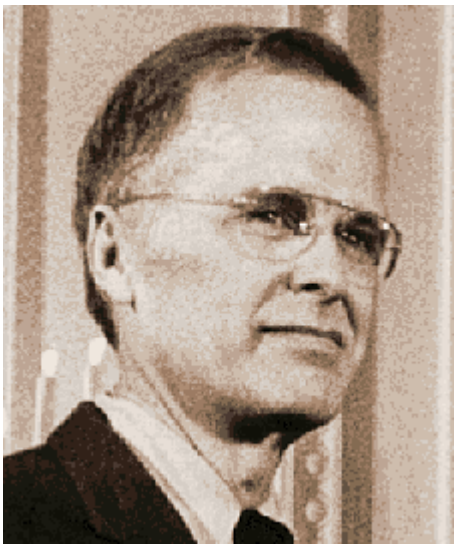
MAX3992
Project Module

"The origins and the development of two of the first high level programming languages ",

by Iason Kastanis
9838733



Module Leader: Dr Tony Crilly
Supervisor: Dr Ivor Grattan-Guinness

Abstract

This project is a historical journey through the ideas that created Computing Science out of Mathematics. The first two high-level programming languages were chosen to be compared as the end of this journey. The fundamental ideas of the logic that is involved with Computing Science are described in the first chapter. The project proceeds to describe the different types of programming languages and their mathematical basis. This is followed by a presentation of language evaluation criteria. A comparison of procedural and non-procedural programming languages is made in chapter 3, where the concept of proceduralism is clarified. The main ideas of the lambda-calculus of Alonzo Church are described in chapter 4. The evolvement of the lambda-calculus with programming languages is also discussed in the same chapter. Konrad Zuse's Plankalkül and John Backus' FORTRAN are discussed respectively on the fifth and sixth chapter. The discussion of the two programming languages takes place in three levels, the scientific surroundings in which they were developed in, the scope of their design and finally the features of the programming languages. In chapter 7 a cognitive tool for discussing about programming languages is presented to the reader. The different types of semantics are described through comparison examples of Plankalkül and FORTRAN. In the conclusion a final comparison between the two first high-level programming languages is discussed in all three levels as mentioned before. This project is aiming to discuss not only historical facts or characteristics of the languages, but to describe the main ideas in mathematics and logic and the way these ideas influenced the development of the (Theoretical) Computing Science.

Acknowledgements

I would like to thank Dr Ivor Grattan-Guinness for supervising the project and offering helpful advice, Dr Philip Maher and Dr Paul Curzon for reading and commenting on the project. I would like to thank my father Nikos Kastanis for providing me with his useful knowledge of the history of mathematics.

Table of Contents

Chapter 1
A short history of programming logic

Computing Science mainly evolved in the last half of the 20th century, even though the basis was laid down a long time before the development of the microchips. In the early nineteen hundreds mathematicians researched and developed the logical basis for all computers. The works of Alan Mathison Turing (1912-1954), Howard Aiken (1900-1973), Maxwell Herman Alexander Newman (1897-1984), John von Neumann (1903-1957) and others explored the Mathematical Logic and created a new scientific field of Logic. Logic offered many things to Computer science: "the very formal syntax, various aspects of programming languages, the algorithmic setting of computational logic, aspects of computational complexity, the λ-calculus, logic programming and many others. In return the Computer science has provided interesting new logical systems (dynamic logic, logics of knowledge, non-monotonic logics) for logicians to study, and many new interesting problems, some with a very direct connection with practice." [Davis 1988, p. 357]

The original invention of the computer had to do, as the word implies, with computing or calculating, but nowadays computers seem to most users to be doing everything else apart from calculating. A very simplified definition of a computer is that it is a translating and calculating machine, a machine that takes various inputs (keyboard, mouse, commands, etc) and translates in to a mathematical and fully algorithmic language and processes them to give a result. It was this idea, based on the theories of Computability and Provability, that the pioneers of Computing Science pursued and developed this universal mathematical language, that allows a machine to understand complicated programs and inputs that seem to have nothing to do with calculations. This plan was understood and visualized by many of the early computer pioneers. Their research focused on the general way algorithms work; it was a general investigation of the algorithmic way of thinking.

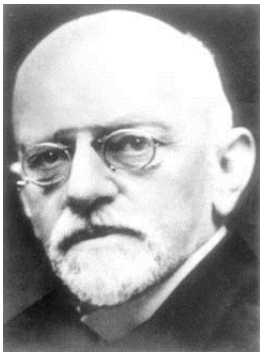


Figure 1.1 David Hilbert

The word computer originally meant a person, who will carry out a computation. The meaning of the word computer as we perceive it today, an electronic device that does performs various tasks, was due to the pioneers of the Mathematical Logic in the 20th century and their realization that computing can be done in a purely algorithmic way [Ceruzzi 1998] and [Davis 1987]. It could be said that Hilbert's (1862-1943) *Entscheidungsproblem*, the "principal problem of mathematical logic" as it was called, was the trigger for many logicians and mathematicians to research the algorithmic way of thinking in general and develop the basis of the Computing Science. The

Entscheidungsproblem is the problem of finding an algorithm that could determine whether a given proposed inference is valid and an algorithm for the *Entscheidungsproblem* could, in principle, be used to give an answer to any mathematical question. Turing's attention to this problem was drawn by Newman's lectures and soon after he was able to prove that an algorithm for the *Entscheidungsproblem* does not exist. It is the tools developed by Turing in his proof, that no algorithm for the *Entscheidungsproblem* exists, that were essential for the basis of the Computing Science. Other equivalent theories appeared in the same time, the most important from these to this project was the development of the lambda-calculus by Alonzo Church (1903-1995), which will be discussed in chapter 4.

Figure 1.2 Alan Turing

In his analysis Turing found necessary to investigate somehow the class of all possible algorithms. Turing started with a human that will carry out all the steps of some algorithm, this human was considered to be the "computer". This computer would be able to read and write on a one-dimensional paper divided into squares and as Turing explains it "the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares." It is worth mentioning that the tape was considered to have an infinitive length. The procedure that this computer would follow consisted of three atomic "simple" steps [Davis 1987, p. 143]:

1.A change of one symbol in one of the "observed" squares, the "observed" squares are the squares that the computer is looking ("observing") at a time.

2. Changes that can be made from the "observed" squares to other squares that are not further away than L squares away, where L is a constant. The computer can only change one square at each step.

3.A change in the state of mind of the computer.

Turing restricted the states of mind or "*m*-configurations" as he called them, so there is only a finite number of states of mind. Similarly there is only a finite number in the states of the paper, this means there is only a fixed number of cells that can be written in a square of the paper. Each state of mind is given in a table, which gives exact instructions for the changes of the three types mentioned before, corresponding to each possible set of "observed" squares and symbols that can be found on those squares. Turing simplified further the instructions given in 1,2 and 3 by allowing the computer to "observe" only one square at a time and to be able to change a square only immediately left or right. The

machine mentioned above is the famous *Turing machine*[1], as first mentioned with this name in 1936 in a paper of Church [Hodges 1988, p. 5]. A Turing machine can be written with a finite set of quintuples, each of them having the one of the three following forms:

paβRq or *paβLq* or *paβNq*

A combination of five symbols like the ones above has the following meaning: if the machine is in *m*-configuration *p* and *a* is the symbol appearing on the "observed square", than the machine will replace *a* by *β*, move one square to the right (*R*), left (*L*) or do not move at all (*N*) and finally change in to *m*-configuration *q*. Every machine constitutes of a combination of five symbols, that determine the way a computation will be made. Turing defined the machines to be deterministic by not allowing any two combinations of five letters for in the same machine to start with same pair *pa*, that means that if the *m*-configuration of the machine is *w* and the symbol on the "observed" square is *e* than the machine would only find one quintuple in its behavior table. Continuing with the same symbols *w* and *e*, if the machine does not find any quintuple starting with these two symbols then the machine would stop, or in more technical terms the machine would "halt", and the computation would end.

Figure 1.3 John von Neumann

A very important development was the idea of how a computer should be constructed. The model developed by John von Neumann was the one that dominated and most computers nowadays are *von Neumann type of computers*[2]. The von Neumann model works by having one central processing unit (*CPU*), a storing device and a connection tube which can transmit one word at a time between the CPU and the storing device [Backus 1978, p. 615].

Figure 1.4 von Neumann model of computer

This idea was brilliant, when it was originally thought of, because it suggested a very simple way the computer can work. The fact though that the connection tube, the "von Neumann bottleneck" as Backus describes it in [Backus 1978, p. 615], is very slow because it allows only one word at a time to be passed between the storing device and the CPU. This means that for a program to execute words have to be passed from the storing

[1] More details about the restrictions of the Turing computer can be found in [Gandy 1998,p. 81].
[2] Information about the von Neumann Model can be found in [Ceruzzi 1997, pp. 6-7]

device to the CPU, then the CPU has answer back and then the program can proceed. This is a very slow way of manipulating data and nowadays there exist computer and programming languages, which escape this von-Neumann type of computer. Parallel computing goes beyond the von-Neumann type of computer since it allows more than one words to be processed at a time.

M.J. Beeson said: "The computer is a completely deterministic machine, and it proceeds in discrete steps..."[Beeson 1988, p.197], this is exactly what Turing has managed to do with the logical development of his machines. This algorithmic way of thinking was the big step in Logic that computing needed to evolve. The big evolvement in Computing in the 20th century was due to the newly evolved cognitive material and not any silicon nor microelectronics! One might argue that if the silicon chips and the micro electronics were not developed, computers would not be as big as they are today, but all this new technology was a consequence of the building of the first mechanical computer that filled up two big rooms and was made out of logical ideas. The mechanisms that a computer works with is not the issue raised in this paper, the issue is the evolvement of logical ideas, that were to create a new science, the Computing Science and more specifically the Software science. This chapter is intended to give an introduction to the reader into the ideas that built the computer and the programming languages, as we know them today.

Chapter 2
Fundamental concepts of programming languages

2.1 Types of programming languages:

In the last half of the century Software has entered almost every home and it has become a science with its own agenda [Mahoney 2000]. Software is the direct product of programming languages, which have been developing rapidly in the last few decades. It is important to make a general investigation in to the different types of programming languages in order to appreciate and understand the technology and the evolvement of the programming languages in the last half of the century. The categorization done here is based on the mathematical models behind the programming languages. It is worth mentioning that some languages can incorporate features of more than of the categories listed below and therefore belong to more than one category. The different paradigms of programming languages are as following:

Procedural languages were the first programming languages to be developed. Symbolic *Assemblers* were developed in the mid 1950's based strongly on mathematical notation [Fisher and Grodzinsky 1993, p. 39]. The first procedural programming language to be widely used was FORTRAN (1954-1957) [Dershem and Jipping 1995, p. 4]. FORTRAN was influenced from the strong mathematical basis that was major in the computing world of the 1950's had an algebraic notation. *Procedural programming languages* are programming languages in which the code is written defining how something is achieved instead of what should be achieved[3]. After the development of FORTRAN many other procedural languages like ALGOL family of programming languages, Pascal (1973), C (1972) and C++ (1984) were developed.

