theory of Games and Economic Behavior, Princeton University Press (1980)

[15] Alfread North Whitehead and Bertran Russell, Principia Mathamatica, Cambrdige University Press (1989)

[16] Kurt Godel, On Formally Undecidable Propositions of Principia Mathmatica and Related Systems, Dover Publications (1992)

[17] Ernest Nagel, James Newman and Douglas Hofstadter, Godel's Proof, New York University Press (2002)

[18] Gregory Chaitin, The Limits of Mathematics: A Course of Information theory and Limits of Formal Reasoning, Springer Verlag (1997)

[19] Nancy Whitelaw, Grace Hopper: Programming Pioneer, W. H. Freeman & Co. (1995)

[20] Robert Pollack, Signs of Life: The Language and Meaning of DNA, Meriner Books (1995)

[21] Peter Frey, Chess Skill in Man and Machine (2nd Ed.). Springer Verlag, N.Y. (1983)

[22] George. Polya, How to Solve It, Princeton University Press (1971)

[23] Benjamin Pierce, Type Systems and Programming Languages, MIT Press (2002)

[24] Giulio Casati and Boris Chirikov, Quantum Chaos: Between Order and Disorder, Cambridge University Press (1995)

[25] R. Jones and R. Lins, Garbage Collection: Algorithms for Authomatic Dynamic Memory Management, John Wiley and Sons, New York (1996)

[26] Hartley Roger, Theory of Recursive Functions and Effective Computability, MIT Press (1987)

[27] E. S. Roberts, Thinking Recursively, John Wiley m& Sons (1986)

[28] Henri Poincare, Andrew Pyle and Bertrand Russell, Science and Method, St. Augustine Press (2001)

[29] Couch Leon W. II, Digital and Analog Communication Systems, Prentice Hall (2001)

[30] David A. Patterson, John L. Hennessy and Nitin Indurkhya, Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann (1997)

[31] William Stallings, Computer Organization and Architecture, Prentice-Hall (2002)

[32] Guido van Rossum, Python Essential Reference (2nd Ed.), Que (2001)

[33] D.G. Brobow, Representation and Understanding: Studies in Cognitive Science, Academic Press, N.Y. (1975)

[34] Margaret Boden, The Philosophy of Artificial Life, Oxford University Press (1996)

[35] Hilary Putnam, Representation and reality, MIT Press (1991)

[36] K. Sayood, Introduction to Data Compression, Morgan Kaufmann, San Francisco (2000)

[37] Darrel R. Hankerson, Introduction to Information Theory and Data Compression, Chapman & Hall (2003)

[38] John Miano, Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP, Addison-Wesley (1999)

[39] David J. Hand, Heikki Mannila and Padhraic Smyth, Principles of Data Mining, MIT Press (2001)

[40] D. E. Knuth, The Art of Computer Rrogramming, Addison Wesley (1998)

[41] Thomas H. Cormen et al., Introduction to Algorithms, MIT Press (2001)

[42] Robert Sedgewick, Algorithms in C++, Addison-Wesley (2001)

[43] Gregory Chaitin, Exploring Randomness, Springer Verlag (2001)

[44] Margaret Boden, Artificial Intelligence and Natural Man, Basic Books, N.Y. (1977) (best book on AI)

[45] Willard Van Orman Quine, Word and Object, MIT Press, (1964)

[46] William Poundstone, Labirinth of Reason: Paradox, Puzzles, and the Fragility of Knowledge, Anchor Press (1990)

[47] Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach (2n Ed.), Prentice-Hall (2002)

[48] Richard Stallman, Lawrence Lessig and Joshua Gay, Free Software, Free Society: Selected Essays of Richart M. Stallman, Free Software Foundation (2002)

[49] Deborah Johnson, Computer Ethics (3rd Ed.), Prentica Hall (2000)

[50] Richard Feynman, QED, Princeton University Press (1988)

[51] Richart Feynman, Robin Allen and Tony Hey, Feynman Lectures on Computation, Perseus Publishing (2000)

[52] Claude Shannon, A. D. Wyner and Neil Sloan, Claude Shannon: Collected Papers, Wiley-IEEE Press (1993)