UDC 004.432
Review Paper

1957-2007: 50 Years of Higher Order Programming Languages

Alen Lovrenčić *alen.lovrencic@foi.hr*
University of Zagreb
Faculty of Organization and Informatics

Mario Konecki *mario.konecki@foi.hr*
University of Zagreb
Faculty of Organization and Informatics

Tihomir Orehovački *tihomir.orehovacki@foi.hr*
University of Zagreb
Faculty of Organization and Informatics

Abstract

Fifty years ago one of the greatest breakthroughs in computer programming and in the history of computers happened – the appearance of FORTRAN, the first higher-order programming language. From that time until now hundreds of programming languages were invented, different programming paradigms were defined, all with the main goal to make computer programming easier and closer to as many people as possible. Many battles were fought among scientists as well as among developers around concepts of programming, programming languages and paradigms. It can be said that programming paradigms and programming languages were very often a trigger for many changes and improvements in computer science as well as in computer industry. Definitely, computer programming is one of the cornerstones of computer science.

Today there are many tools that give a help in the process of programming, but there is still a programming tasks that can be solved only manually. Therefore, programming is still one of the most creative parts of interaction with computers.

Programmers should chose programming language in accordance to task they have to solve, but very often, they chose it in accordance to their personal preferences, their beliefs and many other subjective reasons.

Nevertheless, the market of programming languages can be merciless to languages as history was merciless to some people, even whole nations. Programming languages and developers get born, live and die leaving more or less tracks and successors, and not always the best survives.

The history of programming languages is closely connected to the history of computers and computer science itself. Every single thing from one of them has its reflexions onto the other. This paper gives a short overview of last fifty years of computer programming and computer programming languages, but also gives many ideas that influenced other aspects of computer science. Particularly, programming paradigms are described, their intentions and goals, as well as the most of the significant languages of all paradigms.

Keywords: Programming languages, Programming Paradigms, History

1. Introduction: The Pre-History of Programming Languages

1.1 The Birth of Computers

The word computer evolved in the last 100 years rapidly. Let us return to its basic meaning: *calculating device*. From the ancient time people had a need for devices that can help them in calculating. One of the first such devices was *abacus*. This simple device was invented in China more than 2.200 years ago, but is still in use in many parts of the world. At the same time, Greeks a had device for calculating calendars called *Antykhiteron (Αντικυθηρων)* [16]. These were the first known calculating devices in the world.

Although, they were not real computers, for they did not really make computations. They only helped in the manual computation. The first real mechanical calculator is dated in 1623. It was developed by Wilhelm Schickard (1592-1635). The device was called *calculating clock*. This device was used by Johannes Kepler (1571-1630) in his astronomical calculations. Twenty years after that, in 1645, Blaise Pascal (1623-1662) created *Pascaline*, the first calculator that was cheap enough to be serially-produced.

These calculators made calculation much easier, but they had one disadvantage: they could not store results for future calculations: a memory. The first memory medium was, not surprisingly, paper. In the first half of the 18 century Basile Bouchon and Jean-Baptiste Falcon wanted to solve the problem that was not connected to the computation. He wanted to make an automatic loom that would be able to reproduce a pattern. In the year 1725 he had to develop the way of saving and reading a saved loom pattern, which led to the development of the first external memory: a punched paper loop [91]. His invention and its versions (such as punched card or tape) had a many different applications, from looms, mechanical pianos to calculators.

The birth of the first real computers, machines that were able to make more complex calculations automatically, is dated in the late 19^{th} century when the first automated calculating machines were developed. These machines were nothing like todays computers. They were mechanical and they were more similar to calculating machines that were used in accounting departments of the firms before computers overtake the whole process of accounting.

The first such machine was developed by Charles Babbage (1791-1871) and called *Difference Engine* [84]. This was a mechanical machine that was able to calculate values of polynomial functions using finite difference method. So, this machine was able to add and substract. He has to employ higher mathematics to avoid multiplication and division, which were hard to implement in the mechanical machine.

After completing Difference Engine, Babbage started to make plans for another, more complex machine that he called *Analicital Engine* [71]. The machine, as it was ment to be, had many of the properties of modern computers. Firstly, it should had been programmable by the using of punched cards. The machine should have had some main constructs of the modern computer programming languages, such as branches, loops, etc. Analytical engine was, unfortunately, never built due to his death in 1871. However, the concept of the machine was not forgotten. Augusta Ada King, countess of Lovelace (1815-1852), the daughter of lord Bayron, developed a program for analytical engine that was able to calculate a sequence ot Bernoulli's numbers. That was the first known computer program made ever, and she was the first computer programmer in the world.

In the beginning of the 20^{th} century the first *analog computers* came to the stage. Their development was in the big influence of the mechanical calculating machines developed before them. The main concept of the analog computers lays in the analogy between mechanical components, such as springs and dashpots with electronic circuits like capacitors, reductors and resistors. Using this analogy, electrical engineers developed first analog computers from the schemes for mechanical calculating machines. These machines were serving as a control devices for the mechanical systems, they were specialized for a single task and were not flexible at all. Many of them had

military use, especially at naval and air vehicles. United States of America used them at the battle ships in the Second World War, and some of them were in the function until the Vietnam war. One of them was *Norden Bombsight* which was installed on the bomber aircrafts, and was calculating a bomb trajectory. The similar device for the battle ship guns was called *Fire Control System*. One of the first analog computer whose purpose was calculation was developed in Soviet Union 1936. It was called *Water Integrator* and it calculated solutions of differential equations. This device was very interesting in one other way: it had very interesting output device. The output device were glass tubes filled with water. The level of the water in the tubes shows a result of the calculation.

These machines were in the first time superior to another concept of the computer - *digital computer*. They were able to solve much more complex problems, while digital computers were very limited. But rapid development of digital computers annulated this disadvantages and in the 1940's analog computers began to loose their position in every field. The first digital computer was *Atanasoff-Berry computer* [44]. It was build in 1936. In most papers on computer history it was not considered as first digital computer, because it was not Turing complete and because it was not programmable. But, if we live Atanasoff-Berry computer on the side, ENIAC, the machine that is widely considered as the first digital computer, still actually was not the first at all. The first Turing complete computer was Z3 [77] developed by the German scientist Konrad Zuse in 1941. This computer also had the first higher order programming language, which we will discus in detail later. It was also the first computer with the internal memory. Z3 was programmable by punched tape. Unfortunately, Zuse's work was lost in the Second World War and did not have influence on the further development of computers. That is the reason why authors often do not consider this computer as the first one. The second digital computer was developed by British cryptographers in 1943. and was called *Collosus*. However, like the Atanasoff-Berry Computer, Collosus was not Turing complete. The first computer that had the architecture known today as *von Neumann Architecture* was EDVAC, designed by John von Neumann (1903-1957). He also described the new architecture for digital computers that became the standard digital computer architecture. On von Neumanns results the well-known ENIAC was [34] developed by John Mauchly (1907-1980) and John Adam Presper Eckart Jr. (1919-1995) in 1943. It was Turing complete and it was programmable by rewiring.

1.2 The Iron Age: Digital Computers and Assemblers

The first complete von Neumann machine was built in Manchester in 1948 and was called *Manchester Small Scale Experimental Machine* or simple, *Baby*. It had 32-bit word with the binary language with only few instructions. The program could not exceed 32 words. Baby was followed by much stronger machine called EDSAC. It still had simple assembler, but had much more program memory and was able to work with floating-point aritmetic, arithmetical and trigonometrical operations, vectors and matrices. EDSAC had 17-bit word and two registers: accumulator and multiplier, each of them was able to hold two words. The first programs for EDSAC were made in May, 1949. and were calculating squares of numbers and list of prime numbers. In EDSAC computer some of the basic concepts of programming languages, as we know today, were implemented for the first time. For example, dual complement which avoids negative zero in the domain of the numbers was implemented. There were also some concepts that are abandoned today. For example, fractions were implemented in the fixed-point notation.

EDSAC was especially important for the history of programming languages because of the first assembler with 41 different instructions that was developed for it. For the first time in the history the idea of mnemonics, two or three-letter words that represent instructions was implemented. Although EDSAC could not compete with analog computers in speed, it was recognized as a major improvement in computer engineering, because of new concept of internal memory

that can contain both programs and results of calculations, and because of this new concept of programming that was much easier than programming of analog computers.

By the 1951. EDSAC became rather popular due to its commercialization called LEO 1. 97 routines were developed for it, including implementation of floating point arithmetics, implementation of complex numbers, , power series, logarithms of n-th root, trigonometric functions, vector operations, matix operations and so on.

It was the matter of time when those computers will be used not only for mathematical calculations, but for entertainment. The first computer game was developed right next year. In 1952. tic-tac-toe was developed for EDSAC. The game was rather simple, with very known draw strategy, but it was important as the beginning of one of the widest fields of the computer usage – computer games.

After LEO 1 was presented on the market, many companies were interested in the new technology. Many well known computers were developed based on the von Neumanns. In 1950. UNIVAC I (Universal Automatic Computer) was developed. In the same year the first computer in Soviet Union was developed, and it was called MESM. One of the largest companies that started with computer production was the company whose main product were typing machines: IBM. In 1952. they announced their first mainframe computer called IBM 701. This computer was the first of one of the most successful series of computers: IBM 700/7000 series. UNIVAC 1 computer and its successors, as well as IBM and its successors were very important for the development of programming languages, because the first higher order programming languages were developed for them.

2. The Antique: The First Generation of Higher Order Programming Languages

2.1 The Mayan History: Konrad Zuse

As is the case with the history of the mankind, the history of the programming languages also had branches of very advanced researchers whose work stayed unknown, and although it was amazing and respectable, it did not have any influence on the major development due to its isolation.

The best example of this is the work of Konrad Zuse, German scientist who was definitely in front of his time. As it was told before, during the Second World War he developed his Computer named Z3, which was non-von Neumann, but Turing complete computer. The second of the peaks of his research was Plankalkül [8], the first developed higher-order programming language in 1948. Due to war and post-war age, Zuse's work was widely unnoticed and its influence to the development of the early higher-order programming languages was very small.

Plankalkül was developed on the idea of relational algebra and APL. It had assignments, subroutines, arrays, branches and all concepts known from the modern programming languages. This language was never implemented in the computer although some of its concepts were implemented on Zuse's Z3 computer.

The main problem of this language looking from the todays point of view is that it had two-dimensional notation of instructions. Also, language was rather hard to read due to it's mathematical notation.

Despite all difficulties, the first Plankalkül compiler was developed in year 2000, five years after its author's death, fulfilling his wish that "after some time as a Sleeping Beauty, yet will come to life", unfortunately now only as a curiosity in the development of programming languages.

2.2 The Low Antique: The Age of FORTRAN, LISP and COBOL

2.2.1 FORTRAN

Right after IBM 701 computer was developed, its developers started to think about the languages that would make programming process. In 1954. the idea of the first higher-order language is developed, totally separated from Zuse's Plankalkül. The name of the language was FORTRAN (Formula Translator) and its main goal was to make programming of mathematical calculations easier. The original idea was developed by the IBM team whose manager was J. W. Backus. The team started its work in 1953. when they got an assignment to make alternative to assembler, which was very clumsy for programming larger calculations. In the first time, customers were very reluctant about using higher-order language because of the fear of getting slow programs incomparable with the fast assembler programs. That brought a very hard task of compiler optimization to Backus' team.

The time needed to make compiler whose executable is comparable with the hand written assembly code was a hard task indeed. They needed three years to solve it, and in April, 1957. the first FORTRAN compiler that had constant speed factor of 20 regarding the hand written assembly code was presented on IBM 704. This space-time trade-off between the time of developing a program and executing it was rather acceptable at that time, and FORTRAN began to live.

As the language was intended for numerical intensive programs, the numerical data types and functions were much more in the focus than data structures in FORTRAN. The first FORTRAN introduced two numerical data types for integers and floating point numbers. The only aggregation technique was array as an continuous sequence of variables of the same type. It had 32 key language words, including arithmetic three-way IF statement, DO loops, GOTO statement, READ and WRITE I/O statements, STOP, END, PAUSE, CONTINUE statements, and, of course, arithmetic expression assignment to variables. The data types available in the language were numerical: integer and floating point. There were no explicit variable type definitions, except for arrays. Variables get types in accordance to the first letter of the variable identifier. Variables whose name starts with letters I, J, K, L, M had integer data type, and all others had floating point data type.

Program 1

```
      DIMENSION IA(10)
      DO 10,I=1,10
10       READ (*,*) IA(I)
      MAX=IA(1)
      DO 20, I=2,10
         IF (A(I)-MAX) 30, 20, 20
30       MAX=A(I)
20    CONTINUE
      WRITE (*,*) MAX
      STOP
      END
```

The syntax of FORTRAN may seem rather odd today. However, one must have in mind that the theory of programming languages was not developed yet when FORTRAN arrived. The original FORTRAN syntax was highly non-regular. Through the years, syntax was changed widely by adding more and more statements in every new version, but also by changing some of the old statements that became problematic from theoretic or practical point of view.

The very next year the second version of FORTRAN complier arrived, with one of the most important improvements: it allowed subroutines. There were two sorts of subroutines introduced: SUBROUTINE and FUNCTION. This was a major improvement of the language that allowed a basic reusability of segments of the code, and also brought a important issue of writing more understandable code. FORTRAN had so called direct addressing of subroutines. This means that it does not use program stack for calling its subroutines. It assigns a fix memory address that contains the returning address for the every call to every subroutine, instead. This is the reason why FORTRAN is still one of the fastest languages today. This is also the reason why FORTRAN does not support recursions. Although all modern languages use program stack for subroutine calls, FORTRAN stayed with its original concept for long time, giving priority to the speed of the program over the recursion support.

Only four years after the first FORTRAN complier was introduced, FORTRAN IV was presented, bringing new LOGICAL data type. Related to that, logical IF, in the form that is usual in modern programming languages, was included.

Shortly after that, in 1964. ASA (today ANSI) supported FORTRAN IV as a standard. This was the first case of the standardization of some programming language. The standardization of FORTRAN brought new order, but also a new fuel to the development of the language. Two years after the first standardization of FORTRAN a new standard was presented by ASA, called FORTRAN 66. This version of FORTRAN was important because it added several new important properties to the language. The first one was data blocks, or, as we would call it today, records. It also introduced several new data types such as DOUBLE PRECISION and COMPLEX. The whole arithmetic for complex numbers was built into the language. The syntax of the IF statement was enhanced, so logical IF was allowed beside the original arithmetic IF statement. Because of that, the new comparison and logical operands were added. The operands did not have syntax as it is usual in mathematics and programming languages today. Their names were .LT., .LE., .EQ., .GT., .GE. and .NE. meaning ”less than”, ”less or equal to”, ”equal to”, ”greather than”, ”greater or equal to”, and ”not equal to”. The logical operators were .AND., .OR. and .NOT. and they had usual logical meaning. The modular programming paradigm was also introduced in this version of the language by adding external subroutines and subroutine libraries. Another major improvement of the language introduced in this version, bringing wide usability to FORTRAN was statements for file reading and writing.

After FORTRAN 66 standard there were several more steps in standardization of the language. FORTRAN 77 standard was published in 1977. with the standardization of function libraries and INCLUDE statement; FORTRAN 90 standard that finally introduced program stack and recursions; FORTRAN 95, that introduces first-order logic, with WHERE and FORALL clause; and finally FORTRAN 2003 that widely redefines FORTRAN syntax in accordance to C programming language with the major goal of interoperability of FORTRAN and C.

FORTRAN has its followers and opponents today, too. There are many discussions wether FORTRAN should be developed any further. Even petitions can be found on the Internet demanding retirement of the FORTRAN. But FORTRAN stays the first higher-order programming language, and it stays alive as a definitely the most long living programming language. For sure, it is not used widely anymore for common programming tasks, but still remains one of the fastest higher-order programming languages for numerical intense computer programs.

2.2.2 LISP

In that time the theoretical ideas regarding programming were much more developed than practical tools. It may surprise someone, but the foundations of artificial intelligence, particularly expert systems already existed. From 1956 at Darthmouth was organized Summer Research Project on artificial intelligence. Te main field of research at that project were systems based on the sequences

of decisions (expert systems). In 1958. John McCarthy and Marvin Minsky started the artificial intelligence project at M.I.T. and decided to create language that would be specialized for these systems. In the fall of 1958 they specified language called LISP (LIst Processing Language). LISP was developed on the solid theoretical foundations, particularly on Church's λ-calculus, and its syntax resembles FORTRAN syntax.

The idea of programming in LISP was very different from the imperative style FORTRAN had. As λ-calculus naturally leads to recursive functions, the main LISP programming mechanisms were recursions and lists, both implemented through stack ADT. Resembling the λ-calculus syntax of expressions, commands and functions in LISP, as well as expressions were in prefix notation that was more appropriate for stack implementation. So, we can say that LISP was the first programming language that was using programming stack for the function maintaining '- the idea that all modern programming languages use. The second idea, the idea of using dynamic data aggregations, namely lists instead of fixed-length arrays implemented in FORTRAN, was also used in many languages after, such as Prolog, LOGO and other AI-oriented languages.

LISP was never popular as FORTRAN, regardless its completeness and theoretical foundations, because it has rather weird syntax and concepts that required theoretical knowledge, but it gave a new, more theoretical oriented concept of programming language development that had the great influence to the development of many languages created after it, such as ALGOL, Pascal, Fortran, C, Prolog, LOGO and so on.

The way LISP treated the program was completely different from FORTRAN. Firstly, LISP is interpreter. That means that LISP program is loaded into LISP machine, and is treated as a database in memory. When the program is loaded, its becomes an addition to all LISP interactive commands, and it functions can be called from the command prompt. In fact, after the program is loaded into LISP environment, every function defined in the program becomes a command that can be called from the command prompt.

LISP is the language that incorporates functional programming paradigm with procedural paradigm in the elegant way. Namely, definitions of functions are procedural, but their treatment as objects is functional. The idea of functional approach that was introduced in LISP was fully developed in Prolog programming language that is fully functional in its approach and definitions of programming objects (predicates), and does not have any loops or any other procedural mechanism (except cut predicate).

The second idea, which is extremely different from FORTRAN and most of other procedural languages is the way that variables and data types are treated in LISP. Generally, data in LISP programming language can be divided into atoms and lists. LISP programming language is not a typed programming language. That means that variables in LISP does not have to be defined with one of the language data types. Instead of that, variables can contain any valid LISP data. In one moment it can be an integer, and just a line of the code after it can be a list of data. Even more, lists does not have to contain elements of the same type. They can contain elements that are of any LISP data type, including other lists. In fact, LISP programs are nothing else than lists of commands and their arguments. If you look deeper, you can say that any LISP program is a database of LISP functions that has the syntax of a λ-expression [69].

Program 2

```
(defun rd (lst x) (
       if (= x 1) (
              setq lst (cons (read) lst)
              )
              (
              setq lst (cons (read)
```

```
                              (rd lst (- x 1)))
             )
))



(defun find-max (lst) (
       if (null (cdr lst)) (
          car lst
          )
          (
          if (> (car lst)
                (find-max (cdr lst)))
             (car lst)
             (find-max (cdr lst))
          )
)
)

(defun strt ()
       (find-max (rd nil 10))
)
```

There are two fundamental notations of lists in LISP programming language. The first and the more common one is parenthesis notation of the list. For example, let us see how can we define a list from the LISP command prompt:

Program 3

```
:(setq lst (list 2 3 4))
  (2 3 4)

:(list 1 lst)
  (1 (2 3 4))

:(cons 1 lst)
  (1 2 3 4)
```

The second notation, which is allowed is so called dot notation. The same definition as above could be defined in following way:

Program 4

```
:(setq lst (list 2 . (3 . (4 . nil)))
  (2 3 4)

:(1 . lst)
  (1 2 3 4)

:(1 . (list lst))
  (1 (2 3 4))
```

The main problem with LISP programming language was not related to its theoretical foundations and incompleteness, as it was the case with FORTRAN, but it was related to its rather complex idea of interpreting LISP programs as λ-expressions and its functional orientation. These two facts made LISP too tough for regular mortals, and make it understandable only to mathematicians and computer theorists.

Actually, LISP is, if we look at its opponents, programming language whose value is the least questionable. There are no big opponents of LISP programming language, but there are so many people that do not understand how LISP really works. That is the reason why LIPS never became very popular among the programmers that stick to procedural programming. Nevertheless, LISP is still very live today. There are two very popular systems that still have LISP as their underlaying language: AutoCAD and EMACS. LISP, or some of languages that have LISP as their base (such as Scheme, ObjectLISP, LOGO and so on). LISP is still a favorite language of the part of AI scientists which opposed to logical programmming paradigm and Prolog programming language.

Since LISP syntax is defined on the rather high level of λ-expressions and not on the level of commands, soon after McCarthy and Minsky published their LISP definition and implementation, many of LISP dialects appeared. Because of that the great need for standardization appeared very quickly. In 1990 ANSI published their first standard of LISP programming language called Common LISP. Common LISP was a subset of may LISP dialects available at that time. In 1994 ANSI made a rather compact Common LISP foundations in their document X3.226-1994, "Information Technology Programing Language Lisp". After that the interest for LISP programming language became very low, and it looks like LISP will be forgotten as many other languages. However, 10 years after, when the new millennium began, LISP became interesting to many scientists again. For sure, AutoCAD and EMACS had a great influence in the survival of LISP, but after year 2000, some completely new dialects of LISP (such as Scheme) arrived and gave LISP a new meaning and fields of usage.

2.2.3 COBOL

Not long after FORTRAN was introduced, the need for the higher-order language of other kind was recognized: the need for a language that would be more business oriented. While IBM employed their resources to the improvement of FORTRAN language, UNIVAC and US Ministary of Defence started a project of development of business-oriented language. Exactly one year after Zürich ALGOL meeting at the meeting at Pentagon so called *Short Range Committee* was founded and its main goal was to develop business-oriented programming language. The members of committee were six computer manufacturers: Borroughs Corp, IBP, Minneapolis-Honeywell, RCA, Sperry Rand, and Sylvania Electric Products, as well as three US government agencies: US Air Force, David Taylor Model Basin, and National Bureau of Standards (now NIST). The committee had never had a single meeting, but sub-committee was founded with the goal of defining business-oriented progamming language. This subcommittee had six members: Gertrude Tierney and William Selden from IBM, Howard Bromberg and Howard Discount from RCA, and Vernon Reeves and Jeran V. Sammet from Sylvania Electric Products.