Structured programming languages were the next step forward in producing more coherent programs. These languages provide the programmer with control structures that allow him to write programs in a 'top to bottom' manner. A 'top to bottom' design means that a program can be written in a way that every operation has its place on the code according to when it should be executed. The operation that is executed first should be on the top of the code, the operation that is executed next is underneath and the last operation to be executed is on the bottom of the code. The difference between procedural and structured programming languages is the control structures that are provided to the programmer in order to achieve the 'top to bottom' design of a program. Proceduralism has to do with the commands used in a program, while structured programming implies the order in which the commands are written in a program. It is natural for some programming languages to combine both features of procedural and structured programming languages, which is exactly the case in C.

Here is an example of a 'top to bottom' program in C that calculates the sum, the difference and the product of two numbers and stores them in three variables called sum, difference and product.

```
int sum;
int difference;
```

[3] A more extensive discussion on procedural programming languages can be found in Chapter 3.

```
int product;

sum =  8 + 5;

difference = 8 - 5;

product = 8 * 5;
```

The first three lines initialize the three variables sum, product and difference to be integers. Then the line sum = 8 + 5; assigns the 8+5 to the sum variable, then similarly the difference and the product variables are assigned. The operations in this program are written in the order that they will be executed from top to bottom.

Structured programming languages avoid the command GOTO, which requires the program to go from one command to another that is written above or below the GOTO line without executing the intermediate command lines. A non-structured programming example of the use of GOTO in FORTRAN is the following:

```
100  PRINT ('Type your age')
     READ *, (AGE)
     GOTO 150
140  PRINT ('You are', AGE , 'old ')
150 PRINT ('HELLO')
```

The first line in this example prints to the screen 'Type your age and the next line READ *, (AGE) assigns the user input in to a variable called AGE. The line GOTO 150 commands the program to go to line 150, resulting the program to print on the screen 'HELLO' without executing line 140, which would print on the screen 'You are', then the user input, and then 'old' if it was executed.

A very important feature of programming languages is iterations, which can be achieved with an IF and GOTO pair of statements in FORTRAN. An example of iteration in FORTRAN can be found in [Fischer and Grodzinsky 1993, p. 294, Exhibit 11.4]. Iterations in structured programming languages are achieved with special control statement like *for* in C and *While-end* pairs in Pascal[4].

Functional programming languages are based on the mathematical theory of functions. LISP (1959) is a prime example of a functional language that is based on the theory of recursive functions and has strong relations with the lambda calculus.

Functional programming languages have a nonproceduralistic character, even though they might contain procedural operations. A program in a functional language is written by calling functions.

Functions in programming languages are very similar with the mathematical sense of a function. A function in a programming language takes variables in a more abstract sense than the mathematical use of variables, which are numbers or functions, and returns results or executes a series of commands. In the sense of the variables, functions in programming languages can be thought as the *predicates* of first-order Logic. Predicates in Logic take propositions for variables. In order to appreciate the full extent of the use of complicated functions in programming languages, it would be suitable to employ a higher

[4] An example is in [Fischer and Grodzinsky 1993, p. 294, Exhibit 11.5].

order Logic. It is clear that these similarities with Mathematics are due to the mathematical and logical background of the theoretical Computing Science.

A significant feature of the functional language is *referential transparency.* Referential transparency is the ability to call a function without having any side effects. This means that the internal structure of the function cannot be changed upon the function call. Referential transparency also means that a function will produce the same result, if the data given to the function is the same, no matter where it is in a program. These two features of the functional programming languages, called referential transparency, can improve the efficiency of a programming language, which will be discussed in chapter 2.2.

A program in a functional programming language is a nested collection of functions calls and expressions of the language itself [Fisher and Grodzinsky 1993, p. 40]. It is possible for the programmer to define his own functions by defining the exact variables the function will take, and the exact way the function will operate and produce the result. This feature of the functional programming languages brings programming in to a higher level of autonomy, since the programmer is able to abstract the data further away from the simple operations that procedural programming language provide.

One of the latest developments of the programming languages is object-oriented model of programming languages. It is a combination of *strongly typed programming languages* and functional languages. "A language is said to be strongly typed if all *type checking* that is feasible to do at compile time is done then and all other type checking is done at run time", where "Type checking is the process of determining the type of a specified data object." [Dershem and Jipping 1995, p. 58]. The object-oriented programming languages provide the programmer with a great ability of defining his own data objects and associated function with each object, which are called *methods.* This feature of object-oriented languages allows the programmer to encapsulate the data inside an object [Dershem and Jipping 1995, pp. 334-335]. The first language to introduce the object-oriented model was SIMULA 67 (1967), but the object-oriented model became popular by Smalltalk (1971-1980), which used SIMULA 67 as a foundation [Fisher and Grodzinsky 1993, p. 38].

Logic programming languages are based on the theory of Logic and Set theory. The logic programming languages often allow programs to be written in a very similar manner with symbolic Logic. The way programs work in logic languages is very similar to the way theorems are proved in Mathematics. This is exactly one of the purposes of Logic, to examine if the way a conclusion was deduced from a set of given propositions is valid. Logic languages do not always implement the classic monotonic Logic. PROLOG (1972) for example implements a non-monotonic Logic as illustrated in [Gillies 1996, pp. 75-79].

The logic programming languages work in a very similar way with Logic; the programmer inputs the data and the predicates or relations and expects the program to verify if his query or the argument is valid or not. The programming language contains some system, which should make an automatic deduction and produces a result of whether the argument was valid or not. In this sense logic programming languages differ from all other programming languages, since the programmer will expect the program to verify his argument, while in non-Logic languages, the programmer would expect a program that performs a serious of tasks for various reasons. It might be the case for a

program in non-Logic programming language to draw pictures on the screen without verifying any argument. Logic programming languages are commonly used in applications concerning timetables and route problems.

2.2 Language Evaluation Criteria:

Until the mid 1960's computer scientists thought the most important thing for a programming language was its reliability and did not concern themselves that much whether a programming language had a good factor of readability and writability. It was that time that the Software industry started having more and more commercial applications. The Software in early computing days was produced usually by a very small number of people in each programming team, but this is far away from the manner Software is designed nowadays. Software companies now have a big number of programmers working on the same project, which means that the code has to be read by various people, who have to implement different parts of the project. The size of the code has increased considerably, since programs nowadays can take up to a few million lines of code, while the original FORTRAN compiler was only 25000 lines of code. These are the reason why the readability and the writability of a programming language are very important.

This chapter is going to introduce a set of evaluation criteria that will be used as a basis for describing programming languages. This set of criteria is definitely subjective, but its scope is not to mark a programming language, but present the various ideas behind each language in a historical manner.

The three main criteria are the following:

- Readability
- Writability
- Reliability

For each of these three criteria different factors can be introduced in order to explain and evaluate the programming language. It is obvious that these factors are not necessarily going to belong only to one criterion; some of them have to do with two or even three of them.

In the last twenty years programming languages become more and more readable. Languages like BASIC (1964), which was an educational language based on FORTRAN (1954-1957) or even C (1972), which was a systems programming language based on ALGOL 68 (1963-1968), can be read by people that have some programming experience, but do nor necessary know the particular language. Readability was something that computer scientists did not consider in their original designs for programming languages. It is characteristic to mention that von Neumann considered that the perfect and the only universal language was the machine language. The machine language constitutes out of zero and ones ordered in a particular series, but it is almost impossible for the human brain, maybe not von Neumann's, to understand the internal logic of this sea of zeros and ones [Breton 1991, p. 219].

A first effort to make a language more readable is the language designed to program the Manchester MARK I (1948)[5] by Alan Turing. Turing wrote a manual that had some sort of shorthand code for the coders to write the programs with the use of the keys of a teleprinter instead of the huge series of huge series of binary numbers. It is important to mention that the readability of that language is very far from what we would

[5] The MARK I started with the support of the Royal Society in 1946.

imagine as readable language now, it can probably be compared with assembly languages, that are used to today for systems programming.

To evaluate if a language is readable, the following factors should be examined:

- Syntax design
- Simplicity and Orthogonality
- Control structures

The syntax design is the form of the elements that the programming language constitutes of [Sebesta 1996, p. 13]. The design of the syntax of a programming language can strongly affect the readability of the language. Two examples of different syntactic design are the following:

- Identifier forms: It is important for the design of a programming language not to restrict the length of the identifiers. If the identifiers are not restricted then the program can be written in a way that the identifiers imply their function. An example in C++ that can be understood even by people who are not familiar with the language is the following:

```
int t;

int temp;

int temperature;
```

These three different initializations of a integer (*int*) variable are meant to hold the value of the temperature. The first initialization is very unclear, since 't' does not imply much about the variable. The second one, 'temp', is more obvious to the reader, but it can still be confused with temporary. The third one 'temperature' is unambiguous, on what it means. This very simple program is easy to read because of the ability the language has for identifiers to be of any length. This feature of the language makes it definitely readable, but in the early computing years, computers did not have much memory available so identifiers had to be short. An extreme example of this is (ANSI) BASIC, which only allowed a single letter or a single letter followed by a single digit [5. Sebesta1996, p. 13].

- Grouping statements: The way the syntactic design deals with the grouping of statements, this can be when a *if* statement ends or when a loop ends. C (1972) and C++ use braces for all grouping activities, which maybe simple to implement, because the programmer does not have to remember different types of grouping statements, but it can be extremely hard to read. For example this C function has a number of *if* and *else* statements and it is very difficult to see where each one ends.

```
if (x>0)
{
if (x>100)
{
……………
}
```

```
else
{
……
}
}
else if (x<0)
{
if(x<-200)
{
……
}
else
{
……
}
}
```

In Pascal (1971) this problem is not that extensive, because Pascal uses instead of the braces *begin-end* pairs, which would make a program more readable than this C example given above. The following example code illustrates the readability of a Pascal program concerning the grouping of statements:

```
if X>0 then
    begin
    if X>100 then
     ……
    else
    ……
    end;
if X<0 then
    begin
    if X<-200 then
    ……
    else
    ……
    end;
```

In this example the reader can easily identify where a grouping ends, because of the use of the *begin-end* pairs. The syntax considerations affect, apart from the readability of programming languages, its writability, which will be discussed later in this chapter.