The goals of this language were significantly different from those of FORTRAN. While in FORTRAN the main goal was efficiency, in this language the main goals were readability, easy to learn syntax and capability of data aggregation.

There were several concepts that were used for COBOL definition. The first concept was developed by IBM expert R. Barner. Its name was COMTRAN. IBM intended to make language more appropriate to businessmen, but, obviously, its development was too slow. On the basis of IBM's COMTRAN, Grace Hopper, admiral at US. Navy, developed her language concept known as FLOW-MATIC, and it was direct predecessor of the first commercial business-oriented language named COBOL (Common Business Oriented Language). It was presented in 1959. by

CODASYL (Conference on Data System Languages). CODASYL was one of the most important professional body whose goal was the development of business-oriented languages until the beginning of 80's when F. Codd proposed his relational model and practically made procedural business-oriented languages obsolete.

The ideas that leaded to COBOL definition were to make programs essay-like and English grammar like. Therefore, programs were divided into three main parts called divisions. It was IDENTIFICATION DIVISION, DATA DIVISION and PROCEDURE DIVISION. The divisions had been divided further. The IDENTIFICATION DIVISION consisted of PROGRAM-ID and FILE-CONTROL. DATA DIVISION was divided into FILE SECTION and WORKING-STORAGE SECTION. At the end, PROCEDURE DIVISION had MAIN ROUTINE and other routines that MAIN ROUTINE called.

The main difference COBOL had regarding FORTRAN and other mathematical-oriented languages was its orientation to files at secondary memory storage for input and output data, while other languages were mainly oriented on data in main memory.

Program 5

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
     Max.
FILE-CONTROL.
     SELECT ARRAY-FILE
            ASSIGN TO DISK.
*
*
DATA DIVISION.
FD ARRAY-FILE.
01 ARRAY-RECORD.
   02 NO PIS 9(7)V99.
*
*
WORKING-STORAGE SECTION.
77 MAX 9(7)V99.
*
*
PROCEDURE DIVISION.
OPEN INPUT ARRAY-FILE.
READ ARRAY-FILE.
MOVE NO TO MAX.
PERFORM 9 TIMES
        READ ARRAY-FILE
        IF NUMBER>MAX THEN
           MOVE NO TO MAX
        END-IF
END-PERFORM.
DISPLAY MAX.
STOP RUN.
```

From program 5 it can be seen that COBOL is highly adjusted to file manipulation problems. For problems that have arithmetic operations, COBOL was completely inadequate, because of

its extremely poor arithmetic syntax. The second problem is that it has over 400 keywords and that makes it hard to learn. Finally, the third problem is that it is not adequate for small programs, because of its decomposition of programs into divisions and sections. If the algorithm for program solution is rather simple, the big overhead had to be created to define identification and data divisions.

Regarding its poor algorithmic properties, COBOL was widely adopted as a de facto standard for development of business applications. Before relational databases COBOL was the only language that was adequate for large business applications development. However, its poor arithmetic properties very fast became a large problem. So, COBOL was very often used as a language for data manipulation, while FORTRAN routines were used for all calculations.

The first standard for COBOL programming language was published in 1960. and called COBOL-60. COBOL-60 was not a real standard, but it was a language specification. After this release the first implementations of COBOL language were released, too.

Of course, not all scientists were very happy about the COBOL syntax. Especially scientists grouped around ALGOL were very bitter regarding COBOL capabilities. For example, Edsger Dijkstra in his letter to his editor in 1975. said that "The use of COBOL cripples the mind; it's teaching should, therefore, be regarded as a criminal act." [23]

Some scientists that were little less critical started to work on the improvement of COBOL syntax to solve the problems that become obvious. Even Dijkstra was amazed with the Michael Jackson's ideas about incorporating structural programming paradigm into COBOL syntax realized in COBOL-85 standard.

However, before that two standards were released. COBOL-68 standard released by ANSI was merely an attempt to overcome incompatibilities between COBOL dialects and did not significantly improve COBOL. The first real improvement was COBOL-74 standard. Among other improvements, maybe the simplest, but one of the most powerful was COMPUTE command, which includes arithmetic expressions and their calculations into the language.

The next COBOL standard, COBOL-85 brought many controversies into COBOL programmers community. It brings many new features into COBOL, and made COBOL more regular. The problem was that COBOL-85 standard was not backward compatible. The syntax of some commands was drastically changed, and old COBOL-74 programs was not correct for COBOL-85 compilers. That divided COBOL community into two parts: one part adopted new COBOL-85 standard and other stayed with COBOL-74.

The last COBOL standard was published in December 2002 by ANSI. This standard brought an object-oriented paradigm to COBOL.

2.3 The High Antique: ALGOL, BASIC, and LOGO

2.3.1 ALGOL

Immediately after FORTRAN was introduced, some scientists became aware of several problems which were connected with the way FORTRAN worked. Some of them were interested in researching ways to solve these problems within FORTRAN, but very fast the group of European and American scientists started to work on the specification of new higher-order programming language whose syntax and semantics would be in accordance to theory of languages. The language that was developed by this group was presented in the meeting that was hold in ETH Zürich in 1958, under the name ALGOL 58 (short from Algorithmic Language). That was not the end of language definition, but the start. On that meeting the committee was founded whose purpose was to standardize the ALGOL language to be much more logical to program than FORTRAN. In this committee there were six scientists from Europe: F. L. Bauer, P. Naur, H. Rutinshauser, K. Simelson, A. van Wijngaarder, M. Woodger, as well as six scientists from the USA: J.W. Backus,

J. Green, C. Katz, J. McCarthy, A.J. Perlis and H. Wegstein. After nearly two years of work they announced ALGOL 60 standard in January, 1960.

Only few months after ALGOL 60 standard was announced, the first implementation of the language was developed by C. A. R. Hoare, called Elliott ALGOL.

Program 6

```
BEGIN
     ARRAY A[1:10];
     INTEGER I, MAX;
     FOR I:=1 STEP 1 UNTIL 10 DO
         READ(A[I]);
     MAX:=A[1];
     FOR I:=2 STEP 1 UNTIL 10 DO
         IF A[I] GREATER MAX THEN
            MAX:=A[I];
     WRITE(MAX);
END.
```

Another side-product of ALGOL definition was so called Backus-Naur form (BNF) grammar, which was used for ALGOL definition for the first time. After that BNF became the standard way of defining syntax of programming languages. It was originally invented by John Backus and presented at World Computer Congress in Paris, 1959. Generative grammars was the field of interest for many mathematicians and theoretical computer scientists as a declarative equivalent to automata. BNF had productions that have general form:

```
<symbol> ::= <expression>
```

where left side of the production contains variable and right side contains regular expression made of variables and terminals. It is easy to see that BNF is, in fact, redesign of context-free grammar as it is known in the theory of languages.

By the suggestion of D. Knuth, P. Naur extended and formalized Backus' idea. The formal system for language syntax known as Backus-Naur Form was presented in 1963.

ALGOL language was not widely adopted by the commercial users, which mostly stick to FORTRAN, but it had a lot of good influences to other languages that were developed after it, such as Pascal and C. The approach that was firstly used in the design of ALGOL brings another very important influence: brought the questions about readability of programs written in particular language. These questions opened a new programming paradigm called *structured programming*, and finally brought alternative to FORTRAN clumsy syntax in languages like Pascal and C.

2.3.2 BASIC

The first two widely accepted languages, FORTRAN and COBOL, were domain oriented. The first one was oriented to numerical intense problems and the second one to data intense ones. The problem with them was that both of them had rather complex syntax, optimized for the domain a language was oriented to. Therefore, the languages were rather hard to learn, especially for the beginners. For that reason, many programming lecturers at universities appealed for the language that will be simple for teaching and that will include basic concepts of both languages described above.

In 1963. J.G. Kemeny and T.E. Kurtz from Dortmauth College, Hanover, USA presented the language called BASIC (Beginner's All-purpose Common Instruction Code). The language was

intended for teaching purposes and as introduction to programming languages. It was not so well accepted at the first time, because computers at that time were rather expensive, and the most of programmers at were computer scientists and mathematicians. As time passed, computers became much cheaper and more non-scientists used them. In the 1970's microcomputers, based on microprocessors were introduced, and in 1980's home computers came to the scene. The most famous "computer from the garage" was Apple II Microcomputer, developed by young computer scientists S. Wozniak and S. Jobs. The Apple II was the first mass produced home computer. It also changed the destiny of BASIC programming language. Namely, Apple II had BASIC as standard programming language in its ROM memory. The millions of sold Apple II computers made millions of BASIC programmers. BASIC shared many critics with FORTRAN. Scientists that were promoting structural programming had the same objection to BASIC as they had to FORTRAN: extensive usage of GOTO (and GOSUB) statements, which makes code highly unreadable. E. Dijkstra in the same letter in which he strongly opposed FORTRAN [23] made a negative observation to BASIC, saying that any person who starts to learn programming with BASIC is forever lost for any serious programming tasks.

After Apple II microcomputers, including BASIC interpreter in ROM memory of microcomputer became standard, most of microcomputers developed after Apple II, such us Commodore microcomputers VIC 20, C64 and C128, Sinclair's ZX 81, ZX Spectrum, and Acorn's BBC had some dialect of BASIC interpreter it their ROM memories.

At the end, IBM's microcomputer concept, that will become de-facto standard, the concept of personal computer, or PC, started with PC models that did not have any hard disks. Three enthusiastic university drop-outs B. Gates, A. Davidoff and M. Davidoff founded a small software company named Micro-soft. The idea of the company was to develop software for microcomputers. In 1975 they developed BASIC interpreter for floppy-based IBM PC computers, known as Microsoft BASIC, or MBASIC. As every user of PC needed some programming interface for her computer, MBASIC was quickly included into floppy bundle that was sold with the IBM PC's. Microsoft made an agreement with the IBM about including their BASIC interpreter into their new, ROM based PC's, and in 1979. BASIC interpreter became standard programming interface for PC's.

The language had no subroutines. Programs were written in one single file (if it can be called a file). Subroutines were simulated by GOSUB command which jumped to the part of the code just like GOTO command. The difference was that when the interpreter came to RETURN command after GOSUB jump, it returned to the command after the command that contained GOSUB jump. The language had single line, logical IF statement without ELSE clause. Therefore, IF was mostly used with GOTO statement. The only loop in the language was FOR loop, that had rather standard form, which was adopted by many other languages after BASIC, such as Pascal. In the original BASIC [95] the only data type allowed was numeric data type. The completely new concept of the language was DATA command that allowed simple lists of data that were read by the statement READ. In that way programs could contain files of data that did not need to be read from the secondary memory during the program execution. The drawback of this concept was that data at the DATA sequences were read-only. Language did not separate floating point numbers from integers, and variable identifiers could contain one letter, possibly followed by one digit. When first commercial interpreters were presented, the string data type was added. Still, language did not yet separate floating point arithmetic from integer one. Rules for identifier building were also slightly different. It still could have maximally two characters, from which first had to be a letter. The second character could be a digit or a letter. Like FORTRAN, BASIC did not have explicit assignment of the data type (except for arrays) to the variable, nor explicit variable declaration. All variables were considered as numbers, unless the identifier was followed by the "$" sign, in which case the variable was considered as a variable of string type. The last new concepts of the language that was different from the languages developed before BASIC (and most of languages

developed after it) is obligatory labeling in programs. Every line of the code had to start with the numerical label, used by GOTO and GOSUB commands for jumping through the program. There were several standard functions for numerical and string data.

The language was conceived to work in the environment that was some kind of rudimentary operating system. Therefore, language had a batch processing. Any command without label were immediately interpreted, while labeled commands was memorized for later execution. There were special commands that were intended to be executed only in batch mode. RUN command executed program in the memory, while LIST command listed it to screen. With the first commercial interpreters LOAD and SAVE batch commands were added for loading programs from and saving to floppy disks.

The whole language was really simple and easy to learn. It had no more than 30 keywords (while COBOL, for example, had more than 400 keywords), and no more than 20 functions.

Program 7

```
10 DIM A(10)
20 FOR I=1 TO 10
30 INPUT A(I)
40 NEXT I
50 LET M=A(1)
60 FOR I=2 TO 10
70 IF A(I)>M THEN M=A(I)
80 NEXT I
90 PRINT M
```

BASIC language was never strictly standardized as FORTRAN or COBOL. In 1978. ANSI standardized so called Minimal BASIC that contained a core features that any BASIC interpreter should have. The full standard for the language was presented many years after the language was developed in the year 1987.

BASIC had its own development, which was, because of the lack of standards, proposed by companies that were developing BASIC interpreters and compilers. The most famous representatives of the second generation were GW Basic and Quick Basic (or QBasic), both developed by Microsoft. They were still interpreters, but the programs were mainly written in standard text editors outside the BASIC environment. Many new structure programming features were added, such as WHILE and DO WHILE loops, subroutines and functions, SELECT CASE multi-selections, operations for working with files, and at the end, obligatory labeling was removed.

The third generation of the language begins with the Visual Basic in 1991. The whole new concept of event-driven programming was developed for the programming with the graphic operating system Windows.

At the end, the development of event-driven BASIC for graphic operating systems brought the forth generation of BASIC language that included object-oriented paradigm.

Microsoft corporation brought BASIC into another very important field - into the field of procedural languages for database programming. Their management system for small relational databases called Access is built around the version of the Visual Basic called Access Basic.

2.3.3 LOGO

BASIC as an educational programming language did not have such a complete success as some scientists were hoping to. It had many syntactic and semantical mistakes, and it was not very good at teaching a theory of programming. So, the search for the perfect educational language continued. It was obvious that theorists will never be satisfied with any language based on FORTRAN

or any other language that results from it. ALGOL had its own followers which were in the conflict with their opponents. It seemed that the only programming language that did not have strong opponents was LISP. But LISP was very complicated programming language in its syntax, as well as in its semantics and functional orientation, and was not suitable for absolute beginners, nor it could be used as the first language to learn. λ-expressions are not something that could be understood by a child, nor is abstraction required by functional programming. On the other hand, some of the concepts presented in LISP, such as its simplicity in data types and recursion orientation looked so natural that it simply had to be included into a new, educational programming language, especially according to the constructivists like Daniel G. Bobrow, Larry Feurzeig and Seymour Papert, who were teaching programming concepts at Cambridge University. The main idea came from at that time very popular book trilogy *Computer Science Logo Style* cite25, [38], [39] that treated computer science in very theoretical, but simple and systematic way. Therefore, in 1967 at Bolt, Barenak and Newmann, the Cambridge University research spin-off firm, Feurzing and Papert developed a new educational programming language called LOGO.

The language was procedural, like FORTRAN, but has been given complete and theoretically based syntax and semantics, as well as very famous visualization tool named turtle graphics that allows very easy visualization of programming concepts. Although LOGO was never considered as a serious programming language for building anything else than concepts in the heads of students, it survived until today as one of the best educational languages and one of the best choices for teaching a beginners course of programming, especially for students of the younger age.

As afore said, LOGO is a procedural language, so it has a sequential structure of command interpretation. On the other hand, it has concepts that were considered as theoretical concepts and were not implemented in other programming languages (except LISP) at that time, such as recursion, lists, and calls of functions based on the programming stack. It had variety of programming constructs, that were not available in other languages. Many parts of LOGO syntax were adopted from LISP programming language, especially a way of threating lists and other data objects, as well as the basics of prefix notation, although prefix notation for mathematical expressions that is used in LISP was abandoned and more common infix notation used. Another concept adopted from LISP was that the language is implemented as interpreter, or, in other words, as an LOGO programming environment.

Program 8

```
to max
make "lst readlist
print (max_rec :lst)
end

to max_rec :lst
make "frst (first :lst)
make "rest (butfirst :lst)
     ifelse :rest = [] [output :frst] [
            make "mrest (max_rec :rest)
            ifelse :mrest > :frst [output :mrest]
                   [output :frst]]
end
```

As it can be seen from the program above, not syntax, but the way of writing programs in LOGO is much like the writing programs in LISP. But, why did we say that LOGO is, in opposite

to LISP, a procedural programming language, and how was it achieved? Obviously, LOGO is like LISP pretty much functional. The difference is that functions in are LOGO not, as they are in LISP, single λ-expressions. They are lists of λ-expressions that are executed sequentially. The second thing is that variables in LOGO are not, like in LISP, defined in the scope of one λ-expression, but are global. That makes easier to pass values between LOGO commands. Therefore, LOGO is, in fact, a functional programming language that can be easily programed procedurally. The program above is highly functional and it is hard to believe that any beginner would write such a program. Nevertheless, the same thing can be solved in LOGO in the procedural way, as it is shown in the next example:

Program 9

```
to max
make "lst readlist
make "max (first :lst)
make "rest (butfirst :lst)
until [:rest=[]] [
      make "frst (first :rest)
      make "rest (butfirst :rest)
      if :max < :frst [make "max :frst]
]
print :max
end
```

The LOGO was, despite its clear syntax, its powerful functionality and its simplicity, considered a as serious programming language. That is the reason why it was never standardized, nor scientists ever wrote serious papers about it. Regardless that, LOGO is language that had, and still has a great influence to the generations of programmers who started their programming in this language. Besides that, LOGO had the great influence to the development of other, more serious languages. The one rather famous language that is based on LOGO programming language is Smalltalk.

3. The Medieval: The Age of Many Programming Paradigms

From the beginning of the development there was a group of theoretical computer scientists who were rather unsatisfied with the development of higher order languages. A letter [23] was already mentioned, written by one of the loudest between them, Edsger Dijkstra in which he criticized FORTRAN, COBOL and BASIC as badly constructed languages that does not meet basic mathematical and logical requirements of programming language construction and as languages that does not allow turning algorithms into computer programs in the natural way.

This group was not only a group of critics which criticize others work and do not propose anything better. Their first big step toward better programming languages was mentioned in the previous section: the project that resulted in ALGOL, the first algorithmic oriented programming language. That was only the first step, though. The next generation of programming languages was mostly their work. Because of that programming languages got better, more standard syntax, richer structure and several new features based on the different internal data structures of computer programs (program stack and program heap).

3.1 The Early Medieval: Structured programming

The introduction of structured programming was indeed renaissance of the computer programming languages. It introduced many fundamental concepts of programming languages that raised programming languages on the new, much higher level.

One of the most famous, but not the most important goals that structured programming had was to eliminate extensive usage of GOTO command that was very common in languages like FORTRAN. Jumps made programs hard to read, but also they made them less logical, very hard to debug and maintain. This fight against jumps in computer programs was rather hard. There were many scientists and programmers which did not believe that jumps can or should ever be eliminated from computer programs. Dijkstra in 1967 wrote a letter [22] to the editor of Communications of the ACM in which he hardly attacks usage of GOTO statement in higher order programming strongly. There was a question that was a subject of many discussions: is it possible to eliminate all GOTO's from computer programs. Final answer to that question was given by so called *Structured program theorem* or *Böhm-Jacopini theorem* [12] that states that any computable function (any algorithm) can be programmed using only sequence, iteration and selection. Many eminent, mostly European computer scientists such as Wirth and Hoare were on his side, but on the other hand, many American scientists opposed him. Surely one of the most famous computer scientists that thought that it was not on the course of structured programming was Donald Ervin Knuth. After few years he recognized structured programming as a legal trend of improving programming languages, but he never entirely accepted, as well as he never gave up GOTO statement. In 1974 he tried to incorporate "the old ways" into structured programming in the paper [48] in which he explains that usage of GOTO statement can be rather painlessly incorporated into structured programming. He even today writes his algorithms in assembler. The main argument that scientists that agreed with usage of GOTO statement had was exception handling, which was beyond any logical structure of computer program and was and still is solved by GOTO, or some other jump command. Although the main battle for GOTO statement finished in 1970's, the war continued for many years after it (and maybe still lasts).

But, as afore said, structured programming was not only a fight against jumps in programming languages. It was the first programming paradigm that was based on theoretical, logical models. It introduced systematic approach to control structures and data structures in programming languages, but top-down program design as well. Structured programming divides lower control structures into two categories: loops and selections. It standardizes two types of loops: loops based on counting and loops based on logical condition. The first category contains FOR loop, while the second one contains, by the original idea presented in structured programming paradigm, loops with condition in the beginning of the loop, with the condition at the end, and with the condition in the middle of the loop. Some of the older programming languages (such as Ada) based on structured programming paradigm had all three types of loops based on logic condition, but at the end, the third type was abandoned as too complicated to understand and implement.

Regarding selections, structured programming paradigm mainly define IF and IF..ELSE selection. Many languages have more general CASE or SWITCH selection, but this type of selection often goes with some structural problems (as in programming languages based on C programming language).

Structured programming paradigm defines higher-level control structures also. On this level the subroutines and functions are considered. One of the requirements that languages form previous generation did not meet was recursion. As it was said before, FORTRAN had direct addressing strategy of subprogram calls, which was very efficient method of calls, but, on the other hand, because of it recursions were not possible. Because of structured programing requirements of recursion, the novel approach was developed: the calls of subroutines through program stack.

In addition, new type of variables was introduced – the variables that contain memory address of other data – pointers, and according to that, another program structure, that was dealing with this kind of variables: program heap.

3.1.1 Pascal

Although ALGOL is definitively the first programming language that was based on ideas of structured programming, it was developed before the structured programming paradigm was fully defined. The first programming language that was fully developed in accordance to structured programming paradigm was Pascal.

Pascal was developed by Niklaus Wirth at ETH Zürich. The first Pascal compiler was developed in 1969 and was written in FORTRAN. Therefore, it had many restrictions that FORTRAN had, and did not cover all advantages of the paradigm. Nevertheless, it was a very big breakthrough in programming languages design because it showed that paradigm can be implemented in the efficient way. In the next 20 years Pascal became one of the milestones of the development of programming languages and had influence many programming languages that were developed after it, such us Ada, C and Visual Basic.

Program 10