Orthogonality and simplicity are two factors that are very close related to each other. Continuing on the discussion on Pascal, it was mentioned before that Pascal uses *begin-end* pairs for grouping statements, but this is not true in all cases, for example in the repeat statement the *begin-end* pair can be omitted. This is characteristic of the age of this programming language (1971) and its lack of orthogonality. Orthogonality, as [Dershem and Jipping 1995, p. 43] explains it, ‘refers to the interaction between concepts

– namely, the degree to which different concepts can be combined with each other in a consistent manner'. Orthogonality is lacking in the case of Pascal, because there are exceptions, where the *begin-end* pair does not have to be used. This lack of orthogonality makes the language less readable since the reader has to know the different exceptions of the *begin-end* pair rule to be able to understand the program. C and C++ are orthogonal in this sense since the braces rule applies always[6]. An orthogonal language requires less exceptions in its set of rules and thus it has a higher factor of simplicity, since the programmer and the reader have less rules to remember.

In 1968 a big step forward was done with the completion of ALGOL 68, which introduced the structured programming. The structured programming languages contain control statements in order to achieve the 'top to bottom design'. Languages like BASIC and FORTRAN use the *goto* control statements. Because of their limitation in control statements the code produced in languages like that is called today 'spaghetti programming'. The ability to write code without jumping from one statement to the other without any logical order increases the readability of the program. This 'top-down' design can be achieved in languages that rely on the *goto* control statements if some restrictions in the use of the *goto* statement are set. The restrictions are as follows:

- The goto statements must be before their target, except when used to form loops.
- Their targets must never be too distant.
- Their numbers must be limited. [Sebesta 1996, p. 12]

Most languages after the late 1960's had enough control statements in order to avoid unstructured programming, especially after the standards ALGOL set in 1968.

The evaluation of a programming language cannot be done only based on the readability of the language. Other factors, like how easy it is to write a program, have to be examined to evaluate a programming language. Some of the factors mentioned before do not only affect the readability of a programming language, but they affect the writability of the language.

Orthogonality and simplicity can be viewed in terms of the writability of a programming language. A programming language that has a lot of different constructs, different rules and many exceptions in these rules cannot be described as an orthogonal and simple language. Languages like that are hard to use and programmers often do mistakes in their use of different constructs, since it is hard to remember the big set of constructs. As [Dershem and Jipping 1995, p. 43] mentions "violations of orthogonality occur when two concepts cannot interact with each other or when they interact in a manner inconsistent ". If these violations in orthogonality results to a violation in the validity of the program then a rule has to be introduced to avoid a case like that. With the introduction of new rules the language becomes not only less orthogonal, but harder to write code as well.

Support for abstraction has become a very important factor of the writability of a programming language in the last thirty years. Abstraction in a programming language is the idea of constructing a complex structure or function that will allow many of the details to be overlooked. Data abstraction can be found in the *enumeration types* of Pascal[7], but it was languages like C and ADA (1979), which was partly based on Pascal[8],

[6] One might argue that in single line *if* statements braces can be absent. Not putting the braces though is a bad programming practice, since the program will be valid with braces and more readable as well.
[7] For more information see [MacLennan 1983, pp. 188-190].

that set the standards for data abstraction. The data abstraction makes a language easier to write code for, since logical mistakes, which will not be detected by the compiler, can be avoided and the ability[9] of the programmer to ignore details after having abstracted a complicated structure or a function. An example of this is the following: A programmer creates a structure for the human. This structure will hold its sex, age, height and weight. When the programmer wants to use this structure, then the structure automatically will have a place to store the sex, age, height, and weight details of the human. In C this would look like this:

```
struct human
{
char sex[6];
int age;
int height;
int weight;
};
```

Data abstraction is definitely a very important factor of the writability of the language, because the Software that is being produced nowadays is of great complexity and it often describes or simulates real situations.

The final factor that affects writability is the expressiveness. A language has to give the ability to the programmer to be able to represent data structures and procedures in a natural way. This makes the language easier to write since the names that are given to various procedures or data structures imply their functionality or meaning, which makes a program more readable as well. The expressiveness can have as bad result the loss of simplicity. There is a fine line which between expressiveness and simplicity, which can make a language successful or not.

For a programming language to be successful readability and writability are not enough, the language has to be reliable. With the term reliable it is meant that the language will perform under its specifications at all conditions. This does not imply that the compiler[10] of the language will produce always a valid program, since the code might be itself invalid.

For a programming language to be reliable, it has to have a type checking facility that does not allow type errors to go through the compilation process[11]. If the type checking facility does not check correctly, then the program might have a 'run-time' error, an error which occurs while the program is being executed, which may produce unexpected results. This facility has always been in the center of the attention of the language designers even though some languages did not succeed to perform a very good type check[12].

Another facility, which was not supported in programming languages until ANSI's development of IBM's PL/1 in 1976, is the exception handling. Exception handling is the ability of a program to intercept unusual conditions while it is executing,

---

[8] For more information see [MacLennan 1983, pp. 270-311] and [Dershem and Jipping 1995, pp.184-192].
[9] Example in [Dershem and Jipping 1995, pp. 188-189].
[10] For information on compiler process see [Dershem and Jipping 1995, pp. 36-41].
[11] The type checking can be done while the program is executing, but this is time-consuming, so the checking during compilation is preferred.
[12] An example of bad type checking in [Sebesta 1996, p. 16].

correct them if possible, and then continue. This facility in a programming language can be a great tool for its reliability.

Aliasing is a facility of a programming language that can put its reliability in great danger. “Aliasing is the ability to reference the same location by two different names” [Dershem and Jipping 1995, p. 147]. This problem can occur, when a variable in a *local* environment has the same name as one that is *global*. Then if function calls one or both of them in the local environment, the compiler will not know, which one to use. Some programming languages use aliasing in order to solve data abstraction problems they have. In order to avoid problems caused by aliasing many programming languages nowadays avoid or limit the use of it.

Most programming languages do not fulfill all the criteria mentioned above, but that does not mean they are not worth to be looked at. On the contrary, languages that do not fulfill all the criteria are very interesting from a historical point of view, since the evolvement of the ideas of programming languages can be investigated. This is exactly the scope of this project, to present the history of ideas of two programming languages that made Software, among other programming languages, a science of its own.

Chapter 3
Procedural programming languages versus nonprocedural programming languages

A procedural set of instructions can be defined as an operation that describes the way something is done, describing the steps one by one. A nonprocedural operation can be explained as an operation that instead of describing the way something is achieved, it describes what it is that should be achieved through a particular operation. It is the question between how and what, that distinguishes proceduralism and nonproceduralism. This distinction though is not unique; it is related to a context. To be more precise this context is defined as the *level of proceduralism*, which can be explained as what it is perceived as a simple operation in a set of instructions that can be executed without further explanation. The level of proceduralism will be a deciding factor for the distinction between proceduralism and nonproceduralism.
An example of this is the following set of instructions for making a cup of coffee:

1. Boil water
2. Put a spoon of coffee in the cup
3. Poor the water after it has boiled in to the cup with the coffee.
4. Stir the water in the cup with the spoon.
5. Remove the spoon from the cup.

It should be clear that this is a procedural set of instructions for the coffee, even though it can be argued that boiling water is not procedural since we do not define how water should be boiled. This is where the level of proceduralism plays an important role. In this example boiling water and stirring the water should be considered as operations that are simple enough and they do not require further explanations. Obviously if the level of proceduralism was set lower, then the instruction 'boil water' should be further analyzed in order to have a completely procedural set of instructions.

The relation of proceduralism and nonproceduralism can be viewed as the relation between language and metalanguage. A nonprocedural operation has its basis to a procedural set of instructions. Nonproceduralism cannot exist by itself, there has to be a procedural basis for it. The similarity can be seen in language, where the language is the basis for the metalanguage. The metalanguage cannot be 'talking' about a language, if the language does not exist in the first place. The concept of a level of proceduralism allows us to know in which level we are, and then we can make a distinction between a procedural and a nonprocedural operation. If this level did not exist, it would make no sense to discuss proceduralism. This can be similarly observed in Logic. If the distinction between Logic and *Metalogic* is not offered, then this can lead to paradoxes. A famous example of this is the following: "A Cretan says: "All Cretans are liars". Is this true or is this a lie?" If the answer is that it is true then the Cretan is a liar, which means that he is lying, so the answer is not true. But if the answer is not true then he is a liar and so saying the truth. This is a vicious cycle, which leads us to a paradox.

In the early days of the Computing Science proceduralism was very important since the computer, human or machine, executed instructions without any intelligence involved. The instructions given to a computer had to be completely procedural in order to achieve a computation. The programmer then had to be very precise on every

instruction he wrote in his program, which made his job very hard[13]. Computer scientists were in need of tools that would make programming easier for them in order to achieve programs of great complexity. This step was taken forward by programming language designers. Programming languages started supporting nonprocedural operations, which made large programs easier to program. It was the division of the programming languages in two parts, the *low-level* procedural *programming* of the compilers and the *high-level programming* of the applications, which took programming a step further.

The notion of a level of proceduralism is significant in this division of the programming language. The low-level design of compilers is procedural since the instructions are describing how the computer should execute a command. It can be argued though that the low-level commands are not procedural, since they do not define how the electricity will be passed between the computer circuits, this is exactly were the level of proceduralism is set. The level of proceduralism is set to a level, which is higher than the electrical activities inside the computer, since this is of no interest to the programming language.

A procedural programming language gives the ability to the programmer to be more exact by specifying how the program will achieve its aim. This ability can be very significant when programming a compiler, but of less significance when programming a general-purpose application. The proceduralism of low-level programming languages offers control by trading off the ability to produce very complex programs. A programmer, who is using a procedural programming language, will find it hard to bind data together in order to produce some sort of data *abstraction*[14], while another programmer, who is using a nonprocedural language that supports data abstraction, can very easily define an abstract type of data writing a command of what to do. That is exactly what the separation of programming in to compilers and applications offered. The procedural approach to the data abstraction gives the programmer more control to define exactly how the data will be bind together, which can result to a more efficient use of the computer's memory.

The effects of proceduralism in programming languages do not only concern data abstraction, but the whole approach to the design of a program. The programmer can take proceduralism in to a higher level by defining his own functions of what to do or even his own structures of data. This opens up new horizons of what computers can achieve. A program, with this nonproceduralistic approach, can process complex data and produce results far away from simple computations. The nonproceduralism of the programming language is based on procedural programmed compilers, which are programs of great complexity that support the nonproceduralistic approaches of the programming language. Nonproceduralism springs from complex proceduralism, in other words simplicity comes from complexity.