```
PROGRAM Maximum(INPUT, OUTPUT);

VAR a:ARRAY [1..10] OF INTEGER;
    i, max:INTEGER;

BEGIN
     FOR i:=1 TO 10 DO
         ReadLn(a[i]);
     max:=a[1];
     FOR i:=2 TO 10 DO
         IF a[I]>max THEN
            max:=a[I];
     WriteLn(max)
END.
```

Pascal introduced some of the very important programming language constructs into programming languages that were defined theoretically by the structured programming paradigm. Besides loops based on counting (FOR), the language had logic based loops that were for the first time included in the language (WHILE and REPEAT). Pointers, as a new way of addressing memory from the higher order programming language have to be mentioned, as well as recursion, tools for data structures definition (TYPE) and totally new way of threating subroutines that enables recursion through the newly defined internal data structure - programming stack, full IF-THEN-ELSE selection, and generalized CASE selection.

The language inherited the relatively strict organization of programs into data definition and procedural sections where all data has to be defined before the beginning of the procedural section of the program. On the other hand, Pascal introduced a very flexible, multi-level structure in the data definition, as well as the subroutine definition. The abstract data types are allowed to be defined, using four basic data constructs: arrays, records, sets and pointers. The first versions of the language had total lack of jump instructions, which consolidated authors beliefs in structured programming paradigm. In the later versions of the language GOTO instruction was added, although it has never been extensively used in Pascal programming language.

Program 11

```
TYPE
    nm=RECORD
                first:string;
                last:string;
    END;

    addrss=RECORD
                  country:string;
                  city:string;
                  street:string;
                  no:INTEGER;
    END;


    prec=RECORD
               name: nm;
               address:addrss;
               year:INTEGER;
               salary:ARRAY [1..12] OF REAL;
    END;

    list=^list;

    rlist=RECORD
               data:prec;
               next:list;
    END;

VAR
   employees:list;
```

From the procedural point, multi-levelness was introduced in the definition of subroutines, allowing local, nested subroutines:

Program 12

```
FUNCTION ModPlus (m:INTEGER,n:INTEGER,b:INTEGER):INTEGER;

         PROCEDURE Modulo(VAR n:INTEGER,b:INTEGER);
         BEGIN
              IF n>b THEN n:=n MOD b;
         END;

BEGIN
     ModPlus:=m+n;
     Modulo(ModPlus,b);
END;
```

Although in the mid 1970's a new language arrived that was in many ways, especially in the lower-level programming, superior to Pascal, called C, Pascal was one of the most used all-purpose programming languages for more than 20 years. The great merit for that surely goes to Borland Software Co. and to Anders Hejlsberg who developed Turbo Pascal compiler and programming environment in 1980. Turbo Pascal gave, besides very reliable and fast compiler for Pascal, one of the most popular programming environments that was used in the next decades not only for programming in the Pascal programming language, but was also implemented with other programming languages such as C in Borland's Turbo C and Turbo C++, and Microsoft C++, Borland's Turbo Basic, Turbo Prolog and so on.

The proof of Pascal popularity is the fact that many computers and its operating systems from that time took Pascal as a main programming language. Even more, some of the operating systems from that time, such us Apple Lisa and earlier versions of MacOS, as well as some versions of Burroughs' BTOS were written in Pascal in whole or in some segments.

In the 1980's and especially early 1990's Pascal began to lose its popularity due to serious criticism by the followers of C programming language [45]. The main objections were the strictness of the data definition, which resulted in many data conversion functions, its strict structure which was more appropriate for teaching than for commercial use, as well as its closed, controlled use of pointers, where programmer was not able to address memory directly, but through Pascal memory assigning machine. During the time many of this objections proved to be false, especially the last one. The modern programming languages such as Java and C# abandoned C-style of using pointers and turned back to Pascal controlled pointers which assures much less insecurity in pointer usage.

Regardless of growing popularity of C programming language, followers of Pascal's purity of programming never gave up. That was not only the fight of followers for their beliefs, but also the battle of corporations for their concepts of development of programming languages. On one side, Borland Software Co. and Apple Inc. stick with Pascal and its derivates, while Bell Laboratories and Microsoft Inc. tried to push it out of use with their C and C++ programming languages, languages that were scarified much of Pascal's purity and security of programming for efficiency, direct access to the memory and computer resources. The breakthrough that C and C++ done, as well as problems in the development of these languages will be covered in the following section.

Two main persons in further development of Pascal were Niclaus Wirth with his more theoretical line of development, and Borland's Anders Hejlsberg who tried to build up the language to meet growing technical requirements that C and C++ languages imposed to all other languages.

In the 1978. Niklaus Wirth developed a new programming language (that was never in wide use outside of the academic society) named Modula, the general-purpose programming language designed to meet requirements of programming in a team. Two years later, in 1980. Wirth published a new definition of the language called Modula-2 [92]. The language did not really give many new properties which would bring it in front of than very popular Pascal. The only reason that Modula-2 really stayed alive for a some time was its famous author.

The battle of Pascal versus C looked like a draw until Bell Laboratories announced new, object-oriented language based on programming language C - the programming language C++. That was the time when Pascal's popularity started to fall. The answer comes from both Wirth and Hejlsberg.

In the year 1986 Niklaus Wirth founded Oberon project whose main goal was to build object-oriented operating system. As a programming language for system programming Modula programming language was chosen at first. But, very soon it became obvious that Modula was not good enough for that job, both because of speed and executable code length. Wirth developed in 1991 a new, completely object-oriented and much more compact programming language based on Modula-2, named Oberon, the language that kept clarity of Pascal and brought complete, theoretically based object-orientation. The syntax was almost the same as one of Pascal and Modula-2,

but more concise and consistent. Unfortunately, Oberon came much to late and never had a chance against C++ and Hejlsberg's Delphi.

Hejlsberg was much faster in his answer than Wirth this time. Apple Software Co. needed three years to define Object Pascal which merely copied C++ object-oriented constructs and brought them to Pascal. Both Wirth and Borland were included in this project, so just a few months later Hejlsberg and Borland came with Turbo Pascal version 5.5 which was based on Object Pascal.

Little later, in 1991. Microsoft Inc. brought a new challenge to Hejlsberg and other Pascal followers. The main problem of programming languages of that time was that they were not compatible with the new programming paradigm based on graphic-oriented operating systems, such as Microsoft Windows and MacOS. So, in 1991. Microsoft developed Visual Basic, programming environment based on the programming language that was the first great success of the corporation – BASIC. Visual Basic was not merely a new programming language. It presented a completely new programming paradigm - event-driven programming. The Visual Basic was the first rapid application development (RAD) environment for event-based operating systems. The paradigm allows much easier programming in the Windows and MacOS environment, bringing a bunch of built-in procedures with complete control of operating system objects and triggers to system events. Borland corporation needed five years to bring their answer. The main developer was was Danny Thorpe, who with Anders Hejlsberg developed Delphi, RAD environment based on Object Pascal and event-driven programming paradigm. Delphi had all advantages of object-object oriented and event-driven paradigms, and Microsoft did not have the real answer to this system for more than 10 years. The main disadvantage of Delphi system was that it was strictly bounded to Microsoft Windows operating system and that excluded growing UNIX and Linux community. In 2001. Borland team leaded by Danny Thorpe developed a Delphi equivalent for Linux operating system, named Kylix. The intention of Borland was obviously to bring Object Pascal and Delphi platform to ever-growing Linux society, which becomes more and more important and takes an important part of software market. Kylix did not have significant success in Linux community, because it was traditionally bounded to C programming language. So, after few years of developing Kylix besides the Delphi, Borland gave up of further development, opened Kylix code and after some time announced discontinuing of Kylix development. Todays most similar Linux project to Delphi is open-source project named Lazareus.

Standardization did not have such a great impact to the development of Pascal programming language as was the case for FORTRAN or COBOL. The reason for that was that Pascal language was rather strictly defined from its first appearance, and Niklaus Wirth as well as Anders Hejlsberg closely watched its development from the beginning to nowadays. Regardless that, the language was standardized by ISO/IEC 7185 and ANSI/IEEE770X3.97-1983 standards in 1983. The second set of standards regarding Pascal came in 1990 by ISO/IEC 10206 standard and in 1993 by ANSI organization pointer to ISO 7185-1990 standard.

3.1.2 C

The C is the programming language that probably had, besides Pascal, the greatest influence to the development of modern programming languages. Unfortunately, its development was not so theoretically founded nor so straightforward as the development of Pascal. The formulation of the language was much more ad-hoc and that brought many problems that stayed until today.

Although C and Pascal have the same roots - ALGOL and PL/1, and Pascal was also partially a model for designing C programming language, C differs much from Pascal. The difference is maybe not so obvious in the syntax of the language, because both languages have the same root, but in the way the languages treat programming objects and language constructs.

The first definition of C programming language came from Bell Laboratories young researcher Dennis Ricthie. The language was developed as all-purpose programming language and its main intention was to be a language for UNIX operating system development. This main purpose had a great impact to the structure of the language and its way of treating language objects and computer resources. Although C programming language does not have any new concepts regarding Pascal, many of concepts were drastically changed to become faster, easier to compile and closer to an assembly language.

In 1969 AT&T Bell Laboratories began the development of a language based on ALGOL definitions that would have many low-level capabilities, to be used by UNIX developers, but would also be machine independent. That means that C, despite it's low-levelness does not depend on the concrete organization of operating system that is implemented on a computer. That was the important property that allows C programs to be ported to any computer regardless version of operating system that is running on it.

The direct predecessor of C programming language was Ken Thompson's B programming language defined in 1969. And built on the syntax of B programming language in the year 1972 the first C compiler was published for UNIX operating system. In 1973 all UNIX kernels were re-written in C and C became the standard language for UNIX programming. In that time UNIX, regarding theoretical foundations, security properties and price, became a standard operating system for minicomputers and medium size business computer environments. With the rise of UNIX popularity in the most productive part of the computer community – business oriented programmers and operating system developers – and with the properties that the programming language had, C started to push all other languages out of use. The only language from that time that survived this big-bang of C programming language was Pascal. All other all-purpose languages were literally banished from the market or pushed to the isolated islands of worshipers.

Program 13

```
#include <stdio.h>

int main() {
    int a[10], i, max;
    for(i=0; i<10; i++)
       scanf("%d",&(a[i]));
     max=a[1];
     for(i=1; i<10; i++)
         if (a[i]>max)
            max=a[i];
     printf("%d",max);
     return 0;
}
```

The main advantages of C comparedto Pascal are already mentioned: low-levelness and machine independence. These requirements of C programming language implied more differences in the semantics of the programming language. The first, and maybe the most important one is significantly weaker typing of C variables than in Pascal. Variables became what they are - named parts of memory, a memory containers. This property made data type conversions much easier, and usage of variables much more relaxed so that programmers had much more freedom in accessing the computer memory. In addition to this, pointers that were presented in Pascal (so-called controlled pointers) were redefined into the concept of direct pointers, through which programmer can freely access to any memory address. This, of course, is advantage, especially in operating

system design and development, but also could be a problem: this is the reason why C programs can crash in much more peculiar ways than programs in any other programming language. In fact, in modern programming age, when low-level programming became rare and programming turned to fast development of bigger programs or information systems, it became a serious drawback. That is why modern programming languages based on C programming language, such as Java and C#, abandoned this property of C programming language and turned back to Pascal typing and pointer definition, which brings much more security and comfort in programming.

The usage of pointers in Pascal was optional. On the other hand, usage of pointers in C programming language became obligatory, and cannot be avoided, even in the simplest programs, such at the one presented above. The pointers to functions were also added into the language. This property brought a rudimentary polymorphism that is fully developed in C++ programming language.

Besides that, many operators that optimize code were added, so-called compound operators, such as ++, –, +=, -=, *=, /=. This operators allow programmers to build even more efficient programs.

C core language is very simple, contains only several tens of most of basic instructions and program concepts, while the most other functions are defined in standard libraries that can be included in the program if needed. That brings compactness of the program, without unnecessary ballast in the code.

Different semantics based on jumps of the generalized SWITCH CASE selection are defined. In this case, but also in many other cases C programming language brought jump instructions back into programming. Although GOTO instruction is still not used widely, other instructions, based on code discontinuation are introduced. These are RETURN, BREAK and CONTINUE instructions. Those instructions discontinue code of functions and loops. They were the target of many criticisms from the structured programming puritans.

At the end, the syntax of C programming language allowed definition of variables within program blocks, bringing a new concept of defining a scope of the variable. This property is very important in code optimization because it allows programmer to define variables to be allocated as long as they needed to be, and not, like in some other programming languages, such as Pascal, where all variables defined in main program part have to exist all the time program is running. On the other hand, nested functions were not (and still are not) allowed in C programming language.

The first, informal specification of C programming language came in 1978. by B.W. Keringham and D. Ritchie [46], known as K&R C. The compound operators of type $=-$, $=+$, $=*$ and so on were substituted by $-=$, $+=$, $*=$ and so on. The reason for this change was the ambiguity of expressions of type $a=-1$, which could be interpreted as $a=-1$ or as $a=-1$. Besides that, several new data types like LONG INT and UNSIGNED INT were added. Regarding function declaration, a new data type (if it can be called a data type) VOID was defined, allowing subroutines that does not return a result to be defined. At the end, the important property, that was not defined in the original C definition was added - the STRUCT declaration, that allows easier management of abstract data types.

The first de iure standard of C programing language was published by ANSI in 1989. This standard was adopted by ISO in 1990. The standard was based on K&R C which was slightly widened.

In 1995. ANSI/ISO published the normative amendment to ANSI standard from 1990. This was not enough, and standardization of C went further. The new standard was published by ANSI/ISO organization in 1999, including in line functions, LONG LONG INT, COMPLEX data types, variable length arrays and variadic macros. This standard is still active, and its last revision, informally called C1x is defined in 2007.

Most criticism of C programming language target its minimalism, which makes programming of larger applications really hard, its lack of control over program regarding data types as well as

memory usage. This makes C inappropriate for less experienced programmers. Although Ritchie and Keringham argued this remark with the statement that C was never mentioned to be closed regarding data types and memory usage, even some of the most important people in the C/C++ development, such us Bjarne Stroustroup agreed with it. In 1986 Stroustroup said "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it, it blows your whole leg off." Other complains target a lack of garbage collection in C programming language, lack of boolean type, lack of native support for multithreading, lack of nested functions, lack of complete support of polymorphism and encapsulation and so on. This is the reason why the new programming languages based on C programming language, such as C++, Java and C# starts to appear. The last set of criticism addresses the C programming syntax that makes use of some advanced concepts, such as pointers, obligatory, and makes C not so good as a first programming language to learn. Some concepts of C programming language, such us compound operators, discontinuing instructions and new FOR syntax makes C source code hard to read, and that is another strong disadvantage of C programming language.

C programming language is still in wide use, especially in operating system programming, mainly by UNIX/Linux programmers, but C looses its battle against new languages in programming other applications that do not have such a great need for code optimization, but have great and ever growing need for programming speed. The another reason for this is the fact that until the very recent history there were not good rapid development environments for C/C++ programming language that could be compared with Visual Basic or Delphi environments. The first really usable RAD tool was Borland C++ Builder developed in 2003.

3.2 The High Medieval: The Dispersion of the Programming Paradigms

This section will go through 1970's and early 1980's - the time when the large amount of programming languages was created that differing in the programming paradigm they were intended to cover. As well, many programming languages were developed for special purposes.

The two-dimensional classification of programming paradigms became obvious and most of the nodes in the graph of programming languages classification were populated. Programming languages, in accordance to this two-dimensional classification are classified depending on their static and dynamic part. Regarding the dynamic part of programming language – the way that algorithms are described in the programming language, languages are classified in imperative or procedural languages, functional languages and logic languages. On the other side, regarding the static part of programming, languages are classified to relational, record or structure oriented and to object oriented languages.

Functional Programming

The development of programming languages started from the bottom-left corner of the graph. That means that the first languages, such as FORTRAN, COBOL, and ALGOL were procedural, structure-oriented languages. But very soon, only a few years later, the first functional programming language arrived. The reason for it lays in several things. The first, and maybe the most important one was that introduction of higher programming language was the process that brought easiness into computer programming. Therefore, the tendency of further simplification of programming process became natural. This tendency that drove into functional programming development. The most of the computer code were dealing with standard calculations, sorting and other standard actions that repeated over and over again in every computer program. The idea was to transcend the programmers thinking from the question "how to do" something to the question "what to do". In the other words, the idea was to build system (server, interpreter, management system) with standard algorithms for standard actions that would allow programmer only to describe what (s)he want to do, and leave the algorithm selection to the interpreter. The

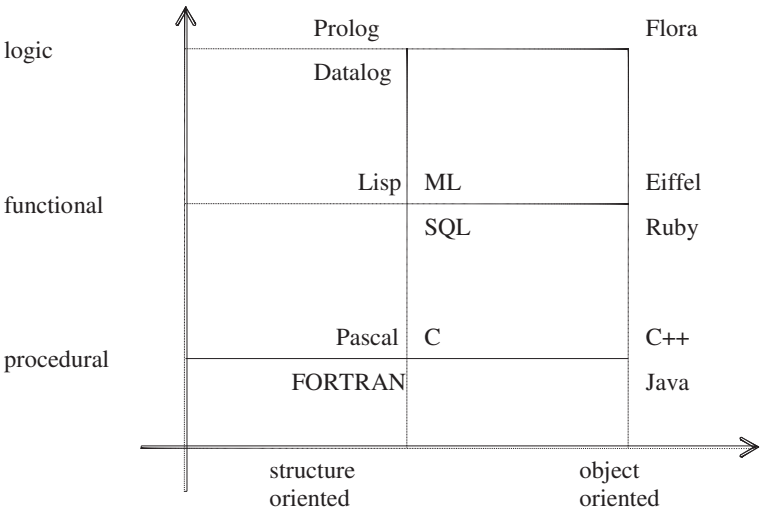

Figure 1: Classification of Programming Languages [47]

second reason that was important for the functional programming development was the theory, mathematical model of λ-calculus that was functional-oriented. The the functional programming languages (Lisp, Logo, ML) were, in a way, an attempt to implement λ-calculus as a programming language.

To resume, functional programming can be seen as a style of programming whose basic computation method is the application of function to arguments. Functional programming language is one that supports and encourages functional style. Its name comes from the fact that program written in such language consists entirely of functions (see program 2). More often then not, main function is defined in terms of other functions, which in turn are defined in terms of even more functions, and this goes all the way down to language primitives. Functional programs do not contain assignment statements, so variables, once they get a value, never change. Functional programs have no side-effect, meaning that a function cannot have any other effect then to compute a result. Such feature eliminates major source of bugs and makes order of execution irrelevant. Because their foundations were very simple, they give the clearest possible view of central ideas in modern computer world, including data abstraction, genericity, polymorphism and overloading, the concepts that were later adopted by object-oriented paradigm and implemented in procedural programming.

Logic Programming

The next step in this direction was introduction of logic programming. Logic programming developed even further the idea that was brught in by the functional programming. While it kept the main idea of functional programming, it brought more clearer non-procedural syntax based on the first order logic language, as well as the terms of rules, that embody implicit data and knowledge, and facts that are used to define explicit data. In other words, logic programming contains two new concepts that are not clearly defined in the functional programming: the concept of implicit data and knowledge – the data do not have to be explicitly defined in the program, but are derived by the derivation rules from facts; and, the second, the new organization of data in the facts – the dynamic data base that is the part of the program. This style of programming is popular for data base interfaces, expert systems, and mathematical theorem provers.

In opposite to classical mathematical logic languages, logic programming languages and Prolog, as their most famous representative, beside syntax and declarative semantics have procedural

semantics [25]. Procedural semantics allows logic programming language to make a bridge between mathematical logic that is completely declarative and computer programming, which is in its base procedural. That makes all logic programming language different from first order logic (FOL) language in its very basis. According to definition of FOL one does not have to think about efficient way to interpret logic formula. That is the reason why logic programming languages differ from FOL in their syntax, but also in their declarative semantics.

The basis for procedural semantics of logic programming languages is so called Herbrand theorem, proved by Jaques Herbrand in 1930. According to this theorem every formula of FOL can be equi-contradictory clause. For clauses, on the other hand, efficient interpretation, called *resolution*, exists. Resolution process was developed by J. A. Robinson in 1965. [72], [15], [56]. Although resolution was the procedure that the first programming language procedural semantics were based on, today there are other procedures, not based on resolution that are used to build procedural semantics of logical programming languages. One of the most famous procedures that is used besides the resolution, is *stratification*, defined by W.V. Quine in 1937, on which is based Datalog procedural semantics [14].

But not only procedural semantics is added and syntax specialized. The very interpretation of the formulae, and that means the declarative semantics, is changed in the way to better serve the logic programming. Namely, in FOL not all formulae have to be uniquely interpretable. Even not all clauses have to be so. The problem is with the definition of negation. FOL language is not closed to negation. That means that negation of satisfiable formula does not have to be unsatisfiable. That is not acceptable for programming language, in which interpretations for all formulae have to be defined. That is the reason why declarative semantics of negation in logic programming languages differs from the definition of negation in FOL. The way that the negation is defined in the logic programming languages is based on the *closed world assumption*, which says that everything that is not true is false. Based on that assumption, the equality in logic programming languages is defined differently than in FOL. The definition of equality in logic programming languages says that two terms are equal if and only if it is defined so in the program (explicitly by a fact or implicitly be a rule). This way of defining equality is called intensive equality definition, while FOL has extensive definition of equality.

Object-Oriented Paradigm

The idea of object-oriented paradigm was to make easier to manipulation of data and functions attached to it through the complex computer program. Usually, structure-oriented programming languages have data and functions separated, and there is no efficient way (although modular programming tried to solve these problems through breaking programs in modules that, in a way, simulate objects) to define data structure and operations on it as a compact unit. However, as it is well-known, data and functions that process it are inseparable. This idea was the basis of object-oriented programming languages development. The object-oriented programming languages do not define data structures and functions on them separately, but enclose them all in objects. Mainly, objects are static entities in the programs, although they contain functions, called methods, that are used for object data processing. The reason for that is the fact that objects are built around data they contains. The definition of the object is called the class. The class defines types of data objects of the certain class have, as well as methods of processing them. Objects are instances of some class, containing real data in the processing [2]. Objects do not contain any methods and that is the reason why object-oriented paradigm must be seen as a way of data organization in computer programs.

Although the beginning of the object-oriented programming was in the late 1960's, when Simula and Smalltalk programming languages were developed, the real expansion of object-oriented

languages began in the late 1970's and in the 1980's when C++ programming language was developed.

The paradigm was created in 1967 by Alan Kay and in his words it is "only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things".

Object-oriented programming paradigm introduced several new concepts[5]. Some of them, such as class, object, instance and method were mentioned above. However, there are a several other concepts that are connected to the object'oriented paradigm, although some of them were firstly introduced independently from it. Maybe the most important and the most original concept of object-oriented paradigm is *encapsulation* – the binding of data structures with its methods. This organization of data enables *decoupling* or dividing programs into independent parts that communicate with with each other. This communication has to go through the *message passing*, because with encapsulation implementation, a part of a program cannot access directly the data of other parts, nor it can call other parts methods. The only way to communicate is through defined channels and interfaces. The second, and maybe the most controverse concept of object-oriented paradigm is *inheritance* – the idea of hierarchical organization of classes in a program, where subclasses can inherit data definitions and methods from a super-class. This concept is natural in the object-oriented programming, but in the implementation it brought many problems that address conflicts of priority of data and method definitions in classes. The solution of this problems leaded to the *polymorphism* – the concept of co-existence of the entities with the same name in a program, as well as the concept of overlaying of definitions. At the end of discussion about object-oriented paradigm, a series of reoccurring problems should be identified. Concepts called *design patterns* [31] were created to help in solving these series. Simply said, design patterns are reoccurring solutions for reoccurring problems.

***

Most of the programming paradigms were covered in this period in the many programming languages that were developed. Naturally, many of them did not survive and do not have any influence today. However, in this period this diversity was necessary because it was the time of defining main concepts paradigms as well as solutions for some specific problems in the programming languages development. Some of the concepts were weird (like loops with the logical condition in the middle) and died with the language in which they were introduced, but some of them make the base of the modern programming language.

3.2.1 Prolog

Some mathematic is behind everything. While this is arguable, one of the areas where mathematical logic is very important is the logical programming. In this paradigm, the programmer specifies relationships among data values that constitute a logical program and then sends queries to the execution environment in order to test those relationships. In other words, a logic program through explicit facts and rules defines a base of knowledge from which implicit knowledge can be extracted. This kind of programming is very useful when creating database interface, expert systems and mathematical theorem provers. One of the most popular logic programming languages in the field is certainly Prolog. Built in the year 1970 by a team of people with Alain Colmerauer as a leader, it quickly gained traction. Its first compiler was written in Edinburg, Scotland by David H.D. Warren. From that date Prolog continues to maintain its popularity.

Prolog is based on so-called SLDNF (Selective Linear Definite resolution with Negation as Failure) resolution algorithm [72], [56]. The definition of syntax, declarative and procedural semantics of Prolog, that is completely adopted from the theory of logic programming, shortly described above allows resolution (that is generally procedure of the exponential complexity) to

be linear. That makes Prolog efficient, but, on the other hand, it makes it vulnerable to infinite loops.

Prolog commands are not procedural, but declarative. They do not define the way in which the program should be executed. They only define data that are true in the context of the program. Therefore, every row of the Prolog program can be seen as definition of some data or as definition of the rule that can be used to calculate the data that are also true in the context of the program, but are not defined explicitly. The Prolog commands are called clauses, because they have a mathematical form of Horn clause. We can say that Prolog, as well as the most of logic programming languages, looks at the program as main memory database. That means that the order of the clauses in the program is mostly not relevant. Unfortunately, there are cases in which order of the clauses matters, ant that is the main objection regarding the Prolog that can be heard from the scientists and engineers.

The Prolog clause has following general syntax:

```
p(X1, ...,Xn):-q1(Y11, ..., Y1k1), ..., qm(Ym1, ..., Ymkm).
```

p, $q_1, \ldots, q_m$ are called predicate names. In clause is the definition (not necessary the only one) of the predicate n-ary p in the program, through the predicates $q_1, \ldots, q_k$ which should be defined somewhere else in the database (program). X's and Y's are terms or variables. Furthermore, all variables that are defined on the left side of the clause have to occur at least one tim more on the either side of the clause. Also, variables that are not defined on the left side, but only on the right side of the clause have to occur at least twice, to be bounded. Simply said, every variable that occurs in the clause has to occur at least twice and at least one of the occurences has to be on the right side of a clause.

Program 14

```
max(X,[X]).
max(Y,[Y|Z]):-max(X,Z), Y>X.
max(X,[Y|Z]):-max(X,Z), not(Y>X).
```

The special type of clauses are facts. The facts are clauses that have no tail (right side of the clause), nor neck (:- connective). They are used for explicit data definition – the definitions of facts.

Program 15

```
father(joe, george).
father(joe,georgina).
father(phil,joe).
father(paul, phil).

ancestor(X,Y):-father(X,Y).
ancestor(X,Y):-ancestor(X,Z), father(Z,Y).
```

The predicate in Prolog is not defined only by its name, but by its name and arity. That means that Prolog will consider the definitions that defines the predicate with the same name, but with the different number of arguments as a definitions of two completely different predicates. That, of course, means that Prolog has polymorphism defined, regardless it is not object-oriented language.

The terms are constants, variables or functions. Generally, constants are terms that start with small letter or digit, while the terms that start with capital letter or underscore are considered as

variables. There is possibility to define terms that contains more than one word or starts with capital letter as a constant by enclosing them within single quotes. The functions are special type of terms that contains more than one value. The variables in Prolog are called logic variables. They differ from the variables in the procedural languages, because when once logic variable is unified with some value, its value cannot be changed (similar to functional languages).

If there is no need for some argument in the predicate, it can be excluded from the resolution using special term _. For example, in the database of fathers described above, we could add the predicate that will define the condition when some person has a father (in the database) as follows:

Program 16

```
not_orphan(X):-father(_,X).
```

At last, there is a third type of clause in Prolog – query. Queries are not very often used in the definition of Prolog program, but they have a main role in the interaction with the Prolog engine. A query in Prolog is the clause without the head (left side). The queries in Prolog are expected to give answer yes or no, and all other output is considered as a side-effect of the query. As a side-effect the query will return the values of all unbounded variables (variables that occur only once in the query) after each successful unification.

For example if we apply the query

```
:- ancestor(joe, georgina).
```

it will answer to the question ”Is Joe Georgina’s ancestor?” On the other hand, query

```
:- ancestor(X, georgina).
```

will answer to the question "Does Georgina has any ancestors in the database?" Of course, the variable "X" is unbounded, so as a side-effect, this query will give names of all Georgina’s ancestors. If we want to avoid this side-effect, we can make a query as follows:

```
:- ancestor(_, goergina).
```

There are some differences between logic database from classical, relational database. Of course, a logic database is the part of a program, and, because of that, it is contained in the main memory, but this is not a main difference. The main difference is that Prolog, unlike SQL or other classical database programming language, does not recognize arguments of predicates by their name (terms in Prolog do not have names), but by their position in the predicate definition.

Before the end of the Prolog syntax and semantics presentation let us go back to the program 17. If we make a query

```
:- ancestor(X, goergina).
```

it will eventually end in infinite loop (that will be broken by the program stack overflow) after it output all ancestors of Georgina. Why is so? Because before the try to unify variables with the facts from $father/2$ predicate, program will try to unify them with the $ancestor/2$ predicate and that will drive it into infinite loop. With just a small change in the program, rewriting the last clause as

```
ancestor(X,Y):-father(X,Z),ancestor(Z,Y).
```

the problem is solved. Namely, if we change it this way, non-recursive $father/2$ predicate will be considered first and it will be finished before it runs into infinite loop. The recursion implemented in the last clause is called *tail* recursion and the every program should use it, as it minimizes the threat of infinite loops. This problem can be even bigger if recursion is defined before the facts in the Prolog program. For example, program

Program 17

```
ancestor(X,Y):-ancestor(X,Z), father(Z,Y).
ancestor(X,Y):-father(X,Y).

father(joe, george).
father(joe,georgina).
father(phil,joe).
father(paul, phil).
```

which should have the same model (the same answer and the same set of side-effect returned values) as modified program 15, will run into the infinite loop even before it it gives any answer. This irregularity shows the main problem and is the cause of the most objections to Prolog. Prolog, which should be completely declarative and should not depend on the order of the clauses in the database, strongly depends on it. The problem is addressed to the way SLDNF resolution works. In its foundation there is the depth first search backtracking algorithm. Therefore, if algorithm tries to develop infinite branch of the answer tree it will never develop any other branch.

The second most important problem that the opponents of Prolog emphases is the problem of the cut. The cut is a special procedural Prolog predicate that affects the backtracking algorithm by bounding some of the branches of the answer tree. Cut is highly non-declarative property in the totaly declarative Prolog language. Although the cut can be avoided, and is unpopular, it is not excluded from Prolog only because of backward compatibility reasons.

Modern Prolog programs deal with these problems in several ways, all defining a new types of resolution. The last, rather successful attempt to solve this problem is defined by the group of scientists leaded by Warren and Kifer, by defining tabbled resolution with delaying, called SLG resolution. This resolution is implemented in XSB Prolog system, the system that allows HiLog (higher order logic) programming and Flora (object-oriented logic) programming.

Prolog is, in its original definition, untyped language. That means that variables do not have to be declared to any data type before their usage. This property works fine (although it is slightly slower) in interpreters. However, in the history of Prolog there were typed definitions of Prolog (mostly connected to the Prolog compilers, such as Borland's Turbo Prolog and Visual Prolog). Data types strongly affected clarity of Prolog programs and Prolog interpreters gave a better solution than compilers in the most cases, so typed Prolog as well as pure Prolog compiler development were abandoned.

Because of its syntactical pureness, simplicity and rather good theoretical foundations standardization of Prolog is not so much developed as it is the case for procedural languages. Regardless that, there is ISO Prolog standardization (ISO/IEC 13211) published in 1995 that ensures that the core of Prolog, including syntax, streams and some built-in predicates remains fixed.

3.2.2 Smalltalk

Smalltalk is object-oriented operating system and programming language developed at Xerox Corporation's Palo Alto Research Center (PARC) [43]. Besides of being object-oriented, Smalltalk is also dynamically typed and reflective programming language. The language was first generally released as Smalltalk-80 and has been widely used since. Smalltalk has many variants and there is a large community that develops it to present day.

The beginning of Smalltalk was very simple. A few developers made a bet that a programming language that would be based on idea of message passing could be written in a page of code. The very first version of Smalltalk, inspired by Simula was Smalltalk-71 [49]. Versions that followed were: Smalltalk-72, Smalltalk-76 and finally Smalltalk-80. Smalltalk-80 was the first version

that was given outside of PARC to a small number of companies and universities for review [33]. In 1983 a generally available implementation known as Smalltalk Version 2 was released in the form of image and the virtual machine specification. ANSI published a standard reference for Smalltalk in 1998. Two popular implementations of Smalltalk today are Squeak [26], which is an open source version and VisualWorks [87].

Every Smalltalk command represents a definition of an object or a message to an object that triggers some object's method. The program in Smalltalk can be seen as a sequence of messages that flow between objects. As it said before, Smalltalk is reflective language. That means that all features that language has can be seen as basic constructs of the language – objects and messages. In this light, all data types, elementary or compound, can be seen as classes, while variables can be seen as objects. Smalltalk contains variety of compound data types: arrays, bags (sets), stacks, caches, dictionaries etc. All operations, on the other side, can be seen as messages an object respond to in a particular way.

The main syntax construct in Smalltalk is

<object> [*<message>* [*arg1* [. . .]]]

This sends a message with or without arguments to an object. The second, different syntax construct is

<variable > **:=** *<Smalltalk code>*

which assigns a value to the variable. Although Smalltalk is untyped language, there is a syntax construct that allows local variables definition:

| *<var1>* [*<var2>* . . .] |

And, at the end, for more complex calculations, there is a block construct that allows to several Smalltalk commands to be executed sequentially.

[[*arg1* [*arg2* [. . .]] |]
<statement1>
<statement2>
.
.
.
]

Program 18

```
| max |
a := Array new: 10.
1 to: a size do: [ x | a at: x put: stdin nextLine].
max := a at: 1.
2 to: a size do: [ x |
                   max < a at: x ifTrue: [max := a at: x]].
max out.
```

Many programming languages were influenced by Smalltalk in many aspects: syntax and semantics, it also served as a prototype for message passing style (a model of computation), its

WIMP GUI inspired look of personal computer's environments and the integrated development environment served as a prototype for visual programming environments (for example, Smalltalk code browsers and debuggers). Some of influenced languages are: C, Java, Python, Ruby, Perl 6, etc.

Smalltalk is a pure object-oriented language and its objects can hold a state, receive a message from another Smalltalk object and, in a course of processing a message, send a message to another Smalltalk object [51].

Smalltalk syntax is very minimalistic. There are only six keywords (these are not actual keywords but rather pseudo-variables -– singleton instances because Smalltalk does not really define any keywords) reserved: true, false, nil, self, super and thisContext. However, this does not mean that programming in Smalltalk is simple. Namely, Smalltalk has over 200 base classes that can be used in programs. This feature gives Smalltalk flexibility and power in computation. Nevertheless, even programmer with great experience in Smalltalk programming will have to look in the reference manual from time to time to see which class and method to use to solve a problem.

Smalltalk is pure object-oriented language, which means that everything is an object and that it provides support for complete encapsulation, inheritance, dynamic binding and automatic storage management.

When compared to Java, it can be said that Smalltalk did not succeed to become as popular and recognized in time as Java. There are several reasons that can be mentioned and maybe explain this. Firstly, there is technology issue. Smalltalk, as all other virtual machine based platforms is memory consuming. The time when Smalltalk appeared was the time when computers were not as powerful and the first versions of Smalltalk were rather demanding, slow and difficult to use. Java on the other hand has come just in time when technology started to boost its power. It is much more freedom giving that Smalltalk. Java can be used in many different environments and this provides the developers to have their way and to choose what they want. The syntax of Java is similar to C and some other languages so it was easier to recognize it as something similar to what programmers have already known. Secondly, like many other functional and logic programming languages, Smalltalk has very poor I/O functionality. This disadvantage makes it hard to use for programs that are based on interaction. However, the greatest disadvantage of Smalltalk is the big gap between Smalltalk and the rest of the world. One of the main goals of Smalltalk community is to save Smalltalk pure. This means they do not want to make any steps toward the concepts most of the rest programming languages and platforms consider as standard – functions, libraries, etc. There is no way to include any C or C++ library into Smalltalk program. There is also no way to define or use any function in it, therefore, there is no way to use, for example, API functions in Smalltalk, no way to use or make DLL's. In fact, there is extremely small possibility to join Smalltalk with anything else than Smalltalk itself.

Regardless all of this, Smalltalk has made a huge impact on many modern programming languages, whose developers recognized advantages of the concepts Smalltalk promotes, but were not so autistic and single minded as Smalltalk community is. Today, there is a very small chance for Smalltalk to be resurrected, because many languages took all good from it and incorporated it in its more flexible and user-friendly environment.

3.2.3 ML

ML is functional programming language with "Pascal-like" syntax and with procedural elements. It supports functional programming (through e.g. anonymous first-class functions) and imperative programming (through functions with side-effect and reference types). It was originally designed in 1973 by Robin Milner and associates at the University of Edinburgh as the meta-language for a program verification system which was called Logic for Computable Functions (LCF). ML is mostly well-known for its use of the Hindley-Milner type inference algorithm, which can infer

almost all types without annotation. Furthermore, ML was the first programming language, which has used type inference as a semantically integrated component [40]. Although it is strongly typed, most of the time there is no need for the programmer to write any type-declarations. Type of both parameter and return value can be determined from function definition (e.g. if we use arithmetic operator, ML can assume that both function and parameter will be numeric) and that is why we can say that ML is much more compact and readable than programming languages which require explicit type declarations. Unlike Scheme and LISP, ML does not use parenthesized syntax of functional languages but syntax which is very similar to imperative languages like Java or C++ (e.g. arithmetic expressions are written by using infix notation). ML programs are exceptionally concise, with a friendly syntax, easier to reason about compared to imperative counterparts and offer interesting opportunities for parallel execution. Some of ML's main features are call-by-value evolution strategy, parametric polymorphism, static typing, pattern matching, exception handling and "eager evaluation" of all sub-expressions. ML has type constructors (tuples, records and lists) that are type operators which can be applied to any value in data structure. ML also supports module facility for abstract data type implementation and allows a programmer to define his own types and type constructors (e.g. disjoint unions, enumerated types, etc.) [64]. General form of function declaration in ML looks like this:

```
fun function_name (parameters) = expression;
```

Furthermore, if we want to choose an expression that defines return value of function or to access components of aggregate data structures, we can rely on pattern matching [77]. In program examples below, there is a code of the same function for factorial calculation. The first program shows how function can be written without pattern matching. In second program, pattern matching and different function definitions that are separated with OR symbol (|) are used.

Program 19

```
fun max(h::t): int =
    if t = [] then h
    else if h>max(t) then h
         else max(t);
```

Several instances of ML language exist today: Standard ML (SML), CAML, F# and others. Standard ML is a type-safe, modular, strict, multi-paradigm (functional and imperative), polymorphic descendant of the ML programming language. It was proposed in 1983 and defined in Definition of Standard ML [62]. Seven years later a modest revision and simplification of the language has been released and defined in The Definition of Standard ML (Revised) [63]. SML is a statically typed language with an extensible type system that provides efficient automatic storage management for data structures and functions. There are several characteristics of SML programming language [67]. It supports functions definition by pattern matching, supporting programming with recursive and symbolic data structures. It maintains innovative module system for large applications structuring; provides mechanism for building generic modules, enforcing abstraction and imposing hierarchical structure. And, at the end, it has versatile and modular error exception mechanism.

The criticism against ML goes in two directions. The first is the argument that does not target ML exclusively, but functional programming. Functional programming is founded on general, pre-defined algorithms based on tree searching. Because of that, functional languages will never be able to compete procedural languages in speed and code optimality. The same type of criticism goes to complexity of functional thinking and programming, which completely differs from

standard, procedural one. Functional programmers, on the other hand, strongly oppose ML's procedural features and side-effects in functions, and call ML impure functional language.

Today, ML is mostly used in language design and manipulation, but it is also applied in a field of theorem proving (HOL), genealogical databases, financial systems, bio-informatics, peer-to-peer client/server programs, etc.

3.2.4 SQL

In the late 1960's and early 1970's E. F. Codd (researcher at IBM) came up with the idea of relational data model, which he presented in his paper [20]. Along with this model he proposed a language called DSL/Alpha that would be suitable for data manipulation. Based on his proposal, IBM engaged in development of this kind of language. It was first called SQUARE, then SEQUEL and finally SQL [9] in 1979. The first version of SQL was developed by Donald D. Chamberlin and Raymond F. Boyce and it was meant for working with System R – IMB's relational database.

The language was developed as a support for RDBMS programming, and it is not, as its name would suggest, only a query language, but it is a complete language for database programming and manipulation. SQL enclose all three basic languages for relational databases – query language (QL), data manipulation language (DML), and data definition language (DDL). The language is highly functional, excluding any procedural properties. On the other hand, language is not general-purpose, and it works only inside of RDBMS. Therefore, it cannot be compared to other general-purpose languages, like C or Pascal. In its early days the only programming language that it could be compared to was COBOL, but the concepts COBOL and SQL presented were completely different. Hence, it could be said that SQL was unique language that brought completely new concept of database programming.

Although the syntax of all SQL commands follow the same syntax rules, the proof of the claim that it contains three different languages in one becomes obvious from the way commands are executed. While DML commands are transactional, DDL commands are not, but are, in one sense, imperative, while QL commands lay on the concept of snapshots to achieve their consistency. QL commands are interpreted through the concept of query plans, while DML (at least the simple ones) and DDL commands are not. Only DML commands can lead to the problems of deadlocking and live locking and that is the reason why transaction system is introduced. Although there is some kind of transaction system and locking system for QL commands too, this transaction system is much more simple than the one that is implemented for DML commands, and, in fact, it is an additional system for snapshotting.

The following properties can be considered as main properties of language:

- It is name oriented – every data and object in the database is available through its name (while Prolog is, for example, position oriented)
- It is tuple oriented – all data are organized into tables containing tuples of data. In the tuple every attribute of the table has a single value (NULL value is allowed for some data)
- It is functional – there is no way to say how to execute the query (The querying algorithm for a query is chosen by database engine. The user only defines which data have to be retrieved).
- It is typed – every data in the system has its own defined type, and cannot contain a value of any other type.
- It is multi-set oriented - although the theory of relational databases is developed on the relations (tables) that have the structure of a set, SQL and database systems developed around it allow multiple tuples with the same values for all attributes.

The language syntax was developed around a few commands that have variety of optional clauses that give them a great flexibility. QL contains only one command – SELECT, which covers all query capabilities that are needed.

Program 20

```
SELECT avg(salary)
FROM employees
WHERE old > 30
AND sec='F'
HAVING avg(salary) > 3000;
```

DML part of language contains three commands – INSERT, UPDATE and DELETE, which execute three different operations that change data in the database.

Program 21

```
INSERT INTO salary (number, ammount)
VALUES (10, 3500);

UPDATE salary
SET amount = 3500
WHERE amount BETWEEN 3300 and 3499;