The ability of a compiler to produce correct results does not depend only on its complexity, but on the quality of the programmer of the compiler. Simple compilers designed for procedural languages can contain the same amount of errors as very complex compilers designed for nonprocedural languages.

---

[13] This does not suggest by any means that programs of great complexity were not produced. A prime example of this is the work of Max Newman and Alan Turing at Bletchley Park for the Collossus project.

[14] "An abstraction is a representation of an object that hides what could be considered as irrelevant details of that object, thus making use of the object easier."[Dershem and Jipping 1995,p. 134]

To understand better this concept of proceduralism, it is worth to analyze the way a vending coffee machine works[15]. The coffee machine produces two types of coffee, normal and cappuccino. The user can also input the amount of sugar and if he wants milk in his coffee. For the user level of proceduralism the machine is nonproceduralistic, since he inputs his choices by pressing some buttons, which indicate what coffee the machine should make. On the other hand for the programmer's level of proceduralism, who designed the machine automation, the commands of how to make the coffee are procedural, because he instructing the machine of how to make a cup of coffee. These commands though are completely nonprocedural, if the level of proceduralism is considered to be the exact way the machine works, how the electrical signals are passed between the circuits of the machine. For this person, who built this coffee machine it can be said that the proceduralism can be found in the operations of the machine. These operations are procedural since they define how the machine electrical and mechanical parts operate. This example shows the different levels of proceduralism of a simple vending coffee machine. It shows that the very simplistic and nonprocedural appearance to the user comes from a more sophisticated procedural background.

Simplicity through complexity is the basic idea behind many complex programs. The complexity is initially encapsulated in the compiler, and later on in the actual program. The machine language commands that define a function are hidden from the programmer in the compiler program, so the programmer can use functions, avoiding the low-level details that he does not need to know. It is the ability that is given to the programmer to avoid the details of 'how to do something' and concentrate on 'what it should be achieved', which brings great results in computing, results far away from what computer scientists in the 1930's and 1940's thought of computation.

[15] An explanation of automation theory can be found in [Shields 1987].

Chapter 4
The lambda calculus and its involvement with programming language design

In the beginning of the 20th century mathematical logic was not well formed as it is today. The basis of Mathematics and Mathematical Logic were shaken by Russell's paradox, the Burali-Forti's paradox of the greatest ordinal and Cantor's paradox of the greatest cardinal [Grattan-Guinness 2000, pp. 334] in the early 1900's[16], and plenty of research was done in order to build a consistent basis for Mathematics, one that would provide Mathematics with rigor and would avoid all paradoxes. The search for a system that would be suitable for this basis was done in many branches of logic. This paper though will only include the ones that are highly relevant with computer science. Some mathematicians undertook the study of the general class of recursive functions and function abstraction in general. Among them the most important results to the Computing Science came from two people, Alonzo Church (1903-1995) and Alan Turing. The main idea of Turing's system, the Turing machine, was explained in chapter 1. Other equivalent systems came from Stephen Kleene (1909-1994), Emil Post (1897-1954), and some work in Combinatory logic[17] by Moses Schönfinkel, Haskell Curry (1900-1982) and John von Neumann was done in this area in the 1920's and 1930's. The discussion on this chapter will be on the system Church developed in Princeton in the 1930's. This system has now evolved and it is known as the lambda calculus.

In the 1930's Princeton University had some of the finest logicians around that time. Among these, Kurt Gődel (1906-1978) and his work, especially the Incompleteness Theorems, influenced many others. In 1934 Gődel was giving lectures in Princeton entitled: "On undecidable propositions of formal mathematical systems". In these lectures Stephen Kleene, whose advisor was Church, was taking notes in order to develop the Gődel's definition of recursive function in to a mathematical theory. Kleene published his theory in 1936 ([Kleene 1936]). A few years before that, Church started developing a formal system for logic that would avoid the use of *free variables*[18] and limit the use of the *law of the excluded middle*, which he believed were the two causes of the construction of paradoxes [Aspray 1980, p. 121]. In 1932 he published his system in the Annals of Mathematics [Church 1932] and a revised version in 1933, because his first system admitted a form of paradoxes [Church 1933]. Unfortunately even the 1933 system was found inconsistent by Kleene and J.B. Rosser(1907-1989) in their 1935 paper [Barendregt 1992, p. 118]. Church in 1941 gave a consistent version of the lambda calculus that deals only with the functional part[19].

[16] Information on these paradoxes can be found in [Grattan-Guinness 2000, pp. 311-315].
[17] More information about Combinatory Logic in Chapter 2 in [Hindley and Seldin 1988].
[18] "We will say that a variable is bound by a quantifier (linked to it) when it is in the scope of a quantifier and it alphabetically matches the variable in the quantifier… When a variable is not bound we will say that it is free"[Guttenplan 1997, p. 183].
[19] This part is known as the λ *I*-calculus[Barendregt 1980, p. 4].

Figure 4.1 Alonzo Church

Church was disturbed by the way mathematicians used formulas and equations without caring about their domain of definition. This careless use of formulas and equations can lead to paradoxes and invalid proofs in Mathematics. An example, which was most likely very influential in Church's work, that does not care about the domain of definition is Gödel's intuitive proof of his incompleteness theorem. The theorem is proved with a mixture of Logic and Metalogic, where a proposition is declared to be not derivable. Then by listing and using a technique similar with Cantor's diagonal argument, it is proven that this proposition is derivable if and only if it is not derivable. This way Gödel proved that the theory, which he originally assumed to be consistent, is incomplete[20]. Gödel's intuitive proof does not care about the domain of definition of the theory, since it mixes Logic and Metalogic.

To give a solution to this problem Church defined in the lambda calculus functions with their domain of definition built in to the function formula. He introduced the basic notation of a function in the lambda calculus in the following way: $\lambda \underline{X} \, |\underline{M}|$[21], which represents the function whose values are given by the formula M. An example of function definition in the lambda calculus is the following example:

$\lambda x.x^2$

This would be defined in the calculus like this: $f(x) = x^2$.

The $\lambda \underline{X} \, |\underline{M}|$ notation was the only way a function could be individuated in the lambda calculus. A function was said to be *λ-definable*, if the function could be defined within the lambda calculus.

Church introduced a set of rules in order to define the well-formed formulas in his calculus. These three rules are:

"• A variable x standing alone is well-formed

• If F and X are well-formed, F{(X)} is well-formed

• If M is well-formed, so is λx[M]"

[Aspray 1980, p. 124]

Church defined three operations for his lambda calculus: "

[20] This proof can be found in the first part of [Gödel 1931], while the second part of this paper contains the proof using the Gödel numbering technique. A book describing Gödel's proof is [Nagel and Newman 1959].

[21] In the notation of Church (λxM).

I. To replace any part ***M*** of a formula by $S_y^x M$ |, provided that ***x*** is not a free variable of ***M*** and ***y*** does not occur in ***M***.

II. To replace any part ((λ***x M***)***N***) of a formula by $S_y^x M$ |, provided that the bound variables of ***M*** are distinct both from ***x*** and from the free variables of ***N***.

III. To replace any part $S_y^x M$ | of a formula by ((λ***x M***)***N***), provided that ((λ***x M***)***N***) is well-formed and the bound variables of ***M*** are distinct both from ***x*** and from the free variables of ***N***." [Church 1941, p. 12]

Any finite sequence of this operation was named a conversion. If B can be deduced to A according to these rules, then A is convertible into B, or *A conv B.* "A function of F of one positive integer was said to be λ-definable if it were possible to find a formula F such that, if F(a)=r, then F(m) conv r." [Aspray 1980, p. 125]

Church after a substantial amount of work began to believe that every λ-calculable function is *effectively computable.* After many years of research it was proven that this was true. Turing included a result, which stated that λ-definability is equal with Turing computability in the appendix of the famous "On Computable Numbers, with an Application to Entscheidungsproblem" [Turing 1936].

All this work originally triggered by Hilbert's Entscheidungsproblem was building a theoretical basis for the computer science. Research between the 1940's and the late 1950's was very little and the most important part of this work was that lambda calculus became more accessible to non-experts, especially with the publication of Church's 1941 book, which was mentioned previously. Some technical advances were also made in this period, but they are of very little significance compared with the advances made in 1930's. Work in the function abstraction became very popular in programming languages especially after the 1960's and created a new branch of programming languages, based on this mathematical model of function abstraction, called functional programming. A prime example of this is LISP, which converted many of the features of lambda calculus into a programming language.

LISP (LISt Processing language) was developed between 1956 and 1962 by John McCarthy. It is a pure functional language and it is based on the idea of recursive functions [MacLennan 1983, p. 343]. LISP has a universal function that can interpret any other function, a concept very similar with the *Universal Turing Machine.* The basis of LISP was lambda calculus and a function definition in LISP is very similar with the lambda calculus. Instead of using the Greek letter λ, it uses LAMBDA as illustrated in the following example:

In lambda calculus: λx,y.x+y
In LISP: (LAMBDA(X Y)(PLUS X Y))

LISP inherited many elements of the lambda calculus and that was one of the reasons of its syntax to be a model of simplicity [Sebesta 1996, p. 51].

Previous to the development of LISP FORTRAN was the only high level programming language. FORTRAN did not contain any parts of the lambda calculus or the recursive function theory. It is a *von Neumann type of programming language* as described by its creator John Backus (1924-) in [Backus 1998]. FORTRAN did not have any elements of the structured or the functional programming. A description of the development and some features of FORTRAN will be discussed in chapter 6. Backus in

his lecture for the Turing Award in 1977 [Backus 1978] suggested another model for programming languages named FP (Functional Programming)[22], which is 'liberated' from the von Neumann style of programming languages. The model he suggested in 1977 was a functional language that incorporated many of the features of the lambda calculus like the composition and construction of functions and an elementary substitution rule. Backus describes the power of the ability to express functions in a programming language in the form of the lambda calculus functions. He explains that this freedom can lead to chaos, if different forms are combined to suit the occasion without learning the properties of the few forms that are enough for all purposes. His effort was to describe a model or an algebra of programs for FP systems that would incorporate features of the lambda calculus, but in a more simplified and effective way for programming than the lambda calculus.

This description of FP systems can be compared with Konrad Zuse's (1910-1995) Plankalkül that was developed thirty years prior to Backus' Turing award lecture. Zuse was trying to simplify and apply the logic and the recursion functions theory in to the design of a calculus of programs (Plankalkül) and a machine that would realize all these features in its hardware (logic machine). Plankalkül will be the subject of discussion in the following chapter.

[22] Information about FP can be found in [Dershem and Jipping 1995, pp. 261-277].