DELETE FROM SALARY
WHERE amount < 3000;
```

DDL contains three commands - CREATE, ALTER and DROP that deal with creating, altering and erasing of database objects (tables, indexes, sequences, triggers etc) and two more commands for defining user privileges – GRANT and REVOKE.

To summarize, the whole language is built on 9 basic commands, yet it has large capabilities.

The problem is that SQL is highly dependable on a database system it works in, so every producer of RDBMS developed its own dialect of SQL that only partially follows the SQL standard, but in the other part enables some special properties and concepts of the RDBMS.

SQL as a functional language has a serious drawback in programming of functions that are strictly procedural. That is the reason that SQL is often incorporated into some procedural language. Many RDBM systems have interfaces and libraries that allow programmers to use SQL statements in standard procedural programming languages such as C, C++, FORTRAN. Perl and so on, but besides that, there are procedural languages that are especially developed to support SQL in procedural programming, such as PL/SQL and its variants (PL/pgSQL etc).

Although SQL is momentary the only language that is widely used in database systems (particularly in relational database systems), it is not all-accepted and followed language. There have been serious objections to SQL for some time now. Theoreticians object its differences from the theory: multi-set orientation, inconsistent treatment of NULL values. Puritan followers of functional programming object introduction of triggers as a procedural concept into the language. Programmers often object the complexity of clauses that often makes commands in program more complex that it could be with some different approach (logic programming, for example). And, at the end, writers of standards for the language object abandoning the concepts that are defined in standard and are implemented differently in database systems (for example, by the standard, functions of triggers should be written in SQL, which is not the case in any modern RDBMS).

As the database language, that becomes extremely important, standardization of the language is highly developed. The first SQL standard, defined by ANSI/ISO was published in 1986. It contained the basics of the language, but it was not complete and could not follow the development of the language. The next standard was SQL92 standard [21]. This standard had a great impact to the development of the language, introducing schema, triggers, roles and other active database components, referential integrity definitions and queries that uses it for connecting data from different tables. We can say that SQL92 standard was a foundation for the SQL language as it is today. It is true that no RDBMS ever adopted it in whole, but all of them were trying to do so, and it had a great impact to the adoption of common basic properties of the language, and, by that, the much greater portability of database and SQL code. The next, SQL 99 standard, did not add any significant features into the language. It slightly modernized the standard from 1992 and added commands for collations, character sets and sessions. In short, it was a revision of main standard from 1992. The last SQL standard is one from 2003. This standard modernized the language by adding XML into SQL language. The command MERGE is added that merges two multi-sets of data, contextually doing inserts or updates, as well as window functions – functions that simplify aggregates (HAVING clauses) in SQL statements. Some features that were implemented in various systems for a long time, such as counters and creating tables using schema or data of other table, were also standardized. The new standard still does not include all extensions of SQL language that are available at the market. For example, OSQL (object extension of SQL language) is not standardized. So, we can expect further SQL standards, as there are no indications that SQL could be replaced with any other language as a leading language for database programming in near future.

3.2.5 Ada

Ada is a structured, statically typed (the data type of every variable, parameter and function return value is known at compile time), imperative (a programming paradigm that describes computation as statements that change a program state) and object-oriented high-level computer programming language [32], [7], [73].

1970's were the time when the US DoD (Department of Defense) was concerned about a large number of different programming languages that were used for development of their computer system projects. Many of these languages were hardware-dependent and none of them supported modular programming. To solve this problem, HOLWG (Higher Order Language Working Group) was formed in 1975 and the result of their work was Ada.

It was recognized that no existing programming language is suitable for required purpose. Because of this conclusion, four contractors were found to make proposals about solving this problem. The names of proposals were Red (Intermetrics), Green (CII Honeywell Bull), Blue (SofTech) and Yellow (SRI International). In the end the Green proposal was chosen and given the name Ada after Augusta Ada, Countess of Lovelace who is recognized by historians as an author of the first ever computer program [88].

So called Ada mandate started in 1987 and it meant that the use of Ada is required in every project that includes software development and has more than 30% of new code. This requirement was removed in 1997 when DoD started to use COTS technology.

Ada syntax is very simple and readable, mostly adopted from ALGOL and Pascal. It also adopted strong types in the way they were implemented in Pascal, as well as nested procedures, most of basic data types, string representation and many other features. It minimizes the possible ways to perform the basic operations, it prefers english words (and) over the symbols (&&). There are basic mathematical symbols present in Ada (i.e.: "+", "-", "*" and "/"), but usage of other symbols is generally avoided. Blocks of code are packed between the following keywords: declare, begin, end. The Pascal syntax was improved in several ways. Firstly, the readability of

program is improved by adding the type of command at the BEGIN-END block. In Ada every loop block ends with END LOOP, while every IF-ELSE block ends with END IF. The new loop is added – the loop with the exit in the middle. This kind of loop has the following syntax:

```
loop
.
.
.
exit when <logical condition>
.
.
.
end loop;
```

Program 22

```
with Ada.Text_Io; use Ada.Text_Io;

 procedure Max is
    type Int_Array is array (Positive range <>) of Integer;
    package Int_IO is new Ada.Text_IO.Integer_IO(short);
    Item Int_Array;

    function Max(Item : Int_Array) return Integer is
       Max : Integer;
    begin
         Max := Item(1);
         for J in 2..Item'Last loop
             if Max < Item(J) then
                Max := Item(J);
             end if;
         end loop;
         return Max;
    end Max;

    for I in 1..10 loop
        Ada.Text_IO.Integer_IO.Get(Item(I));
    end loop;
    B : Integer := Max(A);
 end Max;
```

Ada has become an ANSI standard in 1983 under the name ANSI/MIL-STD 1815A and as such became also an ISO standard in 1987 under the name ISO-8652:1987. The version of Ada that was actual here is Ada 83, or sometimes also called Ada 87. The numbers in the names come from the date of Ada standard adoption by ANSI and ISO. Ada 95, the joint ISO/ANSI standard ISO-8652:1995 is the latest Ada standard. This standard was released in 1995 and made Ada the first ISO standard object-oriented programming language.

Ada was influenced by many other languages such as ALGOL 68, Pascal, C++ (influenced by Ada 95), Smalltalk (influenced by Ada 95) and Java (influenced by Ada 2005) but it also influenced many programming languages such as C++, Eiffel, PL/SQL and VHDL. Ada today

remains the model for comparison for all other languages in the dimension of safety, security, multi-threading, and real-time control. The number of programmers working in Ada has shrunk over the years but there are still those who work in this languages, especially in the high-integrity niche [36].

The main problem with Ada was that it has never been adopted from a wider community of programmers. When Ada arrived, there was a war between two strong languages – C and Pascal – going on. Both of these languages had its followers that did not want to consider any other programming language. Ada was much closer to Pascal community, but it neither gave a great improvement regarding Pascal, nor it had all programming features, which in that time well developed programming languages had. Ada is the procedural language with almost perfect syntax, but Pascal was not far behind it. The only feature that is discussible is loops with exit in the middle, the concept that was adopted from C, and that implicitly introduces jumps into programs. These are the reasons why Ada stayed a language loved by everyone and used by no one.

3.3 The Late Medieval: The C Empire

After C++ was published, C-like languages became de facto standard in procedural programming. There was no other significant procedural language that was based on different syntactical concept. Pascal-like languages lost their stand. It was, in a way, the loss of theoretically based languages against pragmatic ones. Theoretical aspects of programming languages were moved to functional programming and logic programming.

3.3.1 C++

C++ is general-purpose programming language developed by Bjarne Stroustrup at Bell Labs in 1979 as an improvement of C programming language. C++ is an object-oriented programming (OOP) language that is viewed by many as the best language for creating large-scale applications. Initially C++ was called "C with Classes" but its name was changed to C++ in 1983 (Rick Mascitti takes the credit for the name C++) [81]. Among many enhancements there were: classes, virtual functions, operator overloading, multiple inheritance, templates and exceptions handling.

Stroustrup started his work on C with Classes in 1979 and the idea for this work came from his experience with programming languages that he gathered during creation of his Ph.D. thesis. He has chosen C because it is general-purpose, fast and widely used. In his work he was inspired by many other programming languages such as: Simula, BCPL, ALGOL 68, Ada, CLU and ML [81]. The first commercial version of C++ was released in October 1985.

Stroustrup started enhancements with adding classes to C and that is the reason for the name C with Classes. When name was changed in 1983 new features were added: virtual functions, function name and operator overloading, references, constants, user-controlled free-store memory control, improved type checking and BCPL style single-line comments with two forward slashes (//). In 1985, the first reference to C++ was released under the name of The C++ Programming Language. There was no official standard at this time. New features were added to C++ in 1989 (C++ Release 2.0): multiple inheritance, abstract classes, static member functions, const member functions and protected members. In 1990, The Annotated C++ Reference Manual was published and it has become the basis for future standard. Features that were added to C++ at a later time include templates, exceptions, namespaces, new casts, and a Boolean type.

Program 23

```
#include <stdio>
using namespace std;

int main() {
    int a[10], i, max;
    for(i=0; i<10; i++)
       scanf("%d",&(a[i]));
     max=a[1];
     for(i=1; i<10; i++)
         if (a[i]>max)
            max=a[i];
     cout << max << endl;
     return 0;
}
```

Along with new features of C++, a standard library has also been developed. The first version of C++ standard library [83] included I/O streams that replaced C functions such as printf and scanf. Later, among the most significant additions to the standard library was the Standard Template Library, which provides containers, iterators, algorithms, and functors. A joint ANSI-ISO committee standardized C++ in 1998 under the name of ISO/IEC 14882:1998 [80]. Corrected version of the standard was released in 2003. Although not part of official standard, in 2005 a technical report called Library Technical Report 1 (TR1 for short) was released and it a set of new features that are planned as has a part of new C++ version standard. TR1 features are supported in almost all known C++ compilers.

In one of his books Bjarne Stroustrup states some guiding principles that he used when designing C++ [81]:

- C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C
- C++ is designed to directly and comprehensively support multiple programming styles (procedural programming, data abstraction, object-oriented programming, and generic programming)
- C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly
- C++ is designed to be as compatible with C as possible, therefore providing a smooth transition from C
- C++ avoids features that are platform specific or not general purpose
- C++ does not incur overhead for features that are not used (the ”zero-overhead principle”)
- C++ is designed to function without a sophisticated programming environment

C++ was designed to support many programming styles and Bjarne itself stresses this as one of its major advantages [82]. The styles that are supported include [82]: traditional C style, concrete classes, abstract classes, traditional class hierarchies, abstract classes and class hierarchies, and generic programming.

In 1998, C++ was standardized through a standard that was brought by joint ANSI-ISO committee. The standard was called ISO/IEC 14882:1998. This version of the standard was updated in 2003 and new standard was called ISO/IEC 14882:2003. The standards mentioned were also known by their shorter names: C++98 and C++03. The standard that is prepared currently and follows previously mentioned standards is called C++0x. There are also other standards that are present regarding C++ that consider some additional aspects of C++ language.

These standards are:

- JTC1.22.18015 ISO/IEC TR 18015 – Tehnical report on C++ Performance
- JTC1.22.19768 ISO/IEC TR 19768 - C++ Library – TR1 (formaly only draft version, but support for it is present and increasing)
- JTC1.22.24733 - ISO/IEC TR 24733 - Information Technology - Programming Languages C++ - Technical Report of Type 2 on Extensions for the programming language C++ to support decimal floating point arithmetic (draft version)

C++ is best when talking about general purpose and low level programming and it is very suitable for beginners who want to learn basic programming skills, because it consists of all necessary elements that one should know to be able to understand how to program using object-oriented paradigm or older structural paradigm. In addition, many languages that can be found on the market today have C-like syntax so by learning C++ one makes it much easier for himself to learn a series of other languages such as Javascript, PHP, Java, ActionScript, etc.

Advantage of C++ is that it is rather fast (although very slow when compared with C), especially in computation and memory manipulation so it is very suitable for various low level purposes such as image processing, for GUI programming, programming of various operating system functions, drivers and platform dependent applications, video games, and for various engineering purposes.

Main disadvantages of C++ are that, although it is in its nature platform independent, it is mostly used for applications that are in some way platform dependent and it is usually necessary to change program code quite a bit in order to move some particular C++ program to another platform. The thing that usually connects particular C++ program to particular platform or operating system is a chosen library set. It is quite complex when developing large scale high level programs and it is very difficult to use and debug if used for web development. The majority of C++ syntax and semantics was adopted from C, adopting many of its disadvantages with it: pure readability, syntax that in some cases opposes theory of languages and often is not easy to learn and understand, C strings that often lead to serious problems, unsafe pointers etc.

To conclude, C++ is a powerful language that has its distinct purpose in any serious application that demands a little bit of lower programming. Therefore, it is the one of the first choices for programming of any complex problem that requires speed and precise definition of algorithm. C++ is still one of the three most used programming languages today.

When we talking about this period, maybe the most important think that happened was development of graphic-oriented operating systems graphic user interfaces (GUI), such as X Windows (1984), Microsoft Windows (1985) and MacOS (1984). The new orientation in operating system design had a great impact to the development of the new paradigm of programming languages - event driven programming, the programming that was not based on the sequential execution of the code, but on the execution of the code as an answer to some event that happened in the user interface (mouse click, entering some field in the mask on the screen, etc). In the first, classical programming languages, such as Turbo Pascal, Turbo C/C++ and Microsoft C++, were used to program GUI, and that required a great deal of code that were used to control windows, and objects in them, define properties of objects, and other things that were not oriented to the problem

solving, but to the interaction with the GUI and operating system. The programs became clumsy and large, containing much of the code that many programmers did not even try to understand, but they copy-paste it from one program to another. It was obvious that the new platforms for visual and event driven programming were a great need. The answer first came from Microsoft Inc. and Borland Inc. very soon after it.

3.3.2 Eiffel

Eiffel is an object-oriented language developed in 1986 by Bertrand Meyer and associates at the Société des Outils du Logiciel ŕ Paris. Eiffel language is named after Gustave Eiffel, the engineer who designed the Eiffel Tower. The developers of Eiffel like to compare their language to the well-built structure of the Eiffel Tower; they tend to say that if you use the language for your product development, the project will be completed on time and within allocated budget. The language is based on the principles of object-oriented design, augmented by features enhancing correctness, extendibility and efficiency; the environment includes a basic class library and tools for such tasks as automatic configuration management, documentation and debugging. Furthermore, it contains all object-oriented concepts like information hiding, polymorphism and dynamic bindings. A combination of reusable and flexible methods is one of the main ideas that Eiffel tries to promote. It also offers multiple class inheritance, which until the creation of some straight-forward language rules became a useful programming technique. Eiffel includes only a tiny amount of keywords to keep the source code compact and well organized. In other words, the same notation can be used in specification, design and realization. An interesting feature is its ability to enable C source code generation, allowing compilation and usability of its applications on multiple platforms.

Program 24

```
class
     SEARCH

create
      make

feature

       make is
            local
                 arr: ARRAY [INTEGER]
                 gen: GENERATOR
                 j: INTEGER
                 largest : INTEGER
       do
            create gen
            arr := gen.random_array (10)
            from
                j := arr.lower - 1
                largest := largest.Min_value
            variant
                   arr.upper - j
            until
                 j = arr.upper
```

```
            loop
                j := j + 1
                if arr.item (j) > largest then
                   largest := arr.item (j)
                end
            end
            io.put_integer (largest)
            io.put_new_line
        end
end
```

The most significant contribution of Eiffel to the world of software engineering is DbC (Design by Contract), which is a methodology for designing computer software. In these methodology assertions, preconditions, post conditions and class invariants are used to assist in assuring program correctness without sacrificing efficiency [13]. Eiffel has only six basic executable statements: assignment, choice, conditional, iteration, method call and object creation [74]. The Eiffel language emphasizes declarative statements over procedural code, eliminates the need for bookkeeping instructions and thus promotes clear and elegant coding. Furthermore, Eiffel is not case-sensitive and therefore identifiers and keywords can be written in any combination of upper and lower case. The first international standard for Eiffel was approved in June 2005 and its full title is ”ECMA standard 367, Eiffel Analysis, Design and Implementation Language”. However, standard was not accepted by the whole Eiffel-community because some believed that the basic principles of the original language were put aside in the standard. The second edition was published in June 2006 and in the same month it was adopted by ISO (published in November 2006) [27]. Today, Eiffel is used for applications around the world in the fields of health care, networking and telecommunications, banking and finance, game programming and computer-aided design. Finally, Eiffel programming language is popular for software engineering and teaching programming in universities.

3.3.3 Perl

Perl is a general purpose scripting language designed by Larry Wall in 1987 as an effort to produce some reports for a bug-reporting system [68]. ”Perl” was originally named "Pearl" after the ”Parable of the Pearl” from the ”Gospel of Matthew”. ”Pearl” can also be a symbol of beauty, richness and elegance. No matter how its name is interpreted, either by reading the famous ”Pearl” legend or on ones own, it has very positive implications. ”Moments” before its official release 1987 in the ”a” in ”Pearl’ was dropped and the language has since been called ”Perl,” later dubbed the Practical Extraction and Report Language, and by some, it is referred to as the Pathologically Eclectic Rubbish Lister [70]. On the other hand, regardless Perl is a scripting language, it is complete language with all languages elements, control structures, subroutines etc.

Perl was originally written to manipulate text in files, extract data from files, and write reports, but through continued development, it can manipulate processes, perform networking tasks, process Web pages, talk to databases, and analyze scientific data and do many other things, provided that bindings to various libraries are installed. As an interpreted object orientated language, Perl has many advantages. Firstly, it treats data like objects where every object has its own behaviors and properties. By using this advantage, it is easier to re-use parts of code or even expand the program at a later date. In order to provide extra functionalities, Perl allows implementation of extra modules (e.g. there is a module which adds FTP functions to Perl code, etc). Secondly, it is possible to develop self modifying programs that add or change functions while the code is running.

Data types in Perl are rather different from other languages. We can say that Perl is partially untyped language, although it recognizes four data types. This puts Perl into very high level programming languages that are more oriented to the flexibility and easiness of than on speed and optimality of the code. The variables in Perl are not, like in most of other languages declared to the certain type. The data type of the variable is defined with the special character at the beginning of variable identifier ($ for scalars, @ for arrays, % for hashes and none for filehandlers). This property of Perl reminds to early versions of BASIC or to Clarion programming languages and is mainly abandoned because it was found to be bad for the readability of the program and hard to maintain in larger programs.

The first Perl data type is *scalar*. Scalar data type contains all data types that are considered as element data types in other languages – numbers (integers and floating points), strings and even references (pointers). As Perl is primary created for file handling, it is not a surprise that it has a wide set of functions and operations on strings. Strings can be quoted with double quotas, single quota and they can even be defined as a larger part of the text that is terminated with a chosen symbol. Pointers in Perl worked the Pascal way, the way that is much more benign and safe than C/C++ ”hard core” pointers, called references in Perl. Many of modern languages (such as C#) adopted Pascal style of pointer definition, so it cannot be added to drawbacks of Perl. The thing that must be pointed out is that scalar, as well as other data types that are created from scalars in Perl are not fixed length. The length in the interpretation of the scalar variable is contextual, depending on what is written in the variable at that time. This gives a great flexibility in programming, but it is also a great attack on optimality of the code and the speed of the program.

The second data type is array, a row of scalars. Although the data type is called array, it is much more flexible than standard arrays. Its length can be changed, it can be accessed as an array or as a list, or even as a stack. Arrays can be concatenated, split and so on. Perl fans point all these properties as advantages of Perl, but one should not be so excited with this. Namely, Perl in fact does not make arrays but lists, and much of the access speed that are arrays famous for is lost. It is true that Perl lists are much more flexible than arrays in C or Pascal, but this comes as a trade-off with the speed.

The third data type cannot be found in other programming languages. It is associative array or hash. Hashes in a way resemble hash-tables, in which one or more primary values are associated with one (often much shorter) key value. This data type is good for working with grouped data.

The last data type are filehandles. Filehandles look and work like C++ streams and makes working with files similar to C++.

Furthermore, by using a variety of in-built functions (e.g. `substr` for extracting any part of string or chop for removing the last character of a string) Perl facilitates strings manipulation.

Program 25

```
#!/usr/bin/perl
use strict;
use warnings;
$number = <STDIN>;
@numbers = split(" ", $number);
$max = find_max(@numbers);
print "$max";

sub find_max{
    @numbers = @_;
    $max = $numbers[0];
    foreach $elt (@numbers){
```

```
            if ($elt > $max){
               $max = $elt;
            }
    }
    return $max;
}
```

In order to execute, every Perl program has to be passed through Perl executable that is Perl interpreter. Perl interpreter is usually installed in usr/bin on local machine and almost every Perl program starts with line #!/usr/bin/perl. In UNIX systems, this line tells the shell to look for the Perl interpreter and pass the rest of the file to it for execution [78]. The second and the third line contain something that is called pragma. Pragmas are special modules that tell Perl's internal compiler how to understand the code that follows. There are several pragmas which come with every release of Perl, but two most often used are strict and warnings pragma. Strict pragma, which is lexically scoped, enforces rules for compiling the code and thus helps find errors in code more easily. On the other hand, warnings pragma controls run-time behavior and prints warning messages when certain operations seem to be questionable. In Perl there is no difference between functions which return value and procedures which do not; reusability of code is achieved by user-defined functions popularly called subroutines. Therefore, functions in Perl are routines that are built into Perl while subroutines are chunks of program written by programmer.

Perl is the language that has many high level properties that makes life of a programmer easier, such as matching, various operations with strings and regular expression handling. But in the try to give many good properties, as its opponents say, Perl became large, slow and unreadable language, that does not care for the code optimality and speed. The code that is often given as a proof of it is the code that is published by some Perl fan to show what he can do with only a few Perl lines of code.

Program 26

```
while (<>){
      while ( /(.*?<!--)(.[^-]*)(.*$)/){
          print $2."\n";
          $_=$3;
      }
}
```

The next observation that is often added to the language drawbacks is that some built-in functions have properties that opposes to Perl capabilities, and there is no possibility to write them in language. So, it is obvious that Perl libraries are actually not written in Perl. n short, the problem with Perl is that it is not a reflexive language.

Perl has been through a number of revisions and changes. Right now, two major version of Perl exist: Perl 4 and Perl 5. The last version of Perl 4 was Perl 4, patch level 36 (Perl 4.036), released in 1992, making it ancient. Perl 5.000, introduced in fall 1994, was a complete rewrite of the Perl source code that optimized the language and introduced objects and many other features. Despite these changes, Perl 5 remains highly compatible with the previous releases. At the time of writing this paper, the current version of Perl is 5.10. Perl 6 will the next generation of this language, and will come with not even approximate release date. It will have new features, but the basic language everyone grew to like or hate will still be there.

3.3.4 Haskell

Haskell is one of those languages designed by committee, and is a purely functional language. The first version of Haskell was defined in 1990. Language got its name after Haskell B. Curry, who was one of the first pioneers of lambda calculus, a mathematical theory of functions. He inspired a lot of functional language designers. Some of Haskell's features are really fascinating, whereas the value of a certain data piece does not have to be known until the data actually has to be used. What does this mean? It means that a programmer can define things in terms of infinite series, such as the set of all integers. Pugs, implementation of Perl 6, are implemented in Haskell, and many think that they provide a great marketing platform for Haskell. It features interactive console similar to Ruby's "irb", but unlike its Ruby counterpart, it does not allow function definitions, aside from those imported from the external files. Similar to C, Haskell supports both strong and static typing. As such, Haskell is very suitable for teaching purely functional programming techniques, but it is not limited only to it. Its features include curried functions, higher-order functions, non-strict semantics, static polymorphic typing, list comprehensions, modules, monadic I/0, and layout-based syntactic grouping.

Haskell differs from ML in several ways. Firstly, Haskell uses another type of evaluation technique, which is called the lazy evaluation. The lazy evaluation means that during a program execution only used parameters will be evaluated. In other words, there is a possibility that in single case only one parameter or even its fragment will be evaluated, while everything else will stay unevaluated, which is good for e.g. definition of infinite lists [86]. However, flexibility which lazy evaluation brings to Haskell has its price in slower program execution and more complex semantics. Secondly, Haskell supports lists comprehensions for sets definition, very similar to mathematical notation [42]. In Haskell there is no need to use special keyword for function definition and there is also no need for brackets in formal parameters delimitation. Unlike most of the other functional languages, and like most of logical languages, it is possible to define alternatives of a function by several independent statements while in LISP this is not possible, in ML a programmer has to do that separating definition cases with | operator. Similar to Scheme, Haskell supports recursive and inductive lists definition [50].

Program 27

```
maxi :: [Int] -> Int

maxi[x] = x

maxi(h:t) = if h > maxi(t) then h
                           else maxi(t)
```

While there are many Haskell implementations available, most known one is Hugs system, which is an interpreter. Speaking roughly, it means that it evaluates expressions step-by-step, and therefore will be less efficient then a compiler which translates Haskell application into machine code. Result of compiling a language like Haskell allows its programs to run with a new C or C++ speeds. Since its original publication, Haskell was evolving regularly and in the middle of 1997 there had been four iterations of the language design. In the same year, at the Haskell Workshop in Amsterdam it was concluded that a stable variant of Haskell was needed. One year later, a new version of language called Haskell 98 has been released. In 2003, the committee published "The Haskell report", announcing a long-awaited stable version of Haskell, which is the culmination of fifteen years of work on the language by its designers.

Haskell is pure, almost perfect functional programming language. There is no serious criticism against Haskell, except one that targets functional programming as a paradigm.

4. The Modern Age: Visual Programming and Reflexive Languages

4.1 The Modern History of the World: Modern Programming Languages

4.1.1 Visual Basic

Although BASIC programming language was described above, and Visual Basic partially with it, the truth is that Visual Basic is much more different than BASIC as, for example, C++ is from C, and for that reason it deserves to be treated as a different language.

In the year 1991 Microsoft Inc. finally answered to the great need of programmers for the tool that would make their lives in the world of GUI's easier. They presented at the COMDEX the project named Visual Basic for Microsoft Windows 3.0, which automatized the GUI definition and communication to the operating system through the system of events. The system was based on Microsoft's previous project called QuickBasic as a underlying programming language. The interesting thing was that the first Visual Basic system was a complete compiler and that programs developed with it were totally independent on the VB system. That was changed very soon. The language became (and still is) in its basis interpreter, with the executable object code.

Although the language is based on BASIC programming language, Visual basic started to develop extremely fast and independently from its parent language and very soon became much different and much well developed than BASIC itself. At the first time Microsoft tried to develop QuickBasic together with Visual Basic, so in 1992 they published Visual Basic for DOS, that was, in fact, QuickBasic with event driven programming properties, but that concept was abandoned very soon.

The main development of the Visual Basic has two main tracks: the development of the language and the development of the communication to the operating system. The language was over the years changed dramatically. Explicit data typing was added, so in Visual Basic every variable has to be declared. New data types were added in accordance to the modern data typing (long integers, bytes, double precision floating point numbers), but also many of data types were added that were connected to the Windows objects. Functions and subroutines were added, too. In the version 3, the Jet Engine support was added for easy database programming. In that time MS Access rapid application developer (RAD) was created for rapid development of small database applications. This system was based on the dialect of the Visual Basic called Access Basic, and was even more than Visual Basic adopted to the database programming. On the other hand, Visual Basic very soon (in the version 2) started to treat operating system entities as objects and to communicate with them through standard communication channels. Regardless that, Visual Basic never became an object-oriented language, because it did not allow programmer to create his own objects and classes. In version 4 OLE objects were added to enable easier manipulation with the multimedia data. In the next version ActiveX controls were added to allow usage and programming of component object model (COM) applications. This feature allows easy programming of components that can be added to other applications, such as Microsoft Internet Explorer. In Visual Basic 5 Microsoft tried to nullify one of the main objections to Visual Basic system – they made a step toward to possibility to make totally independent programs with Visual Basic. In the newest version, great deal for changes were made to adopt Visual Basic to .NET system [79]. This changes were the reason of many angry complaints to Microsoft, because they do not assure any backward compatibility. Besides Visual Basic system, Visual Basic as the language is incorporated into many other Microsoft products. The Microsoft Access RAD and Access Basic are already mentioned. Visual Basic is also incorporated into Microsoft Office for programming Macro objects as Visual Basic for Applications, providing scripting version of Visual Basic. Another version of a scripting language based on Visual Basic is VBScript language, the main default language for the Active Server Pages (ASP). This version of Visual Basic will be described in one of the following sections of this paper.

Program 28

```
Sub Main(A(10) as Integer)
    Dim I As Integer, M As Integer
    M=A(1)
    For I=2 To 10
        If A(I)>M Then
           M=A(I)
        End If
    Next I
    MsgBox M
End Sub
```

The main, and almost the only producer dealing with Visual Basic is Microsoft Inc. No other producer has serious system based on BASIC or Visual Basic. That is the reason why standardization of Visual Basic was never interesting to ISO/ANSI, unfortunately, and whatever Microsoft brings in each new version becomes de facto standard.

There are several main points of complaining to Visual Basic system that, somehow, Microsoft cannot or does not want to solve. The first great problem with it is the great lack of backward compatibility. Very often a new version of Visual Basic brings new concepts that are not compatible with previous ones, and to much often Microsoft just does not care to assure backward compatibility for programs developed in the previous version. The second problem that is often mentioned is the problem of the speed. The Visual Basic is still in its basis interpreter that ”compile” programs into object code that has to be interpreted during the execution. The problem is that this property can be a bottleneck in situations when the speed is of critical importance. The next drawback is connected with the previous one: the problem that programs made in Visual Basic cannot be distributed independently. They need some kind of interpreter (VBxxx.dll) to run. This is maybe a minor problem, because they work only on MS Windows platform, but the problem of complete dependence on MS Windows platform will never be solved. At the end, let us mention the problem that is also mentioned against Visual basic very often – the problem of the programming language, which is based on BASIC, and inherits and still were some of its illogic. Regardless these drawbacks, Visual Basic still remains the most simple, easy to learn system for event driven programming and rapid application development in the Microsoft Windows platform.

4.1.2 Python

Python is a high-level programming language created by Guido van Rossum at CWI (The National Research Institute for Mathematics and Computer Science) in late 1980s as the scripting language for the Amoeba operating system and published in 1991 (version 0.9.0). Because Guido was a big fan of the British comedy troupe Monty Python, it is not a problem to guess where from the language got its name. The biggest inspiration to Python was ABC programming language, with some of the features and abilities coming from other languages like C and Modula-3. Starting from 1991, Python is collaboratively developed under a FOSS license by developers world-wide. Python has its own philosophy which emphasizes readability and simplicity. Python syntax and semantics is minimalistic, while the standard library is very large. Unlike many other languages, none of the Python constructs needs to have an explicit end-of-line mark (often a semicolon). Python explicitly embraces the idea of ”There should be one—and preferably only one—obvious way to do it”, reporting this along with other ideas at Python’s interactive prompt when given the command ”import this”. It was not until January 1991 that Python reached version 1.0, which included functional programming tools lambda, map, filter and reduce. The last version of Python

released from CWI was Python 1.2. Several subsequent versions of Python were made while Rossum was working at CNRI (Corporation for National Research Initiatives) in Reston, Virginia, USA. Version 1.4 brought substantial improvements: Modula-3 inspired arguments, built-in support for complex numbers, etc. While working at CNRI, van Rossum launched CP4E (Computer Programming for Everybody) initiative that strived to make programming easily understandable to everyone. Since Python was simple and effective, it was the language of choice for the initiative. CP4E was closed in 2007. Back in 2000, Python core dev team was transferred to BeOpen.com. CNRI release version 1.6 summarizing everything done until the point dev team left. Mostly because of that, releases 1.6 and 2.0 (the only releases from BeOpen.com) had a lot of overlaps. Afterwards, Python development team joined Digital Creations. At the moment of writing this writing, most recent stable version is 2.6.

Program 29

```
for i in [0..10]:
     a[i] = input("")

max = a[0]

for i in [1..10]:
     if (a[i] > max):
          max = a[i]

print max
```

For everything it does, Python uses statements and expressions. Statements are commands to interpreter in form of keywords, which may or may not result with an output. On the contrary, expressions do not use keywords – they can be equations or functions which may or may not take input and may or may not return a value [19]. In Python, there are two interpreter prompts: primary (>>>), which indicates that Python waits for another statement and secondary (...), which warns that current statement is incomplete.

Python is a dynamic object-oriented programming language and can be used for many purposes. It offers a rich standard library and its community has developed bindings to many of the popular libraries. Some of its features include modules, exceptions, dynamic typing, very high level dynamic data types and classes. Its syntax is often a stepping-stone, but the fact is that it enforces clean code. Where time-critical application of Python is needed, it can be extended with C or C++. It runs on most major hardware and software platforms. Most known implementation of Python, CPython, is written in C, but other implementations also exist [57]. Java-based version of Python exists in form of "Jython" and may be used to work natively with Java code. There also exists a C# version, called Iron Python, which is targeted to .Net framework. For research and development, PyPy project (Python implementation in Python) was founded in 2003 to enable Python developers to change its interpreter as they please in a familiar fashion. It is an open source project, developed jointly by community of developers, but is also supported by the European Union. Python is distributed under an OSI-approved open source license that makes it free to use, even for commercial products. The Python's biggest advantage is very expressive and readable syntax, which means that that with a very short code and with fewer bugs achieve a lot.

There are several drawbacks that are sometimes cited among Python community. Firstly, Python has relatively slow execution speed in comparison with programming languages like C, which is quite logical, because Python is interpreted and highly dynamic, while C is compiled language. Secondly, unlike most of other programming languages, Python is still unstandardized.

And at the end, let us mention the syntax of Python, which is rather similar to basic, with all its simplicity and clumsiness. The idea of using a tabulator instead of BEGIN-END or some kind of parentheses could look cool, but it is a serious drawback in debugging of a program, because there is no way for interpreter to see that some statement is in the wrong block (for example, in a loop even though should be outside of it). Although Python gained some popularity in the last few years, it is probable that it will never jeopardize any of famous programming languages like C or Basic.

4.1.3 Ruby

Ruby is an elegant, general purpose, multi-paradigm programming language designed by Yukihiro "Matz" Matsumoto. Name of this language is a result of a "joking session" among Yukihiro and his friends. While it was not until 1995 that Ruby was released, its development started to design it back in the 1993. It takes features and inspiration from many of the other languages, like Ada, Eiffel and Perl, with traces of Python, Lisp, Clu and Smalltalk. Above mentioned includes arbitrary precision arithmetic, true iterators, user-level threads, first class and higher-order functions, continuations, reflection, Smalltalk-style messaging, mix-in inheritance, autoloading, structured exception handling, and support for the Tk windowing toolkit. As a scripting language, Ruby is interpreted, not compiled, making it somewhat faster to develop with, but slower to run. Furthermore, Ruby is object oriented, functional and procedural language that is able to automatically cast type for Boolean tests. It has rich support for introspection, reflection and multi-programming.

Program 30

```
days = 365
hours = 24
```

As it can be seen from the example above, the values of days and hours are integers, but programmer does not have to give them a type, because Ruby does that automatically. Unlike other typed programming languages, Ruby is dynamically (duck) typed, which means that variable can store any type of data and thus we can say that Ruby is flexible programming language. Here is how duck typing works: if some bird quacks like a duck, walks like a duck, flies like a duck and swims like a duck then it is probably a duck. The same is with types in Ruby — Ruby looks at value assigned to variable and if it acts like an integer then Ruby can assumes that it is probably an integer. There are five types of variables in Ruby that can be identified by their prefix character [28]: local variables that start with lowercase letter or with an underscore character (_); instance variables, prefixed with a single at sign (@) that are referenced via an instance of a class that and thus belong to a given object; class variables, prefixed by two at signs (@@) that can be used by all instances of class; global variables which are globally available to whole program and prefixed by dollar sign ($); and constants that hold constant values during the whole life of an application and whose name is all in uppercase or starts with capitalized letter. Like Perl, Python and JavaScript, Ruby also has an ability to do parallel assignment. The main idea is to assign values to a group of variables in one line or in one statement. What is the most interesting is that in Ruby can be assigned different types of values.

Program 31

```
name, age = "John", 26
```

Strings in Ruby can be created in several ways, but the simplest is to surround them with single or double quotes. Unlike other object-oriented languages, where numbers are consider being primitives, in Ruby they are instances of classes (e.g. number 13 is instance of Integer class, while number 3.14 is an instance of the Float class; both 13 and 3.14 are instances of Numeric class). As in every object-oriented language, everything in Ruby is an object. Every bit of information and code can have its own properties (instance variables) and actions (methods). Methods, delimited with keywords def and end, are the simplest way how statements and expressions can be gathered in one place in order to be reused later (e.g. math operators). Methods can return a value that can be delivered with or without explicit return statement.

Program 32

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
```

Ruby also has a name for a group of statements or expressions – it is a block. However, block is much more than just a group of code; it can be used, for example, in conjunction with the each method for retrieving all the values out of an array.

Program 33

```
(1..10).each { |ind| a[ind] = gets.chomp}
max=a[1]

(2..10).each { |ind|
   if a[ind]>max
      max = a[ind]
}

puts max
```

Ruby has built-in regular expression support (REGEX) and a clean, readable syntax, which makes it a quite nice choice for the first programming language to be learned. Ruby programs are parsed as a sequence of tokens that include comments, identifiers, literals, keywords and punctuation [30]. ”For” and ”while” loops are very rarely used in Ruby; instead of them, constructs like ”each” became normal to implement. Unlike some languages that claim they are general-purpose programming, Ruby is indeed that, both in theory and in practice. Similar to Python, with whom it is often compared to, it has a pretty large collection of built-in methods, even allowing a programmer to change or add new ones to the existing set. One big difference between Python and Ruby is that Python supports multiple inheritance versus Ruby’s single inheritance. Like its ancestor

Lisp, Ruby has a real, usable nil value, and it treats all values except for nil and false as true. Lately, Ruby is often regarded as one of the best languages for implementation of domain-specific languages (DSLs) due to its powerful meta-programming capabilities. With a popularization of Leavengood's RubyGems packed management system during the year of 2001, Ruby started to gain its popularity. RubyGems enables transfer of Ruby code (popularly called gems) on every computer that has Ruby installed and thus solves problems with code versions and code interdependency [11]. Currently, Ruby is in top 10 of the most popular programming languages in the world [101]. Much of its popularity can be attributed to the latest surge of various high-quality web frameworks like Ruby and Merb. It is distributed according to Free Software ideas and developed by following open source practices. Ruby, as a lot of modern languages, includes a feature called garbage collection. What separates Ruby from the lot is that its garbage collection is very efficient, resulting in less memory leaks. Memory allocation space does not have to be released as it has to be in C++.

Nowadays, when web has become inevitable part of our everyday life, it is almost impossible to imagine communication, collaboration and business activities without it. However, web still has plenty of constraints which represent a big challenge for programmers. For example, it is not possible to achieve complete compatibility among web browsers, which means that some application can work perfectly in one web browser and do not work at all or work with errors in another web browser. On the other hand, web applications are made of small, mutually connected parts that are becoming more complex and thus heavier to maintain. In order to solve those problems, a web framework Ruby on Rails has been released in 2004. Its goal is to increase speed and simplicity of web development. Ruby On Rails was extracted from Basecamp, David Heinemeler Hansson's project at 37signals. Subsequently, it was extended and improved by a team of core committee members and outside contributors. Rails framework is based on two main concepts: Don't Repeat Yourself (DRY) and Convention over Configuration. According to DRY, it is necessary to avoid duplicating information within application, because it leads to errors and bugs. Therefore, it is recommended to express information in one, authoritative location from which it can be accessible to every part of application. Furthermore, Convention over Configuration is also a crucial concept in web application development. When using some popular frameworks like .Net, Struts, Zope or Cocoa it is necessary to configure plenty of files before the programming and an application is finally developed, it is full of bugs so have to be spent hours on errors removal. What is even worse, all those steps have to be repeated again with every new project. Rails framework solves this problem with naming convention where majority of configuration files are unnecessary and in most cases file for database connection has to be configured. Of course, there is also possibility to create configuration file in some specific situation or problem domain. Ruby On Rails is based on MVC (Model View Controller) architecture that separates logical parts of an application (data, user interface and actions) and enables their separated modification and testing. MVC contributes to Rails' advantages such as faster development, easier maintenance and increased reusability among components. Finally, what distinguishes Ruby from other frameworks is agility. Agile development is based on adaptive approach, which means that an application is developed through variety of iterations and before the development of the next one starst, results of previous iteration have to be validated. When used, agile development approach enables easier and faster web application development and it is less like that development process will veer out of control [52]. Furthermore, Rails has integrated protection against two of ten top web application attacks: SQL injection and Cross-site scripting (CSS) [90].

Program 34

```
class StudentsData < ActiveRecord::Migration
def self.up
```

```
create_table :students do |t|
t.column :name, :string
t.column :email, :string
t.column :phone, :string
t.column :address, :string
end
end

def self.down
drop_table :students
end
end
```

In the example above can be seen Ruby class, that is ActiveRecord migration is used to define the scheme of table. Without the line below that has to be executed within Rake building language for Ruby, migration class does nothing.

Program 35

```
$ rake db:migrate
```

Ruby on Rails is released under MIT license, while Ruby itself is licensed under Ruby license. Nowadays, Ruby is used worldwide for artificial intelligence and machine learning research, text processing, web applications and general system administration [6]. On the other hand, Rails framework is used for development of some popular projects such as chat client Campfire, file storage system Strongspace, podcasting share site Odeo, etc. [90].

4.1.4 Java

Java is a multi-platform, object-oriented programming language [35]. It was initially developed by James Gosling and his colleagues at Sun Microsystems and it was originally indented to replace C++, because C++ and other similar programming languages were not suitable for certain purposes and did not have some needed features such as platform portability. Java was at first called Oak [100] after the oak trees that were growing outside of Gosling's office, but its name was consequently altered because Oak was the name of already existing programming language. The name Java was chosen by several individuals involved in the project: James Gosling, Arthur Van Hoff, Andy Bechtolsheim in 1995.

Many features of Java, especially syntax comes from C and C++. However, Java has a simplified object-model and many other differences. Java source files are compiled to byte code (a binary representation of an executable program), which can run on Java virtual machine that exists on any platform and that is why it can be said that Java is platform independent [55]. When Sun published the first public version of Java (Java 1.0), they promised the WORA concept (Write Once, Run Anywhere), which he fulfilled by making many run-times for most popular platforms public.

Java can be most easily described simply as a third generation programming language that is similar to C but is not superset of C (Java compiler will not compile C code). Java also suppresses some features of C, like operator overloading or multiple inheritance in classes in order to simplify the language and to prevent certain errors that can arise form usage of these features (such as anti-pattern design). When thinking about programming paradigms, it can be said that Java is imperative, object-oriented, generic and reflective programming language.

In comparison to other similar programming languages such as C, C++, Perl or Python, Java has one special feature. It allows for one to create special programs called applets. Applets are downloaded and run in web browsers. Unlike other executable programs that one would be able to download, applets are most secure because of their many restrictions: they cannot write to user's disk, they cannot write to arbitrary addresses in memory, they cannot plant a virus, they cannot crash user's system – although this feature greatly depends on the operating system used (some versions of Windows are very vulnerable).

The original implementations of Java compilers, Java virtual machines and Java libraries were developed by Sun. Sun has released most of their Java technologies as free software in 2007 under the GNU General Public License. There are also other alternative implementations of these Sun technologies, such as GNU Compiler for Java and GNU Class path. Java was very soon accepted and embraced by various browsers.

With the Java 2 version a multiple configuration of Java emerged for different platforms: J2SE (Standard Edition), J2EE (Enterprise Edition) and J2ME (Mobile Edition). These were later renamed to Java SE, Java EE and Java ME. In 1997 Sun tried to formalize Java through ISO/IEC JTC1 and ECMA International, but they soon gave up, so Java is today de facto standard, which is maintained by JCP (Java Community Process) that was established in 1998. In 2006 Sun released a large part, and in 2007 a whole Java source as free software (an open source [58]).

There are many advantages of Java that makes it very popular today, particularly in certain areas of programming (such as banks, assurance societies, etc.) that need something portable, little better and a bit more secure. Java is a complete programming language with many features, concepts, frameworks, design patterns, plug-ins and libraries that make it very powerful. Its platform independence makes it ideal for usage on various servers that use different platforms. Java programs can run as standalone or in a browser. However, all these features and possibilities make Java quite complex, so Java is definitely not the language for beginners. Popularity of Java is increasing and there are new libraries, design patterns and other features and concepts that are added to this language by various communities almost every day.