Chapter 5
Konrad Zuse and Plankalkül

"On one side there are sometimes too many mathematicians in influencing the computer science in a worldly innocent manner. On the other side, relatively primitive methods and programming languages are still applied in practice." [Zuse 1976, p. 11]

Konrad Zuse (1910-1995) studied Civil Engineering in the university of Berlin, where he graduated from in 1935. Most of his life he was considered with constructing computers, the famous Z computers. At the end of the Second World War in 1945 he was unable to continue the construction of his computers and he concentrated on his work on the theoretical Computing Science and the development of a programming language. Konrad Zuse invented Plankalkül (Calculus of Programs) between 1942 and 1945. Zuse may have been influenced in the universal concept of Plankalkül by the philosophical motivations of his compatriot Gottfried Leibniz (1646-1716) and his idea of a *charecteristica universalis*, since he has already been influenced by Lebniz as Wolfgang Giloi mentions: "Zuse received the inspiration to use the binary system from the *Dyadik* of G.W. Leibniz (1646-1716), …"[Giloi 1997, p. 17]. In his efforts Zuse described what he intended to be a universal algorithmic language that would be the theoretical basis for every computer. He continued his work on this basis by explaining how this Calculus of Programs would be implemented in mechanical and electrical relays.

Figure 5.1 Konrad Zuse

He was planning to submit his Ph.D. dissertation under Professor Alwin Walther[23] (1898-1967), which would include Plankalkül and how this Calculus of Programs could be applied to a general-purpose machine[24]. Plankalkül was not published until 1972 [Zuse 1972]. The main reasons for this are that Zuse met some disbelief from other colleagues and he thought that Plankalkül would not be understood and appreciated. This disbelief though was not unreasonable, since the language was rather universal for the specific applicability Computer Scientists of the 1940's and 1950's expected. In this concept of universality the language provided a large amount of instructions, which made many scientists uncertain of its ease to learn and use it. It can be compared with PL/I and its large set of instructions. Information about PL/I can be found in [MacLennan 1983,

[23] I would like to thank Professor Raul Rojas of the Freie Univerisität von Berlin for providing this information.

[24] Alwin Walther was a professor of Applied Mathematics (Praktische Mathematik) in the Darmstadt University. In 1930's he created the Institute für Praktische Mathematik (IPM) in Darmstadt.

pp. 95 - 96]. The ambiguous two-dimensional notation definitely played a role on that skepticism. Zuse used the following notation to describe variables:

	Q
V	2
K	4
S	6.o

These lines define the variable Q2, 2 is the variable-number and it is written in the next line that starts with V. The line starting with K denotes the component-subscript of this structure, which has six members of o. o is the primitive type of the single bit. In modern notation in C this would be written as Z2[3][25] of an array, which is previously defined. The lack of a type checking system among other ambiguities would have made the production of a coherent compiler very hard, which is an enormous amount of work, as it will be discussed in the next chapter in the case of FORTRAN.

Zuse's programming language however was very advanced for its time and contained features that became popular in programming languages at least a decade later. The main characteristics of the language are described here:

"The first principle of Plankalkül is: Data processing begins with the bit" [Zuse 1976, p. 9]. Complex data structures could then be built from the single bit. For the representation of decimals numbers he used a hidden bit, which contained the 'dot'. His plan was to create a language that would support complex structures. Iteration is achieved in Plankalkül with the *while* command. An important feature of his programming language is that in some respect it is structured, since it does not support the *goto* command and it contains the concept of subroutines. The distinction whether Plankalkül is structured or not is not a straight forward task, since the language's notation was two-dimensional and ambiguous. This would require a further investigation into the exact details of the language. The use of Boolean variables, Ja-Nein Werte (True – False Values), is characteristic of the influences of Logic on Zuse, as it will be explained later in this chapter.

His notation was very mathematical and does not resemble any of today's high-level programming languages. He borrowed the notation from the Predicate Logic and he used the conjunction ($\wedge$), the disjunction ($\vee$) and the implication ($\rightarrow$) as his logical connectives in Plankalkül. The use of the existential and the universal quantifier is common in Plankalkül and the notation is the same as in [Hilbert and Bernays 1939]. Zuse's intentions were to construct a programming language that would be strong enough to solve every mathematical problem and would be easily designed in hardware "make it accessible for engineers". He was aiming to build a logic machine, as he called it, which would implement the ideas of his Plankalkül. This logic machine would be able to deal with more general problems than the algebraic machines he had built before. Zuse went on with his idea of applying his Calculus of Programs direct on to the hardware and designed series of relays for the conjunction, disjunction and negation[26]. This logic machine was never completed.

[25] The indexing in an array in C starts from zero and not from one.
[26] This was included in his planned Ph.D.

Zuse's Calculus of Programs used thoroughly the operators μ, λ influenced by Hilbert and Bernays (1939) as mentioned by Zuse himself in [Zuse 1972, p. 38]. The operators μ, λ were used in Plankalkül in order to find terms of a list or a property. Using the definition of Zuse [Zuse 1972, pp. 38 - 39] the operator μ is defined as follows:

$$\mu x \; R\square(x)$$

The next term of the property $R\square$, if that does not exist, then the closing symbol will be given.

The operator λ is similar with the operator μ with the only difference that the search for the next term begins from the last term.

Hilbert and Bernays borrowed the λ symbol from the lambda calculus [Hilbert and Bernays 1939, vol.2, p. 462], which means that Zuse was aware of the development of the lambda calculus[27]. Even though Zuse had studied logic in [Hilbert and Bernays 1939] and the recursion theory is explained in Chapter 7, pp. 279-382, he decided that the use of functions in Plankalkül would be non-recursive. In this book most of the important work in the foundations of Mathematics is described and some of the work of Curry and Church is mentioned [Hilbert and Bernays 1939, vol.2, pp .416-417]. Zuse interests were very different though; all his efforts were designed to be directly applicable to the computer. His idealism was to bridge the gap of the work of the mathematicians and engineers. In other words his purpose was to design a mathematical model specific for computers that could be without difficulty implemented in to a general-purpose computing machine.

Zuse did not manage to produce a programming language that would be popular, rather the opposite as mentioned in [Bauer and Wössner 1972, p. 678], who are trying in their article to diminish the widespread ignorance about the Plankalkül. The language was finally realized and a compiler for it was built in 2000 by the team of Professor Raul Rojas in Germany. Plankalkül though contained many features of contemporary programming languages like the *while* construct, the use of functions and subprograms and the structuring of objects, but most important is Zuse's effort to produce a universal programming language. In this effort he introduced one of the most important ideas in computer science nowadays, the idea of *modularity*. Modularity is the abstraction of the hardware by several layers of software that will make the hardware transparent.

This is exactly where the concept of proceduralism is significant. Zuse efforts were to incorporate the different layers of abstraction with a simplistic design of a machine (the logic machine) that would make programming non-procedural. With the ability in Plankalkül to build complex structures and functions, the programmer could program in a high-level non-procedural manner without having to know the exact specifications of the hardware. This is of course after the abstract structures and functions have been built from the very simple and basic ones in a procedural way. This is the big step forward Computing took many years later by dividing itself in different layers according to the level proceduralism. For this to be understood better imagine that a computer in a simplified form has three levels of proceduralism, the lowest level of the processor's instructions, which provides functionality to the designer of the operating system, which in its turn provides high-level functionality to the programmer that will

[27] It is very unlikely that Zuse was unaware of the developments in recursion theory, since many of the main figures in this area of Mathematics have been in Göttingen for at least a short period of time. Curry was doing a dissertation under Hilbert and Church was visiting in 1929[Grattan-Guinness 2000, p. 453].

create applications like a word processor or statistical application that will be executed on this operating system. In order to design the complete Software a computer requires to execute applications three different kinds of programming are needed, which are usually done by three different programmers, each specializing in one area.

Unfortunately this idea of modularity was met with great disbelief, when Konrad Zuse tried to make his language known to the Computing Community. It was only after many years that Computing scientists understood it well and saw its importance in the evolvement of the Computing Science itself.

Chapter 6
FORTRAN

"Can a machine translate a sufficiently rich mathematical programming language into a sufficiently economical program at a sufficient low cost to make the whole affair feasible?"[Backus 1998, p. 69]

During the 1940's and the 1950's the need for a programming language was great. The cost of operating a computer was very high, because people were needed to program it and without the existence of a programming language most of their and the computer's time was spend programming and debugging. A programming language that would make programming quicker and error free would be ideal for that time. The first efforts appeared under the name 'automatic programming'. Most of them considered themselves with indexing and floating-point operations providing a language very similar to the machine language or a library of sub-routines. Because of the scope of this paper only three of them, who ere influential to the design of FORTRAN, will be mentioned and the other ones can be found in [Knuth and Pardo 1977].

The Laning and Zierler system was working in 1953 and it was on of the first working compilers. It was designed for the WHIRLWIND computer at MIT and a manual was published for it in 1954 [Laning and Zierler 1954]. The system was "rather elegant, but simple one" [Backus 1998, p. 68]. Its input was a simple algebraic language. In early 1954 the A-2 system was working as well. It was mostly concerned with fixing sub-routines and the language used for it was a mixture of compiling instructions in machine code and abbreviated words that would perform floating-point operations or other operations that were not directly supported in the hardware. After the May of 1954 the language for the A-2 was improved to resemble more other systems that were working at that time. Finally an important influence must have been the Speedcoding for the IBM 701. It designed by IBM's New York Scientific Computing Service and it was working in 1953. A description of this system was published in 1954 in the first Journal of the ACM [Backus 1954]. This provided Backus with some useful experience in the 'automatic programming' systems and some idea of what was really needed for a programming language[28].

The disbelief though was great since all the 'automatic programming' failed to prove that a compiler can produce programs close to those humans can produce. To appreciate how this disbelief dominated the computing world the following remark should be made. In the second edition of the first programming textbook [Wilkes et al. 1957], a chapter was added for the automatic programming, which states that experienced programmers could produce more efficient code than any 'automatic' programming system[29].

[28] The team of FORTRAN and most of the Computing world, as Backus clearly states about Plankalkül "Like most of the world …, we were entirely unaware of the work of Konrad Zuse" [Backus 1998, p. 69]. Backus continues describing Plankalkül as "… a more elegant and advanced programming language than those appeared 10 and 15 years later". A comparison between Plankalkül will be made in the final chapter.
[29] A passage of the first programming book that describes this can be found in [Knuth and Pardo 1977, p. 260].