Program 36

```
public static int max(int[] t) {
    int maximum = t[0];
    for (int i=1; i<t.length; i++) {
        if (t[i] > maximum) {
            maximum = t[i];
        }
    }
    return maximum;
}
```

The main criticism against Java is addressed to its speed and to its incomplete syntax and set of properties comparing to C/C++ [85]. Java is, according to its critics, language that is derivated from C++, with the idea of making it multi-platform and easier to learn. Some advanced constructs (such as pointers, multiple inheritance, etc.) are excluded from it, but it still inherit one of the main drawbacks from C++ – its complexity, and it is much slower than C++, and its semantics is not so consistent.

As afore said, Java is not defined by any ISO standard yet, although Microsoft and Intel did apply for standardization, and the first standard of the language is announced to come soon.

4.1.5 F-logic

F-logic is not a programming language, but mathematical concept that introduces object orientation into logic programming. To be honest, there is a very little chance that F-logic will become a programming language of any greater impact to the programming languages history. The reason of including this language in this paper is that it is probably the only logic object-oriented language today.

The development of F-logic started in 1995 with paper [47], in which authors developed the theory of two-dimensional classification of programming languages, adopted in this paper, and tried to develop the language from the only category of programming languages that was still missing – object-oriented logic programming languages. They developed logic framework for, as they called it, object-oriented and frame-based languages, two paradigms that they prove to be in itheir fundamentals the same, differing only in the names of the concepts (objects are, in fact, frames and methods are slots). The first implementation of F-logic was developed at Freiburg University by the G. Lausen and his group in 1998. FLORID system was developed as a query language for databases, and as such, it has a reasoning system based on stratification. The second, more popular implementation, based on tabled resolution (SLG resolution) was developed by the group of scientists at Stony Brook, leaded by D. Warren and M. Kifer, the implementation called Flora. The first Flora implementation was published in 2003. Flora programming language is not implemented in an independent environment, but in XSB Prolog environment, that includes Prolog, HiLog and F-logic programming languages. The problem that still exists is that F-logic and Flora are almost completely oriented to data definition, so their main purposes are database definition (FLORID) and ontology definition (Flora).

Program 37 ([94])

```
paper[authors=>person,title=>string].
journal_p::paper[in_vol=>volume].
conf_p::paper[at_conf=>conf_proc].
journal_vol[of=>journal,volume=>integer,
           number=>integer,year=>integer].
journal[name=>string,publisher=>string,editors=>person].
conf_proc[of_conf=>conf_series,year=>integer,editors=>person].
conf_series[name=>string].
publisher[name=>string].
person[name=>string,affil(integer)=>institution].
institution[name=>string,address=>string].

o_j1:journal_p[title->
               'Records,Relations,Sets,Entities,and Things',
              authors->{o_mes},in_vol->oi_11].
o_di:conf_p[title->'DIAM II an dLevels of Abstraction',
            authors->{o_mes,oeba},at_conf->ov76].
o_i11:journal_vol[of->o_is,number->1,volume->1,year->1975].
o_is:journal[name->'InformationSystems',editors->{o_mj}].
o_v76:confproc[of->vldb,year->1976,editors->{o_pcl,o_ejn}].
o_vldb:conf_series[name->'Very Large Databases'].
o_mes:person[name->'Michael E. Senko'].
o_mj:person[name->'Matthias Jarke',affil(1976)->o_rwt].
o_rwt:institution[name->'RWTH_Aachen'].
```

4.1.6 C#

C# is strong typed object-oriented programming language [66]. It was developed by Anders Hejlsberg, author of Trubo Pascal and co-author of Delphi, as a part of .NET programming languages package. It was later approved by ECMA [99] and ISO [98]. C# has a procedural object-oriented syntax based on C++. It was also influenced by other languages such as Delphi. Its main emphasis is on simplicity.

Sun Microsystems has released Java in 1996 purchased a licence in order to implement Java as a part of their operating system. Microsoft broke the licence agreement and made a few changes that suppressed Java's platform-independent nature. Sun sued Microsoft and they come to a settlment at the end, when Microsoft decided to create their own version of this kind of language that would syntactically be related to C++.

The name C# (C sharp) comes from musical notation where a sharp indicates that a note should be played a half-step higher in pitch. This name concept is similar to C++, where ++ means that variable should be incremented by 1. Because the original musical sharp sign is not part of standard keyboard and fonts, the sign # was chosen to represent a sharp sign. This is defined in the ECMA-334 C# Language Specification. The real sharp sign is however still used in printed materials, commercials, etc. The # sign has been used also in other parts of .NET packages, such as J# (Microsoft version of Java) and other Microsoft programming languages.

C# is written as a simple, modern, general-purpose and object-oriented programming language that supports software components, internalization, robustness, etc. C# differs in many aspects form C and C++, and it adopted many of its semantics from Pascal. C# has no global variables or functions, everything is declared inside of classes. C# is strong-typed unlike C and C++. Its types and their implementation is similar to Pascal strong implementation of data types, with full boolean data type that is not only the interpretation of integer. Memory address pointers can only be used inside of code block that is marked as unsafe and it needs a special permission to be executed. The whole implementation of pointers that is used in C and C++ is abandoned and Pascal "safe" pointers are used instead. In C# multiple inheritance is not allowed, as it is in Java. These are just some of the differences. Another concept adopted from Java is garbage collection that is implemented in C#, too. Therefore, we can say that C# is syntactically very close to C++ programming language, but in its semantics it is much more like Pascal and Java than like C or C++. C# is platform independent and supports and makes it easy to work with web services and is as such a direct competitor against Java [4] and Delphi.

Program 38

```
public int getMax(int[] thearray)
{
    int max = 0;
    for (int i = 1; i <> thearray[max])
   {
        max = i;
   }
   return thearray[max];
}
```

Microsoft Corporation, Hewlett-Packard and Inter Corporation together sponsored the submission of all C# specifications as well as the specification of the underlying CLI (Common Language Infrastructure) to the ECMA International in August, 2000. In December, 2001, ECMA released the ECMA-334 C# Languages Specification. ISO denoted C# as a standard in 2003 under the name ISO/IEC 23270:2006 – Information technology – Programming languages – C#.

ECMA approved 2^{nd} edition of C# specifications in December, 2002 and 3^{rd} edition in June, 2005. ECMA 4^{th} edition and ISO 2nd edition of C# standard followed in July, 2006.

Complete list of C# and CLI standards is given below:

- ECMA 334 - C# Language Specification
- ECMA 335 - Common Language Infrastructure
- ECMA 335 - XML-based Library Specification
- TR-084 - Information Derived from Partition IV XML File
- TR-089 - Common Generics Library
- ISO/IEC 23270:2006 - Information technology - Programming languages - C#
- ISO/IEC 23271:2006 - Information technology - Common Language Infrastructure (CLI)
- ISO/IEC TR 23272:2006 - Information technology - Common Language Infrastructure (CLI) - Technical Report on Information Derived from Partition IV XML File
- ISO/IEC TR 25438:2006 - Information technology - Common Language Infrastructure (CLI) - Technical Report: Common Generics

Although C# was meant to serve as a competitor against Java, there are some lacks of C# when compared to Java. Programs that are written in .NET environment and other similar virtual machine environments generally require much more memory than similar applications written in languages like C++ that do not have an extensive associated library. C# applications are generally also slower that native languages applications. Also, although there are environments, such as Borland Developer Studio, that support C# in Windows, Linux, BSD or Mac OS X and some of them even provide complete implementation of C# language and CLI, none of them provides a complete implementation of all libraries that are available for Microsoft and are currently supported only by Windows implementation.

To conclude, C# is an interesting language with many interesting and new concepts. It is definitely a language that has a potential to grow to be even more popular and, compared to Java has many new features. However, there are also some problems and lacks that have to be solved and improved, like compatibility and better portability of code, better support for error trapping, etc.

4.2 The History of African Tribes: Scripting Languages

Scripting languages are very different type of languages than those described before, yet very often it is not completely clear if some language can be referenced as a script language or not. The main property that makes some programming language a script language is that it does not work as a complete, independent language in the system, but it works as an additional level on other application, program or system. This property makes almost any programming language possible to work as a script language, but there are some languages that can work only in the context of some applications in the operating systems. These are true scripting languages. The first scripting languages were languages that enhanced operating systems in the form of shell scripts, adding some new properties to operating system commands, such as variables, control structures etc. Today every operating system has its own scripting language for shell script writing. Some of them are quite well developed and give great amount of programming capabilities.

The second field in which scripting languages took a great part was text editing and desktop publishing. All modern desktop publishing tools (such as Microsoft Word) are based on some

scripting language, although this language is the most often hidden behind the WYSIWYG user interface. Some of the scripting languages are independent on tools. Probably the most famous scripting languages for desktop publishing are TEX and LATEX. They are the only desktop publishing scripting languages that are still used without any WYSIWYG interface today because of their simplicity and power that cannot be achieved through any graphic interface. Another very popular scripting language for desktop publishing is Postscript language, which is complete language with impressive capabilities. Postscript is very popular scripting language for communication with Postscript printers, making printing files much smaller and much more compact and interpreted on the very printer. At the end, there is another scripting language that could be included in this class of scripting languages, although its purpose is not desktop publishing, but document publishing through the Internet. That is HTML. HTML is the scripting language for representation of World Wide Web pages including all their graphic properties.

HTML gives introduction into another very important field of scripting language implementation – Web programming. This is the field of usage where scripting languages are maybe the most developed. There are several layers of scripting languages for Web programming. The first is the layer where HTML exists. This is the layer of the scripting that communicates directly with the web browsers, and allows description ofgraphic properties of Web pages in the concise way. The problem of HTML is that it is rather static (although it allows some level of interaction through input masks). There are many languages, like PHP, that can be embedded into HTML code to enhance its capabilities. Languages that are incorporated into HTML code are rarely complete languages with all language constructs. These languages form the second layer of Web programming languages. The third layer consist of languages, such as JavaScript, Perl and Python, that are much more complete, and use different technology. They use so called Common Gateway Interface (CGI) that provides an interface for information (usually web) server to external applications, written in these languages.

These are not only fields of usage of scripting languages. Scripting languages are used in GUI programming, as a specific languages in some applications (such as Lisp in AutoCAD or EMACS), as an enhancement of RDBMS (such as PL/SQL) and so on. The main purpose of scripting languages remain an enhancement of some other application or language, but that includes variety of specific fields of usage and concepts of development. For that reason some of the scripting languages can scarcely be compared to classic programming languages, and therefore described them in the separate section.

4.2.1 HTML

HTML stands for HyperText Markup Language. Markup languages have been in use in various areas for many years and HTML defines the usage of these kind of language in structuring and formatting the layout of a web page. HTML consists of many tags that are surrounded by angle brackets that denote some HTML element such as unordered list, a break, etc. HTML can also have embedded scripting code in it (such as JavaScript).

HTML as and idea emerged in 1980 and its founder was Berners-Lee. He mentioned HTML for the first time in the late 1991 in a document called HTML Tags [97], [96]. This document described 22 elements that HTML consisted of then. 13 of these elements are still present in today's HTML. In 1996 HTML was accepted and maintained by World Wide Web Consortium (W3C) and in the year 2000 it was accepted as an international standard (ISO/IEC 15445:2000). The last HTML specifications have been published in 1999 by the W3C (that maintains the HTML specifications) in the HTML 4.01 Recommendation. HTML 5 was published in 2008 but as a working draft only.

Today's HTML consists of several elements [97]:

- document-type declaration

- elements along with their attributes
- character-based data types
- character and entity references

XHTML is a language that is still in development and is based on the reformulation of HTML using XML 1.0 [65]. It can be said that HTML is an evolving language that goes trough constant changes and improvements. Therefore, it has a rich standardization.

The first specification of the language was HTML 2.0, released in 1995. There was no HTML 1.0, but only some draft specification. HTML 2.0 collected those drafts in the single specification.

HTML 3.0 is the version in which more and more people started to be involved and interested in expansion of this language tags and possibilities. The problem that occurred was that then Netscape Navigator was the leading browser and people at Netscape decided to create a set of new tags that will be supported by Navigator. This set of tags was called Netscape Extension tags. Other browsers tried to replicate the same interpretation of this tags in order to stay in the loop, but this was not done exactly accurate so the same page looked different in Navigator than in other browsers, which caused some frustrations among markup language users. The changes in HTML 3.0 were made by HTML working group that was led by Dave Raggett and there was a lot of improvements and new features. But browsers were very very slow in accepting these changes since there were a lot of them, and the whole version of standard that was only a draft was consequently abandoned. Because of this it was decided that any further upgrades of HTML will be done in a modular nature allowing step by step acceptance.

HTML 3.2 was created mainly because there was a lot of browser-specific tags appearing and there was a clear need for some unified standard. In 1994 a W3C was founded in order to standardize the HTML and maintain its further development. Their first work was HTML 3.2, also known as WILBUR. Not all latest achievements were included in this standard they were deliberately left out for further versions of the standard. This version soon got peoples attention and became the official standard in January 1997.

HTML 4.0 was a big step ahead in evolution of HTML standard. Its code name in the early stage of its development was COUGAR. Most of functionalities, tags and features that were introduced in this standard were not new, but old features that were not included in previous standard, especially from unsuccessful HTML 3.0 specification. The feature that was also introduced in this version was support for new HTML presentation language CSS (cascading stylesheets). HTML 4.0 was proposed by W3C in December 1997 and became the official standard in April 1998. This version of standard was accepted by and supported in Microsoft's Internet Explorer 5 and later Internet Explorer 6. Netscape did not support this standard in the proper way until the Netscape Navigator 6. As some time passed since HTML 4.0 came to the scene, its documentation was revised and corrected in some minor ways which resulted in HTML 4.01 which was the final version of the specification.

After HTML 4.01 W3C proposed a new standard and the era of XHTML (eXtensible HyperText Markup Language) began. XHTML 1.0 was the first version of standard and other versions followed. It was a mixture of HTML 4.01 with XML that is far more complicated language than HTML. XML allows a user to create his own tags and attributes in order to suit his needs. XHTML is thus sometimes also called a subset or application of XML. This is a new trend that will be actual in years to come and transition to this standard is not so difficult as one would think. Next year the minor changes of XHTML were included into XHTML 1.1 specification. The future specifications, that are already in drafts are XHTML 2.0 and XHTML 5.0.

Example of language syntax:

```
<div id="menu">
    <ul>
      <li class="first"><a href="index.html">Pocetna</a></li>
      <li><a href="home.html">Home</a></li>
      <li><a href="contact.html">Contact</a></li>
    </ul>
    <div id="date">
      <script language="JavaScript">GetDate();</script>
    </div>
  </div>
```

4.2.2 CSS

As it is well known, the main purpose of HTML is displaying of logical structure of a web page. In the era of HTML 1.0 specification, web documents were very simple as there were only a few HTML tags whose purpose was to handle hypertext links, modest text formatting and images inclusion. In the year 1994, Netscape presented its Navigator, the first commercial browser. Unlike other browsers, Navigator was enriched with new HTML tags for Webmasters giving them the ability to add such things as tables, frames and enhanced text formatting. Shortly thereafter, Microsoft realized that the Web was going to make an impact on the future of computing and communications and they released initial version of Microsoft's Internet Explorer. And that is how "browser war" started in the first place. Throughout many years two of the most prominent competitors of the "web experience", Microsoft and Netscape created new tags that made Web pages not only much more attractive, but also brought plenty of new features. However, as it is with everything, they have their downsides as well. Because tags would increase Web page overall complexity and the page itself would render differently under different browsers, Cascading Style Sheets (CSS) idea was born. CSS development is W3C's effort to introduce standard tags, rather than forcing browser creators to introduce new ones, specific to their own rendering engine [76]. However, it is important to note that CSS does not replace HTML code, but instead interleaves with existing HTML elements, augmenting them. As such, it can be viewed as an extension. That is most evident in a way CSS works: CSS properties are added to existing HTML elements (e.g. font, size, color, etc.) and according to that the renderer interprets them and shows in a standard-compliant way. It retains much of the Web page logical structure, while enabling use of many page layout features in a simple way. And it comes with almost no limitations in elements positioning. CSS information can be specified in a several different ways:

- Embedded within the web page header
- Embedded within the web page body
- Specified in separate file

Program 39

```
body
{
    background color:black;
    color:white;
    font-family:Arial,sans-serif;
    margin: 1px 2px 3px 4px;
    border: 8px solid;
}
```

A CSS rule consists of two main parts: selector and declaration block [60]. Selector is a part of CSS rule outside of curly braces that indicates which part of HTML document will be affected with predefined style. There are four main types of selectors: element, class, ID and pseudo selectors. Element selectors are related to existing HTML tags (e.g. p, H1, body). Class selectors, that start with a ".", can be added to HTML tags with "class" attribute. There are only two differences between class and ID selectors. ID selectors start with "#" and can be used only once within a web document, while class selectors can be used as many times as necessary. Finally, pseudo selectors affected part of web document based on something that is not in the HTML (e.g. hyperlink selectors). In program below there are examples of element, class, ID and pseudo selectors, respectively:

Program 40

```
H1 {
    color: #8855FF;
}
.title {
    font-size: 100%;
}
#header {
    text-align: center;
    width: 80px;
}
A:link {
    color: green;
}
```

Declaration block is a part of CSS rule inside the curly braces that indicates what to do with the item that is selected. Declaration block contains at least one property (e.g. font-size, text-align) and value (e.g. 100%, 80px). When multiple styles are used within one web document and when they affect the same element, they create combined style for this element through cascading property of CSS.

Even though CSS is largely used today, it is little known that it was indeed not the only proposed style language at the time. ISO was developing DSSSL, a complex style and transformation language to be used for printing SGML documents and was supposed to be extension to CSS, too. However, CSS had one important advantage: it understood that web page must be designed according not only to reader and designer, but also has to be well-adapted to display device and browser. Webmasters became accustomed to the features and practices promoted by CSS, but then they wanted something better. CSS2 was introduced in 1998. It was not done in such a way to replace CSS, but to extend it with new functionality and to update many of the previously existing properties. Its importance is in number of fixes to errors and "bugs" encountered in the original CSS specification. Today, CSS3 is the newest there is.

4.2.3 PHP

PHP is yet another scripting programming language that is suitable for creating dynamic websites. It is primarily used for server-side scripting, but it can also be used from command line interface or in graphical standalone applications. PHP is abbreviation for PHP Hypertext Preprocessor [1] that was created as a set of CGI binaries by Rasmus Lerdorf in 1994 [53]. It succeeded an older product named PHP/FI. It was written as a small set of Perl scripts for the need of Razmus'

personal website. Later he wrote a much more complex implementation in C, which could work with databases and was suitable for dynamic websites development. Razmus published his work as an open source under the name PHP/FI. PHP/FI is abbreviation for Personal Home Page / Forms Interpreter and its second version was released in 1997 under the name PHP/FI 2.0. The first version of PHP that resembles PHP as it is used today was PHP 3.0 by Andi Gutmans and Zeev Suraski in 1997 as a complete rewrite of PHP/FI. Andi, Razmus and Zeev agreed to announce PHP 3.0 as an official successor of PHP/FI [1]. Razmus's main motto in PHP development was simple "Be lazy". He intended for PHP to be as simple as possible with a lot of embedded functions for various purposes.

Program 41

```
<?php
function max($a)
{
  $maxVal = $a[0];
  for($i = 1; $i < count($a); $i++)
  {
    if ($a[$i] > $maxVal)
    {
      $maxVal = $a[$i];
    }
  }
  return $maxVal;
}
?>
```

Syntax of PHP is C-like, but simplified. PHP development and improvement was over time generally influenced by. PHP 3.0 brought new features like support for different databases, protocols, APIs, extensibility features and object-oriented syntax support. PHP 3.0 was tested by public for several months and officially released in 1998. It was syntactically largely influenced by Perl form whom it borrowed a great amount of its syntax. When first created, PHP was an alternative of Perl, which provided more easier and simplified way of dynamic scripting. As well as other scripting languages, for example ASP, PHP can be embedded into HTML code by using special tags <? and ?>. PHP 4.0 was officially released in 2000 and intention of this version was to be more suitable for complex applications. It added some new features like: better modularity, support for more web servers, HTTP sessions, output buffering, better security options, etc. A last version of PHP released is PHP 5 which is powered by its core, the Zend Engine 2.0 [4]. PHP 5 includes new features such as: improved support for object oriented programming, better support for MySQL and MSSQL, integrated SOAP support, error handling via exceptions, etc. The last version of PHP is PHP 5.3.0 but PHP 6.0 is also already mentioned to be released soon.

PHP was created as open source and is still maintained as such. This fact as well as its simplicity is to be credited for huge popularity of this particular scripting language today. However, this resulted also in a fact that there are a lot of developed mechanisms that are designed to act as a malicious code and are written to break into PHP based web applications. Consequently, PHP web applications security is one of the main issues and problems that are PHP developers confronted with and this problem is one of the main priorities for all PHP developers/maintainers. PHP is a multiplatform language. There is a PHP interpreter for Windows, UNIX, Linux and Mac. PHP interpreters are very easy to install. They can even be used as plug-ins in combination with a number of web servers such as Apache, iPlanet, IIS, etc. PHP also has a robust built-in support for

a great number of various databases: Oracle, MySQL, mSQL, PostgreSQL, SQL Server, dBase Files, ODBC, ADO, etc. Because of its simplicity and a great number of built-in functions as well as its installation simplicity, PHP development is fast and simple as well as its portability in comparison with other web applications development languages, such as Java and its platform for web applications development JSP. PHP has also built-in support for session handling, cookie handling and form handling. All of these functionalities have been made more secure since PHP 4.1. Similarly to ASP, PHP is in its nature both object-oriented and procedural, which enables PHP programmers to choose whatever approach seems suitable. PHP enables creation of scripted classes using C++/Java like syntax when object-oriented approach is needed or creation of a set of usable functions, if procedural approach is preferred. PHP is weakly-type language similarly as ASP, VBScript or Jscript, which is very different form Java strictly-typed approach. Class method and functions are able to return anything without worrying about casting them as one would have to in C++/Java. However, this can result in more run-time errors since script will sometimes try to access returning value of an object that returns something that it is not suppose to return. PHP is free, simple, fast and equipped with many functions for various purposes, which makes it very suitable for small and large scale web applications. Its popularity is increasing and it is really transforming form an alternative to other scripting languages to a standard of its own.