FORTRAN came in a time where the need for a programming language was great. It had to overcome the difficulty of proving that the compiler can produce good results. This is the reason why the designers of FORTRAN concentrated on producing a combust compiler that will produce correct resulting programs and did not concentrate on the design of the language. The design of FORTRAN is summed up very nicely by Backus' own words: "As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler that could produce efficient programs." [Backus 1998, p. 70]

FORTRAN was originally designed for the IBM 704, which supported floating-point operations. Floating point and indexing operations, as mentioned before, were the main concerns of most the 'automatic programming' systems. FORTRAN had the good fortune not to have to deal with any of those two operations since it was supported in the hardware, but as it is well known every new piece of hardware has its problems.

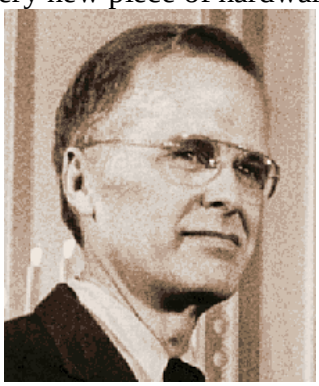


Figure 6.1 John Backus

IBM Mathematical FORmula TRANslation was originally suggested to IBM by John Backus in late 1953. IBM saw the need for a programming language and gave their permission and support to Backus and his team to design a programming language. The design of FORTRAN began in 1954 in New York. The original team constituted out of John Backus, Irving Ziller, Robert Nelson and Harlan Herrick. The input-output language was designed by Roy Nutt. In November of 1954 the first specifications report was produced. In this report the following characteristics of the language were described:

" • variables of one or two characters in length,
- function names of three or more characters ,
- recursively defined "expressions,"
- subscribed variables with up to three subscripts,
- "arithmetic formulas" (which turn out to be assignment statements), and
- "DO formulas"

" [Backus 1998, p. 71]

Most of these characteristics were in the final version of FORTRAN with the main exception of the "DO formulas", which were simplified in the final version. The main important part of FORTRAN was not the design of the language since that was ignored and considered a minor task, but the design of the compiler. The FORTRAN compiler was completed in the spring of 1957 and it was the first compiler successful.

The compiler produced a program in six stages:

1. Read the entire source program, compile what it could and file the rest in tables

2. Analyze the DO statements and the references to subscripted variables
3. Transform the program into a form suitable for section 4 and 5
4. Analyze the flow of a program, divide it in basic blocks, do a statistical analysis of the expected frequency of execution of basic blocks and collect information about the index register usage.
5. Register allocation
6. Assemble the final program into binary code

This work was pioneering since at that time hardly any compilers existed.
In the spring of 1958 the second version of FORTRAN named FORTRAN II was completed. FORTRAN II had a better diagnostics system and subroutine definition capabilities. Soon after that in the winter of 1958-1959 a FORTRAN III came out with a support for Boolean expressions, function and subroutine names could be passed as arguments and facilities for handling alphanumerical data. Other version of FORTRAN followed and FORTRAN became very popular among the programmers. Backus made another very important contribution to the Computing Science. He developed notation system that after some further research became very common in describing the syntax. This notation system and the semantics of the programming language will be described in the following chapter.

Chapter 7
The semantics of the programming languages

To completely define a programming language the following three parts need to be described, its syntax, its semantics and its pragmatics. The syntax is a set of rules, which defines the appearance and structure of the sentences. The semantics describes the meaning of these sentences. The *pragmatics* describes the possible applications of the language. This paper will only concentrate on the semantics. It is worth though mentioning, that for the syntax the most common notation is the BNF (Backus Naur form) and the EBNF (Extended Backus Naur Form). BNF was originally described in paper of John Backus for ALGOL 58 in 1959 [Backus 1959]. Peter Naur presented a later modification of that system in 1960 in order to describe ALGOL 60 [Naur 1960]. This modified version of the syntactic system Backus presented in 1959 became known as the Backus Naur Form[30].

The formal description of the semantics can be done in three different ways, each of them having its own characteristics and uses. The three different strands are the operational semantics, the denotational semantics and the axiomatic semantics. All three semantic approaches are metalanguages describing the meaning of the sentences of a programming language.

The operational semantics are based on the idea of defining an abstract machine. An abstract machine is the equivalent of an idealized computer. This idealized computer can be close to the real computer or it can be a very abstract form of it. The abstract machine is defined in order to map the real programs in to programs of the abstract machine [Dershem and Jipping 1995, p. 25]. The program is then described in terms of the computational steps on this abstract machine. The meaning of a sentence is defined by the change of the machine state that occurs in each computational step. This concept resembles the concept of a Turing Machine. It is most likely that the development of operational semantics has been influenced by the works of Turing and can be considered as the evolvement of Turing's works in a metalanguage for programming languages.

The abstract machine can be very low-level close to the real machine or it can be high-level with an easy translation from the programming language. The abstract machine and the translation have to be defined. Operational semantics can be very useful to *language implementers*, because the have the ability to describe precise formulations of implementation techniques. The problem of correctness arises though with operational semantics, since their linguistic features can be obscured by their representational detail [Tennent 1994, p.171].

Here is a semantic analysis of the iteration commands in FORTRAN and Plankalkül.

[30] More information about the BNF and the EBNF can be found in [Fischer and Grodzinsky 1993, pp. 76-79], in [Ghezzi and Jazayeri 1998, pp. 35-36] and in [Sebesta 1996, pp.107-109]

Operational semantics for FORTRAN

```
                                        DO
do_var := initial                       K = 1,
loop [loop body]
do_var := do_var + stepsize
if do_var ≤ terminal goto loop          10
```

[Sebesta 1996, p. 297-298]

On the left side the operational semantics of the FORTRAN *do* command are shown. The *do* command takes on variable named K, then sets it equal with 1 and after the comma the number of iteration is set to 10. This is analysed using operational semantics in the following way the do_var, which stands for the variable the *do* command takes, is given its initial value. On the next line a command in the metalanguage is given, which is called loop. This loop command takes one variable, in this case the loop body. The loop body variable represents the commands, which are going to be repeated in the loop. On the next line the do_var gets updated by adding the stepsize to it. The stepsize is the amount that the loop counter, increases its time the iteration is repeated. This part of updating the loop counter (in this case K) is built in to the language and is done automatically. The final line checks if the do_var is less or equal to the number of iterations. If this condition is true then the program returns to the loop function and repeats its commands.

Operational semantics for Plankalkül

$W[F \longrightarrow P]$

```
F:=TRUE
loop[P]
eval F
if F = TRUE goto loop
```

In the case of Plankalkül the *W* command is defining a loop, or a Wiederholungsplan (W-Plan) as Zuse defines it [Zuse 1972, p. 25-30]. The *F* proposition is the condition of the loop, which must be true so the *P* function can be executed again. In the operational semantics side the first line defines the proposition *F* to be TRUE, just before the iteration begins. On the next line the loop function, which was used in the FORTRAN example, executes the command *P*. *P* can be more than one command, it simply groups all the commands inside the loop. A new function is defined in the next line. The new function, called *eval*, evaluates the *F* proposition. On the final line if the proposition *F* is TRUE the programs returns to the loop function and repeats the command *P*.

This semantic analysis of the iteration command *do* (in FORTRAN) and the *W* in Plankalkül shows how general the iteration in Plankalkül is. In FORTRAN the programmer is restrained in using a loop that can only be executed a specific amount. The ability to define a terminating condition for the loop, like a user's input, is not available in FORTRAN. Plankalkül offers this ability and the programmer has to define the terminating condition, unlike the case of FORTRAN, where the terminal condition is built into the language by using a *counter loop*. The feature, of defining the condition of the

loop, that Plankalkül provides to the programmer is now standard in almost every programming language.

The denotational semantics employ a different mapping. The mapping is done from the program to its meaning. Dana Scott and Christopher Strachey originally developed the denotational semantics and presented it [Scott and Strachey 1971]. Denotational semantics are based on complete partial orders, continuous functions, least fixed points and the lambda calculus of Church. A valuation function is used to map the program directly in to its meaning [Schmidt 1986, p. 3]. This means that the program will be mapped in object of a domain of mathematical entities like truth values, numbers or functions, which denote the meaning of it. It is characteristic of the denotational semantics that valuations are defined in a compositional manner; that means that the meaning of every phrase is mapped in to a function of its sub-phrases. The denotational definition is generally more abstract than the operational one. It offers a high-level and modular structure that can be of great importance to the *language designer* and *user.*

An operational semantic description, using the lambda-calculus notation[31], for the assignment in FORTRAN and in Plankalkül is explained here.

Denotational semantics	for	FORTRAN
λk.(k = 0)		K = 0

The assignment in FORTRAN follows the mathematical notation and the variable *K* is assigned the value 0. In the semantics side a function λk.(k = 0) is declared, which is assigning the number 0 to the variable *k.*

Denotational semantics	for	Plankalkül	
			Q
λv.(v = 2)		V	2
λk.(k = 4)		K	4
λs.(s = 1)		S	1.o

The assignment in Plankalkül is more complicated than the one in FORTRAN. The declaration of variable in Plankalkül requires the size and the index of the variable to be defined in the V and S lines respectively[32]. This variable Q2 is of size 1.o, which means that it is a single bit. The assignment is done in the K line. In the semantics side three functions are declared in order to analyse this assignment. The first function λv.(v = 2) assigns the index (2) to the v variable, the second function λk.(k = 4) assigns the value (4) in to the k variable and the third function assigns the size (1) in to the s variable.

The assignment in Plankalkül is more complicated, because it requires the user to define the exact details of the variable and make the assignment at the same time. In the case of FORTRAN the declaration of the variable is done automatically, when the variable is assigned a value. The assignment operation is more efficient in the case of FORTRAN and it can be characterized as high-level, while the assignment in Plankalkül is more low-level, since the exact details of the variable have to be defined during the assignment.

---

[31] For the lambda-calculus notation return to chapter 4.
[32] For declaration of variables in Plankalkül return to chapter 5.

The axiomatic semantics are using system of axioms as the name suggests. The axiomatic semantics was initially developed by R.W. Floyd and C.A.R. Hoare (1969)[33]. The meaning of the sentences of a program is not described; instead the properties of the language are defined. These properties are expressed with the help of axioms and the deduction rules of logic. Axiomatic semantics are concerned with proving the correctness of a program or some part of it. The *predicates* immediately before a program statement, which describe the constraints on the program variables at that particular point of the program, are called the *preconditions*. The predicates immediately after are called the *postconditions*. In the development of an axiomatic description for a program it is required that every statement has a precondition and a postcondition. A simple system of axioms would allow the proof of the equality of two programs. A more complex system would allow proofs about some properties of a program. It is of great importance that the axiomatic system that is used is *sound*. If the system is not sound it can lead to invalid proofs of correctness.

The notation used in this analysis of the iteration is the following {P}S{Q}, where P is the precondition, S is the statement form and Q is the postcondition. The definition of a general loop in axiomatic semantics is the following {P}while B do S end{Q}, where B is the statement that has to be true for the loop to be executed.

Axiomatic semantics	for	FORTRAN
{P}={K=1}		DO 20 K=1, 10
while $K \leq 10$		
S = …		20 …
{Q}={K=11}		

On the first line the precondition P of this loop is stated and this is K = 1. The line that follows states the condition for the loop, which is $K \leq 10$. The next line is the line, where the statements S, line 20 of the FORTRAN side get executed. When the terminal condition is reached the loop exits and the postcondition Q is K = 11

Axiomatic semantics	for	Plankalkül
		$W[F \longrightarrow P]$

{P} = *F* is TRUE
while *F* is TRUE
S = *P*
{Q} *F* is FALSE

On the first line of the semantic analysis the precondition P is stated and it is equal with *F* is TRUE. This means that the proposition *F*, which is the condition for the loop, stated in the next line, is TRUE when the loop begins. The following statement describes the statements *P* that are repeated in the loop. In the last line the poscondition Q of the loop is equal with *F* is FALSE, which is the way the loop has ended.

This semantic analysis shows the different way, the operational and the axiomatic semantics are describing the iteration in both languages. Again it is shown that the loop in Plankalkül is more general than the one in FORTRAN.

[33] [Hoare 1969].

It is essential to understand that the different semantics are close to each other and one style of semantics can be used to analyze another one. This relationship between the different semantics is described in [Winskel 1993, p. xv]. Winskel suggests that research in the last few years is promising a unification of semantics.

Figure 7.1 Kurt Gödel

The three strands of semantics discussed in this chapter have been influenced by the work in logic in the early 1900's. In addition the work done in the 1930's was significant to the development of the semantics. The operational semantics is probably the application of Turing machines in to a metalanguage for programming languages. The development of the denotational semantics has been influenced by the lambda-calculus of Church, while the axiomatic semantics seem to have been influenced generally by first-order Logic. It was after Gödel incompleteness theorem and his corollary about the richness of the metatheories[34] that logicians have understood the important difference between language and metalanguage and advanced further the mathematical models of the 1930's. Particularly those logicians concerned with theoretical computer science based their research on the already developed area of recursive functions; thus they were able to construct not only programming languages, but metalinguistic systems that describe these languages in coherent way.

[34] For historical information about Gödel's corollary look in chapter 9 in [Grattan-Guinness 2000, pp. 506-555].

Chapter 8
A final comparison between Plankalkül and FORTRAN

The evolvement of Computing Science in to a science of its own happened mainly in the 20th century. The developments of logic and in particular in recursive function theory were applied in the development of programming languages. The Turing machine became one of the most important abstractions of a computer. The von Neumann model of computer dominated the design of computers and some programming languages. Out of all these developments this project concentrates on the involvement of Mathematics in Computing Science. The influence of Mathematics is apparent in the early days of computing even by the simple fact that most of the early programming languages used a mathematical notation and their scope was to translate mathematical formulas. Two programming languages were chosen to be described and compared in this project.

Plankalkül and FORTRAN are considered to be the first two high-level programming languages. Plankalkül (1942-1945) was developed prior to FORTRAN, but unfortunately a compiler for it was not build until 2000, by the team of Professor Raul Rojas as mentioned in a previous chapter. FORTRAN was originally developed between 1953-1957. The development of the actual FORTRAN language was considered a minor task and the team of John Backus concentrated on the design of the compiler. The two languages have been compared in this project showing the advantages of Plankalkül over FORTRAN. Even Backus states about Plankalkül "Like most of the world …, we were entirely unaware of the work of Konrad Zuse" [Backus 1998, p. 69]. Backus continues describing Plankalkül as "… a more elegant and advanced programming language than those appeared 10 and 15 years later".

Plankalkül was developed in an environment of isolation. It was towards the end of the Second World War and Germany did not have any scientific contacts with the U.S.A. due to the war. It could be considered as one of the reason why Plankalkül did not become known in the computing community. Plankalkül was designed to be a universal algorithmic language that would be the theoretical basis for every computer. Konrad Zuse continued his efforts on the design of his Calculus of Programs and explained how this could be applied to mechanical and electrical relays. Plankalkül contained many of the features of predicate logic, heavily influenced by Hilbert and Bernays 1939 book on logic. Zuse, being a practical man, studied this book and took what he needed to develop his programming language. The λ-operator has been borrowed from Hilbert and Bernays, who in their turn borrowed it from the lambda calculus [Hilbert and Bernays 1939, vol.2, p. 462]. Zuse though decided to leave recursive functions out his programming language. The quote in the beginning of chapter 5 is characteristic of Zuse's beliefs about the connections of Mathematics and computer science. His efforts were to use Mathematics in computing in a manner that an engineer can design it in the hardware. Zuse has studied engineering and as it is natural the influences of his studies can be seen through out his whole life's work. Especially in the theoretical work Zuse has done, he always described the design of the features of his programming language in the hardware. Plankalkül's aim was to build a theoretical foundation, based on Mathematics and particularly on the logic of Hilbert and Bernays, that could be applied to the hardware.

FORTRAN was developed in the U.S.A. about ten years after the Second World War and it was not influenced in any major way by it. In those years, the need for a programming language was great, since programming was becoming complicated and time-consuming. FORTRAN had to overcome the general feeling about the 'automatic programming'[35], which promised many new features and delivered almost nothing. The design of FORTRAN was concentrated on the design of the compiler, while the development of the language was considered a minor task. The main challenge of the FORTRAN project was a compiler that would produce efficient programs, close to the ones human programmers could. A failure of this would have constituted a delay in the acceptance and further development of programming languages. The language was designed to translate mainly mathematical formulas. It had variables of one or two characters in length, due to the small amounts of memory of those days' computers and it could perform iterations using the *DO* command. The language was rather simplistic, but it was one of the first compilers, which could produce efficient programs.

This project is an effort to describe the complete scientific surroundings, the different scopes of the programming languages and the features of the languages. Most of the important and influential ideas in Computing Science, Mathematics and Logic were presented to the reader in order to appreciate the complete history of the two first high-level programming languages. The project's aim was to discuss not only historical facts or characteristics of the languages, but to describe the main ideas in Mathematics and logic and the way these ideas influenced the development of the Theoretical Computing Science.

[35] The historical context and the 'automatic programming' prior to the development of FORTRAN is described in chapter 6.

Explanation of fonts

The font type Times New Roman was used for normal text.
The font type *Monotype Corsiva* was used for all the technical terms, whose explanation can be found in the Glossary of technical terms.

Glossary of technical terms

Abstraction (Chapter 3): "An abstraction is a representation of an object that hides what could be considered as irrelevant details of that object, thus making use of the object easier."[Dershem and Jipping 1995,p. 134]

Assembler (Chapter 2.1): A program that translates Assembly languages into machine code

Assembly languages (Chapter 2.1): "A computer language that represents machine code programs in a form people can read. Each machine code instruction is represented by a short mnemonic code. Memory registers and storage addresses may be referred to by symbolic names rather than numeric codes, and labels and comments can be used to improve legibility. Assembly language programs have to be translated into machine code by a special program called an assembler before they can be run on the computer. Because an assembly language represents machine code instructions directly, it is specific to a particular type of central processing unit." [Oxford 1997]

charecteristica universalis (Chapter 5): A mathematical language that would be strong enough to describe all mathematics.

Compiler (Chapter 3): a computer program that translates high-level computer languages, such as FORTRAN and C, into machine code that can be executed directly on the computer.

Counter loop (Chapter 7): A loop, whose terminating condition depends on a counter.

CPU (Chapter 1): Central processing unit, it is the part of the computer where all the data is processed.

Effectively computable (Chapter 4): A procedure is said to be effectively computable, if for every input of a certain class of symbolic inputs there is a symbolic output. This procedure has to be deterministic, meaning that it will be unambiguous and complete and it will return a certain result after finite number of steps of a computer (human or machine).

free variable – bound variable (Chapter 4): "We will say that a variable is bound by a quantifier (linked to it) when it is in the scope of a quantifier and it alphabetically matches the variable in the quantifier… When a variable is not bound we will say to be free"[Guttenplan 1997, p. 183]. Consider the following example:

$$(\forall x)(Rx \rightarrow Gy)$$

then the variable x appearing in both the quantifier $\forall x$ and the predicate Rx is bound in both cases. The variable y is free, because even though it appears in the scope of the quantifier it does not match alphabetically the variable of the quantifier (in this case the quantifier is $\forall$ and the variable of that quantifier is x) within whose scope it lies.

Global (Chapter 2.2): A global variable is valid in the whole program.

High-level programming (Chapter 3): Programming, which uses languages that have an interface close to a natural language. Examples of high-level programming languages are FORTRAN (1954-1957), C (1972) and Java (1995).

λ-definable (Chapter 4): A function is said to be λ-definable, if it can be defined completely within the lambda calculus.

Language designer (Chapter 7): The person who will design the language. That person will outline the specification of the language, its syntax and pragmatics.

Language implementer (Chapter 7): The person who will implement the language. That person will design the compiler.

Language user (Chapter 7): The programmer who will use the language to develop applications.

Law of the excluded middle (Chapter 4): "The law of the excluded middle states every proposition is either true or false – that there is no middle position…"[Aspray 1980, p. 121]

Level of proceduralism (Chapter 3): The level of proceduralism is the context, when we are distinguishing between procedural and non-procedural.

Local (Chapter 2.2): A local variable is valid only in a specific part of the program.

Low-level programming (Chapter 3): Procedural programming usually done with Assembly languages, which very close to the machine language. Machine instructions like memory exchange are used.

Metalogic (Chapter 3): A theory, which discusses logic.

Methods (Chapter 2.1): Functions in object-oriented programming languages that are associated with a specified object.

Modularity (Chapter 5): The abstraction of the hardware trough several layers of software, which make the hardware transparent.

Object-oriented languages (Chapter 2.1): Programming languages that provide the ability to the programmer to create data objects, which will encapsulate the data and will have associated functions, which are called *methods*.
Postcondition (Chapter 7): The predicates immediately after a program statement, which describes the constraints on the program variables at that particular point of the program, are called postconditions.

Pragmatics (Chapter 7): It describes the possible applications of the language.

Precondition (Chapter 7): The predicates immediately before a program statement, which describe the constraints on the program variables at that particular point of the program, are called preconditions.

Predicate (Chapter 2.1): "The task of characterizing items in a Situation, saying what features they have, is carried out by predicates." [Guttenplan 1997, p. 160]

Procedural programming languages (Chapter 2.1): Programming languages in which the code is written defining how something is achieved instead of what it should be achieved

Proceduralism (Chapter 3): The distinction between how and what always in a context. The content is the level of proceduralism.

Referential transparency (Chapter 2.1): Referential transparency is the ability to call a function without having any side effects. This means that the internal structure of the function cannot be changed upon the function call. Referential transparency also means that a function will produce the same result, if the data given to the function is the same, no matter where it is in a program.

Sound (Chapter 7): "An argument that is valid and has true premises is called sound." [Guttenplan 1997, p. 23]

Strongly typed programming languages (Chapter 2.1): "A language is said to be strongly typed if all *type checking* that is feasible to do at compile time is done then and all other type checking is done at run time." [Dershem and Jipping 1995, p. 58]

Type checking (Chapter 2.1): "Type checking is the process of determining the type of a specified data object." [Dershem and Jipping 1995, p. 58]

Universal Turing Machine (Chapter 4): A Turing machine that is able to compute any other Turing machine.

Von Neumann type of computers (Chapter 1): A computer, which mainly constitutes out of a CPU, a storing device and a connection tube. More information on the von Neumann model of computers can be found in [Ceruzzi 1997, pp. 6-7].

Von Neumann type of programming language (Chapter 4): A programming language, which incorporates the features of the von Neumann type of computer. The language works in the one word-at-time architecture, which is one of the ideas of the von Neumann computer. More information about von Neumann languages can be found in [Backus 1978, pp. 615-616]

Bibliography

- Abramsky, S. , Gabbay, D.M. and Maibaum, T.S.E. , (1992) (eds.), *Handbook of Logic in Computer Science. Volume 2. Backround: Computational Structures*, Clarendon Press

- Abramsky, S. , Gabbay, D.M. and Maibaum, T.S.E. , (1994) (eds.), *Handbook of Logic in Computer Science. Volume 3. Semantic Structures*, Clarendon Press

- Aspray, W.F. (1980), *From Mathematical Constructivity to Computer Science: Alan Turing, John von Neumann and the Origins of Computer Science in Mathematical Logic*, University Microfilms (1982)

- Backus, J. (1954), 'The IBM 701 Speedcoding System', *Journal of the ACM*, vol. 1, no. 1, pp. 4-6

- Backus, J. (1959), 'The Syntax and Semantics of the Proposed International Algebraic Language of Zurich ACM-GAMM Conference', *Proceedings International Conference on Information Processing*, UNESCO, pp. 125-132

- Backus, J. (1978), 'Can programming be liberated from the von Neumann style?', *Communications of the ACM*, vol. 21, no. 8, pp. 613-641

- Backus, J. (1998), 'The History of FORTRAN I, II and III', *Annals of the History of Computing*, vol. 20, no. 4, pp. 68-78

- Barendregt, H.P. (1992), 'Lambda Calculi with Types', in [Abramsky et al. 1992], pp. 117-309

- Barendregt, H.P.(1980), *The Lambda calculus. Its Syntax and Semantics*, North-Holland Publishing Company

- Bauer, F.L. and Wössner, H. (1972), 'The Plankalkül of Konrad Zuse: A Forerunner of Today's Programming Languages', *Communications of the ACM*, vol. 15, no. 7, pp. 678-685

- Beeson, M.J. (1988), "Computerizing Mathematics: Logic and Computation", in Herken (1988), pp.191-225

- Breton, P. (1991), *Histoire de l'Informatique*, Editions La Decouverte, in Gouskos.D. and Pefanis.G.(trans.), *Ιστορία της Πληροφορικής*, Diavlos

- Ceruzzi, P.E. (1997), 'Crossing the Divide: Architectural Issues and the Emergence of the Stored Program Computer, 1935-1955', *Annals of the History of Computing*, vol. 19, no. 1, pp. 5-19

- Ceruzzi, P.E. (1998), *A History of Modern Computing*, MIT Press

- Church, A. (1932), 'A Set of Postulates for the Foundation of Logic', *Annals of Mathematics, Series 2*, vol.33, pp. 346-366

- Church, A. (1933), 'A Set of Postulates for the Foundation of Logic (2nd paper)', *Annals of Mathematics, Series 2*, vol.34, pp. 839-864

- Church, A. (1941), *The Calculi of Lambda-Conversion*, Princeton University Press

- Cohen, I. (2000), *Howard Aiken: Portrait of a Computer Pioneer*, MIT Press

- Davis, M. (1987), 'Mathematical Logic and The Origin of Modern Computers', in Phillips, E.R. (ed.), *Studies in the History of Mathematics*, The Mathematical Association of America, pp. 137-165

- Davis, M. (1988), 'Trends in Logic: Relations with Computer Science', in Ferro, R., Bonotto, C., Valentini, S., and Zanardo, A. (eds.), *Logic Colloquium '88*, Elsevier Science Publishers B.V., 1989

- Davis, M., Sigal, R. and Weyuker E.J. (1994), *Computability, Complexity, and Language. Fundamentals of Theoretical Computer Science*, Academic Press

- Dershem, H.L. and Jipping, M.J. (1995), *Programming Languages: Structures and Models*, PWS Publishing Co.

- Fisher, A.E. and Grodzinsky F.S. (1993), *The Anatomy of Programming Languages*, Prentice-Hall International

- Gandy, R. (1988), "The Confluence of Ideas in 1936", in [Herken 1988], pp. 55-111

- Ghezzi, C. and Jazayeri, M. (1998), *Programming Language Concepts*, John Wiley and Sons

- Gillies, D. (1996), *Artificial Intelligence and Scientific Method*, Oxford University Press

- Giloi, W.K. (1997), 'Konrad Zuse's Plankalkül: The First High-Level "non von Neumann" Programming Language', *Annals of the History of Computing*, vol. 19, no. 2, pp. 17-24

- Gödel, K. (1931), "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I", in [van Heijenoort 1981], pp. 596-616

- Grattan – Guinness, I. (2000), *The Search for Mathematical Roots 1870-1940*, Princeton University Press

- Guttenplan, S. (1997), *The Languages of Logic*, Blackwell Publications

- Herken, R. (1988) (ed.), *The Universal Turing Machine. A Half-Century Survey. A Half-Century Survey*, Oxford Science Publications

- Hilbert, D. and Bernays, P. (1939), *Grundlagen Der Mathematik*, 2 vols ,Verlag Von Julius Springer

- Hilton, P. (1985), 'Obituary: M.H.A. Newman', *Bull.London Math. Soc* (1986), pp. 68-72

- Hindley, R. and Seldin, J. (1988), *Introduction to Combinators and λ – Calculus*, Cambridge University Press

- Hoare, C.A.R. (1969), 'An Axiomatic Basis for Computer Programming', *Communications of the ACM*, vol. 12, no. 10, pp. 576-583

- Hodges, A. (1988), "Alan Turing and the Turing Machine", in Herken (1988), pp. 3-15

- Kleene, S.C. and Rosser, J.B. (1935), 'The inconsistency of certain formal logics', *Annals of Mathematics, Series 2*, vol.36, pp. 630-636

- Kleene, S.C. (1936), 'General Recursive Functions of Natural Numbers', *Mathematische Annalen*, vol.112, pp. 727-742

- Knuth, D.E. and Pardo, L.T. (1977), 'The Early Developments of Programming Languages', in [Metropolis et al. 1980], pp. 197-273

- Laning, J.H.Jr. and Zierler N. (1954), 'A Program for Translation of Mathematical Equations for Whirlwind I', *Engineering Memorandum E-364*, MIT Instrumentation Lab

- MacLennan, B.J. (1983), *Principles of Programming Languages: Design, Evaluation, and Implementation*, CBS College Publishing

- Mahoney, M.S. (2000), 'Software as Science – Science as Software', *Proceedings of the International Conference on History of Computing 2000 – Mapping the History of Computing: Software Issues*, Heinz Nisdorf MuseumsForum, Paderborn, Germany

- Martin, J. (1985), *Fourth - Generation languages*, Volume 1 principles, Prentice Hall

- Metropolis, N. , Howlett, J. and Rota, G. (1980) (eds.), *A History of Computing in the Twentieth Century*, Academic Press

- Nagel, E. and Newman, J.R. (1959), *Gödel's Proof*, Routledge & Kegan Paul Ltd.

- Naur, P. (1960) (ed.), 'Report on the Algorithmic Language ALGOL 60', *Communications of the ACM*, vol. 3, no. 5, pp. 299-314

- Newman, M.H.A. (1948), 'General Principles of the Design of All-Purpose Computing Machines', *Proceedings of the Royal Society of London*, Ser. A., vol. 195 (1948), pp. 271-274

- Oxford (1997), *Oxford Interactive Encyclopedia*, The Learning Company

- Randell, B. (1973) (ed.), *The Origins of Digital Computers. Selected Papers*, Springer Verlag

- Schmidt, D.A. (1986), *Denotational Semantics. A Methodology for Language Development*, Allyn and Bacon

- Scott, D.S. and Strachey, C. (1971), 'Towards a Mathematical Semantics for Computer Languages', *Symposium on Computers and Automata*, Polytechnic Press, pp. 19-46

- Sebesta, R., (1996), *Concepts of programming languages*, Addison-Wesley

- Shields, M.W. (1987), *An Introduction to Automata Theory*, Blackwell Scientific Publications

- Tennent, R.D. (1994), "Denotational Semantics", in [Abramsky et al. 1994], pp. 169-311

- Turing, A.M. (1936), 'On Computable Numbers, with an Application to Entscheidungsproblem', *Proceedings of the London Mathematical Society*, series 2, volume 42, pp. 230-265

- van Heijenoort, J. (1981), *From Frege to Gödel. A Source Book in Mathematical Logic, 1879-1931*, Harvard University Press

- Wilkes, M.V., Wheeler, D.J. and Gill, S. (1957) (2nd ed.), *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley

- Winskel, G. (1993), *The Formal Semantics of Programming Languages. An Introduction*, MIT Press

- Zuse, K. (1962), 'The Outline of a Computer Development from Mechanics to Electronics', in [Randell 1973], pp. 171-186

- Zuse, K. (1970), *Der Computer – Mein Lebenswerk*, Springer Verlag

• Zuse, K. (1972), 'Der Plankalkul',*Berichte der Gesellschaft fur Mathematik und Datenverarbeitung*, vol.63, Bonn

• Zuse, K. (1976), *Some Remarks on the History of Computing in Germany. Lecture at the International Research Conference on the History of Computing*, Konrad-Zuse-Zentrum für Informationstechnik Berlin (www.zib.de)