4.2.4 JavaScript

JavaScript is a scripting language primarily used for web development on the client-side [29]. It is dynamic and weakly-typed. JavaScript looks similar to Java, but is much easier to use and has no essential relation to Java. It was developed by Brendan Eich from Netscape. Firstly it was called Mocha, then LiveScript and finally JavaScript. JavaScript was formed as an original dialect of ECMAScript.

ECMAScript is a scripting language that is the basis of JavaScript. In other words, JavaScript is a superset of ECMAScript which means that is has all abilities of ECMAScript and a set of aditional features. ECMAScript is standardized by the Ecma International organization in the ECMA-262 specification. The latest version is ECMAScript 3^{rd} edition. JavaScript 1.5 and 1.6 are based on this version. ECMAScript 4^{th} edition is currently being developed. JavaScript 2.0 will be based on this version of the standard.

Creator of JavasSript, Brendan Eich (member of Netscape Communication), introduced JavaScript for the first time in Netscape Navigator 2.0 in 1995 [59]. Netscape 2.0 was the first browser that supported JavaScript and this was recognized as a marketing stunt of Netscape to link itself with Java that was emerging and popular programming language. The basic purpose of JavaScript was rather different than that of Java – it was developed to enable web developers to easily add interactivity to their web pages.

JavaScript in fact has 3 forms: Core JavaScript, Client-Side JavaScript and Server-Side JavaScript [59]. Core JavaScript is the basis of JavaScript language and it consists of all elements that are needed to make JavaScript a programming language (operators, control structures, objects, etc.). Client-side JavaScript enables core JavaScript to access and manipulate web documents using DOM (Document Object Model). This is the most popular form of this language. Server-Side JavaScript enables access to databases.

Microsoft named its version of this language Jscript to avoid similar presumptions and it was introduced for the first time in Internet Explorer 3.0 in 1996. Netscape has finally submitted JavaScript to ECMA International which resulted in standardization of JavaScript under the name ECMAScript [59].

One can use JavaScript as a procedural and object oriented language. It can function as both of them. In JavaScript, objects are created programmaticaly. Methods and properties are attached to otherwise empty objects at run time, which differs from classes approach that is used in other

object oriented languages, such as C++ or Java. Object that has been created serves as a prototype for creating similar objects.

JavaScript has C-like syntax, but its purpose is quite different, as already mentioned. Creator of JavaScript summarized the origin of JavaScript syntax in the very first paragraph of the JavaScript 1.1 specification: "JavaScript borrows most of its syntax from Java, but also inherits from Awk and Perl, with some indirect influence from Self in its object prototype system." JavaScript went through versions of the language: Javascript 1.0 and 1.1 (used in Netscape Navigator 1.0), Javascript 1.2 (used in Netscape Navigator 4.0-4.05), Javascript 1.3 (used in Netscape Navigator 4.06-4.5), Javascript 1.4 (used until Netscape Navigator 6.0), Javascript 1.5 (used in Netscape Navigator 6.0, Mozilla Firefox 1.0), Javascript 1.6 (used in Mozilla Firefox 1.5), Javascript 1.7 (used in Mozilla Firefox 2) and Javascript 1.8 (used in Netscape Navigator 3). JavaScript is today a standard when talking about dynamics on web pages and dynamic client side scripting. It has some competitors such as Microsoft JScript, Falsh and ActionScript, VBScript, etc. The tehnology that is commonly used today is Flash, so ActionScript is gaining in its popularity, but there is currently no reason for JavaScrpt to fear form this, because it has many advantages and it has a distinct place in web development. JavaScript is fast and powerful. As it is client side scripting language, there is always a security issue, but nowadays there are many possible solutions that are used to simply hide the JavaScript code such as combining it with PHP, for example. To conclude, JavaScript is still number one choice for most of client side scripting and web pages dynamics.

Example of language syntax:

Program 42

```
var x=new Array(3,4,5,6);
var j=0;
for(i=0;i<x.length;i++)
{
     if(x[i]>x[j])
     j=i;
}
alert(x[j]);
```

4.2.5 ASP

In early days of Web, all web pages were static. The main purpose of web server was not dynamic generation of some parts of web page, but serving clients' requests for static web pages. Therefore, content that was sent from server to client 1 was the same as content that were sent to client 2. In other words, there was no interaction between client and server. After a short period of time, sound and video became constitutive parts of web pages but they were still static with minimal interaction with the meaning of hyperlinking. First step in web evolution was development of Common Gateway Interface (CGI). CGI is a mechanism that enables execution of web application on the web server in a way that application generates HTML code on clients' request and then sends it to web browser. With CGI, Web became very popular among business users as a dynamic medium for information processing. Furthermore, CGI had very high impact on development of client-side scripting languages like JavaScript and VBScript. The main disadvantage of CGI usage is reduced efficiency of web server; every request for web page executes separate instance of web application, which in case of many requests overloads servers' resources. In order to solve mentioned problem, Microsoft proposed Internet Server Application Programming Interface (ISAPI) based on dynamic link libraries (DLLs). Every ISAPI application is a separate DLL

that is placed into memory, where it stays until it is explicitly released. In this case, efficiency of client's requests serving is increased, but another problem has to be resolved – memory pre-occupation. In order to satisfy both efficiency and optimal memory occupation, Microsoft started with development of new technology whose code name was Denali. It was finished and published in December 1996 and today it is well known as Active Server Pages (ASP). ASP is an open, compile-free server-side scripting environment that combines HTML, scripting, custom server components and robust database publishing to create dynamic Web-based applications. Furthermore, ASP allows writing dynamically generated scripts in VBscript (most often used), Jscript, JavaScript or Perlscript. With ASP being one of the earliest adopters of methods by which web pages and their execution were integrated directly into the server, it leveraged their power to boost the performance of a website. When a browser requests an .asp file from Web server, a server-side script begins to run. Web server then calls ASP, which processes the requested file, executes script commands and returns a Web page to the browser. The main advantage of server-side scripting is that scripts cannot be read or copied, because only the result of the script is returned to the browser. It is important to emphasize that ASP is not object oriented technology and that ASP pages are not compiled but interpreted. Furthermore, by using various built-in objects, programming ASP websites is made much easier, because they enable development of dynamic content by each of them being a representative of most commonly used functionality. In ASP 2.0 that was released in September 1997, there are six such built-in objects: Application, ObjectContext, Request, Response, Server, and Session (e.g. cookie-based object that maintains variables from page to page) [89]. Three years later, ASP 3.0 brought relatively modest changes regarding its predecessor. One of the most important additions was the Server. Execute methods, as well as the ASPError object were also added. By using ASP, programmer is able to intersperse scripting code snippets written in a scripting language such as JavaScript or VBScript within the HTML code. One thing that all scripting languages have in common is the way how they are embedded into web page and how browser recognizes them as a script and not as a HTML code. All scripting code should be delimited by matching <script></script> tags that are <% ... %> they are about ASP (see example program below).

Program 43

```
<%

Function Max(Array)
Dim I
Dim Highest
Highest = Null

For I = LBound(Array) to UBound(Array)

If CDbl(Array(I)) > Highest Or IsNull(Highest) Then
Highest = CDbl(Array(I))
End If

Next

Max = Highest
End Function

%>
```

In January 2002, Microsoft introduced a successor to it's now "classic" ASP technology which was called ASP .Net. Its significance lies in the fact that it solves many of the problems associated with the earlier mentioned ASP scripting engine and provides integrated and clean development environment on top of .Net platform by taking advantage of its many available libraries and languages. It was actually initially announced as ASP+ and it was done so in conjunction with the .Net platform in July 2000. Basically, .Net is a new development framework consisting of bits and pieces containing a fair number of technologies that Microsoft created during the late 1990's. COM+ component services strive for XML and object oriented design, support for new web service protocols like SOAP, WSDL and UDDI, along with number of other improvements related to web are just part of what it brings to the table. Certainly, ASP .net provides substantial benefits over the "classic" ASP. The name of ASP successor, ASP .Net serves the combination of its two development technologies: Web Forms and Web services. Most important differences between ASP .Net and ASP are [41]:

- ASP.NET is much more event-driven, with the event handlers running on the server,
- ASP.NET separates code from HTML,
- The code in ASP.NET is compiled, not interpreted,
- Configuration and deployment are greatly simplified.

ASP .Net differentiates in the fact that it does not use interpreted languages anymore, even though it can if they are part of the .Net family of languages. Instead, it most often takes advantage of modern, object oriented languages, like C# or Visual Basic .Net (as can be seen in the program bellow). In any case, this brings considerable performance boost.

Program 44

```
<%@ Page Language="VB" Debug="true" %>
<script language="VB" runat="server">

Public Function Max_value(ByRef Array As Variant) As Integer
Dim i As Integer
Dim Max As Integer
Max = 0
For i = 1 To UBound(Array)
    If Array(i) > Array(Max) Then
        Max = i
    End If
Next
Max_value = Array(Max)
End Function

</script>
```

The biggest changes in the ASP.Net releases came between versions 1.1 and 2.0. ASP .Net 1.1 introduced the concept of "code behind pages", where the interface was developed on the screen using a GUI format and the code was placed in a separate class file "behind" the interface file. When these files where compiled, they together created an ASPX web page. As opposite from mentioned, ASP .Net 2.0 introduced "code separation", which promoted object orientation and

led to more modular code. The main idea was to enable the separation of the layout from the code so that the markup page can be modified separately from the code page. Unlike previous two, ASP .Net 3.0 was not new version of ASP .Net framework. It was just a name for an extra set of application programming interfaces for an existing ASP .Net 2.0 framework with support for Windows Presentation Foundation, Windows Communication Foundation, Windows Workflow Foundation and Windows CardSpace. ASP .Net 3.5 is the latest version of ASP .Net framework, which was released in November 2007. It came with new client-script libraries, controls and types in order to support AJAX Web applications development. ASP and especially ASP .Net are still very popular today, with Microsoft behind them. However, PHP and JavaServerPages are gaining on popularity and are becoming a sort-of-standard on non-Microsoft platforms.

ASP is implemented only on Microsoft Windows platform, and this is pointed out as its main drawback. PHP, which is much less developed language than ASP, remains one of the leaders in field of scripting languages for accessing databases through the Web, just because ASP cannot be used on other platforms than Microsoft. The second serious objection to ASP is that VBScript language is used as a programming language. Microsoft solved this problem by letting ASP pages be written in JScript or PerlScript language.

5. Postmodern Age: The State of the Art and the Conclusion

Approximately 15 years ago there was a great expansion of different programming languages using different paradigms and with different purposes. There were many ”small” languages that were promoted by different groups of programmers and were used almost only by them. Many of these languages survived until today in some form, but only few of them are profiled as a main languages today, those that hold the majority of programming field.

In the classic programming environment today C-like syntax became de facto standard of programming languages. According to [101], more than 45% of all program code today is written in one of three languages with very similar syntax: Java, C and C++. That covers not only a code for operating system and desktop programming environments, but also Web programming and other programming environments. The only programming language with significantly different syntax that takes a greater part in the percentage of program code is Visual Basic with little less than 10%. It is significant that Java is still the most used programming language, regardless the C#, which is designed to take a part of the programming field that is covered with Java. The second thing that is important and has to be pointed out is that C# is relatively new programming language, and part of the people that like new things were attracted to it at first, but its popularity in the last time is stagnating, or slightly falling down, and its place in the popularity table is seriously jeopardized by Delphi, the most popular Pascal-like language, which got a new bust with the new Borland Developer Studio.

The significant fact is that popularity of all main three languages failed for 5% each, while languages like Python, Ruby, PHP and Delphi slowly gain more and more popularity.

Web applications are becoming more complex today and that is the reason why percent of code written in classical programming languages, like C and C++ stagnate while languages that are connected with Web programming increase their popularity. The only language that is in a way connected to the Web programming whose popularity significantly failed is Java, but it is not clear wether the percent of code that Java gave to other languages is the part that is connected to Web programming, because Java is often used for non-Web programs, too.

If the languages are observed according to programming paradigms, it can be seen that procedural object-oriented paradigm is the leading paradigm, having more than 50% of programming code today. Functional and logical programming languages have no more than 3% of code. That gives us more than 97% of programming code written in procedural programming languages today. Although functional and logical programming paradigms are theoretically important, it can

been can that programmers stick with classical, procedural programming. The reasons for that are several. A procedural programming languages are much easier to learn, while for functional and logic programming language a programmer has to have much more theoretical knowledge about a paradigm, and practical knowledge of a language to start programming. The next thing is that procedural programming environments are much more developed than functional and logic ones, and it is very hard to develop in functional or logic environment any serious program that includes modern user interface, database access and other properties that are present in almost every modern computer program.

In the last few years Microsoft Inc. has the tendency to bind programming languages tighter with the operating system (.Net technology) to make computer resources easier to access them. The second important tendency that is present, especially in Windows environment is multi-language platforms. .Net and ASP have interfaces to several languages, such as VBScript, JavaScript, etc. The new Borland Developer Studio allows programming teams to build components of their applications in Delphi, C/C++ and C# and run them together without any need of interfaces among languages, as well as native interface to Borland Interbase. In the same way new Microsoft Visual Studio treats Visual Basic, C/C++ and ASP and languages that ASP has interface to, as well as native interface to MS SQL and Access databases.

The similar interface in Linux environment is built for Perl 6, which has extensions for Prolog, Python and Babel, but also for regular expressions and for λ-calculus, covering functional programming in its most native form.

The programming platforms are today becoming more and more language independent, more and more integrated with the operating systems. Through the native interface, ODBC, OLE DB or some other interface SQL language is embedded into procedural languages. Web programming is becoming the important part of the programming field, although it is still not (and never will be) so dominant part as some of people that are specialists for Web programming hope it will become. Most of modern programming environments have easy to use Web programming module. Procedural or functional languages are often used as scripting languages, giving more power to programming scripts for other applications. These are modern tendencies in programming, and they will become even more significant with the time. Not many new programming languages nor programming paradigms can be expected in the following years, but for sure new aspects of usage of programming languages, development of multi-language environments, and even simpler and easier to use interfaces to Web, databases and computer resources can. Nowadays the amount of the code that can be automatically generated is significant and will raise in the future, leaving only the most special tasks and the most inventive part of programming to be done manually. This fact already changed the profile of programmers that are needed and will change it in the future even more. Programmers would have to become more educated in solving problems and less in coding trivial tasks in some programming language.

References

[1] Achour, M. et. all: *PHP Manual*, URL: http://hr.php.net/manual/en/index.php, Retrieved: 22. 10. 2008.

[2] Abadi, M; Cardelli, L: *A Theory of Objects*, Springer-Verlag, New York, 1998.

[3] Apt, K.R: *From Logic Programming to Prolog*, Prentice-Hall International, London, 1997.

[4] Anderson, T: *C# pulling ahead of Java – Lead architect paints rosy C# picture*, Reg Developer. The Register, URL: http://www.regdeveloper.co.uk/2006/11/14/c-sharp_hejlsberg/, Retrieved 9.12.2007.

[5] Armstrong, D.J: *The Quarks of Object-Oriented Development*, Communications of the ACM, pp. 123-128, 2006.

[6] Baird, K. C: *Ruby by Example: Concepts and Code*, No Starch Press, San Francisco, 2007.

[7] Barnes, J: *Programming in Ada 2005*, Addison-Wesley, London, 2006.

[8] Bauer, F.L; Wössner H: *The "Plankalkül" of Konrad Zuse: A Forerunner of Today's Programming Languages*, Communications of the ACM, Vol. 15, No. 7. pp. 678-685, 1972.

[9] Beaulieu, A: *Learning SQL*, O'Reilly Media, Sebastopol, 2005.

[10] Bentley, J. H: *Cross-Cultural Interaction and Periodization in World History*, The American Historical Review, Vol. 101, No. 3, pp. 749-770, 1996.

[11] Berube, D: *Practical Ruby Gems*, Apress, Springer-Verlag, New York, 2007.

[12] Böhm, C; Jacopini, G: *Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules*. Communications of the ACM 9 (5), 1966.

[13] Bruce, K. B: *Foundations of Object-Oriented Languages: Types and Semantics*, MIT Press, Cambridge, 2002.

[14] Ceri, S; Gottlob, G; Tanca, L: *Logic Programming and Databases*, Springer-Verlag, Berlin, 1990.

[15] Chang, C; Lee, R.C: *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1975.

[16] Charlette, F: *High tech from Ancient Greece*, Nature 444, pp. 551-552, 2006.

[17] Chapman, S.; Chapman, J: *Fortran 95/2003 Software Scientists and Engineers*, 3^{rd} ed., McGraw-Hill, New York, 2007.

[18] Chen W; Swift, T; Warrem D.S: *Efficient Top-Down Computation of Queries under the Well-Founded Semantics*, Journal of Logic Programming, 24, 1995.

[19] Chun, W. J: *Core Python Programming*, Second Edition, Prentice Hall, Core Series, London, 2006.

[20] Codd, E.F: *A Relational Model of Data for Large Shared Data Banks Communications*, Communications of the ACM, Vol. 26, No. 1, (reprint of the article from 1970), 1983.

[21] Date, C.J; Darwen, H: *A Guide to SQL Standard*, 4^{th} ed, Addison-Wesley, Reading, 1996.

[22] Dijkstra, E: *Go To Statement Considered Harmful*, Communications of the ACM, 3, 1968.

[23] Dijsktra, E: *How do we tell truths that might hurt?*, University of Texas in Austin, Revised and translated letter, 2006.

[24] Dijkstra, E: *Notes on Structured Programming*, 2^{nd} ed., Technological University Eindhoven, 1970.

[25] Doets, K: *From Logic to Logic Programming*, MIT Press, Cambridge, 1994.

[26] Ducasse, S: *Squeak: Learn Programming with Robots*, Apress, Springer-Verlag, New York, 2005.

[27] ECMA International: *ECMA standard 367, Eiffel Analysis, Design and Implementation Language*, Second Edition, URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf, Retrieved 14 October 2008

[28] Fitzgerald, M: *Learning Ruby*, O'Reilly, Sebastopol, 2007.

[29] Flanagan, D, Ferguson, P: *JavaScript: The Definitive Guide*, 4th Edition, O'Reilly, Sebastopol, 2002.

[30] Flanagan, D; Matsumoto, Y: *The Ruby Programming Language*, O'Reilly, Sebastopol, 2008.

[31] Gamma, E; Helm, R;Johnson, R; Vlissides, J: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, 1994.

[32] Gilpin, G: *Ada: A Guided Tour and Tutorial*, Prentice Hall, London, 1991.

[33] Goldberg, A; Robson, D: *Smalltalk 80: The Language*, Addison-Wesley, Reading, 1989.

[34] Goldstine, H. H: *The Computer: from Pascal to von Neumann*, Princeton University Press, Princeton, 1972.

[35] Gosling, J., Joy, B., Steele, G., Bracha, G: *Java(TM) Language Specification*, 2nd ed, Addison-Wesley, Java Series, Reading, 2000.

[36] Hamilton, N: *A-Z of Programming Languages: Ada*, http://www.computerworld.com.au/index.php/id;447175928;pp;1, Retrieved: 5.10.2008.

[37] Harvey, B: *Computer Science Logo Style: Symbolic Computing*, 2nd ed., Vol. 1., MIT Press, Cambridbe, 1997.

[38] Harvey, B:) *Computer Science Logo Style: Advanced Techniques*, 2nd ed., Vol. 2., MIT Press, Cambridbe, 1997.

[39] Harvey, B:) *Computer Science Logo Style: Beyond Programming*, 2nd ed., Vol. 3., MIT Press, Cambridbe, 1997. Machine, BasicBooks

[40] Hudak, P: *Conception, Evolution, and Application of Functional Programming Languages*, ACM Computing Surveys, Vol. 21, No. 3., pp. 359 – 411, 1989.

[41] Hurwitz, D; Liberty, J: *Programming ASP.NET*, 2nd Edition, O'Reilly, Sebastopol, 2003.

[42] Hutton, G: *Programming in Haskell*, Cambridge University Press, Cambridge, 2007.

[43] Kay, A. (1996): *The Early History of Smalltalk*, ACM SIGPLAN conference on History of Programming Languages 1993, pp. 69-95, 1993.

[44] Kempbell-Kelly, M; Aspray, W: *Computer: A History of Information Machine*, BasicBooks, Jackson, 1996.

[45] Keringham, B.W: *Why Pascal is Not My Favorite Progamming Language*, AT&T Bell Laboratories, 1981.

[46] Keringham, B.W., Ritchie, D: *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.

[47] Kifer, M; Lausen, G; Wu, j: *Logical Framework of Object-Oriented and Frame-Based Languages*, Journal of the ACM, Vol. 42, pp 741-843, 1995.

[48] Knuth, D.E: *Structured Programming with go to Statements* ACM Computing Surveys 6(4), pp 261-301, 1974.

[49] Krasner, G: *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, 1983.

[50] Krishnamurthi, S: *Programming Languages: Application and Interpretation*, Brown University, 2007.

[51] Lalonde, W: *Discovering Smalltalk*, The Benjamin/Cummings Series in Object-Oriented Software Engineering, Addison-Wesley, Reading, 1994.

[52] Lenz, P: *Build Your Own Ruby On Rails Web Applications*, SitePoint Pty. Ltd, 2007.

[53] Lerdorf, R.; Tatroe, K.; MacIntyre, P: *Programming PHP*, Second edition, O'Relly, Sebastopol, 2006.

[54] Lewis, B; LaLibrete D; Stallman R: *GNU Emacs Lisp Reference Manual*, 1998. URL: http://www.gnu.org/software/emacs/manual/elisp.html, Retrieved: 10. 10. 2007.

[55] Lindholm, T; Yellin, F: *The Java Virtual Machine Specification*, 2nd ed, Prentice-Hall Int, Palo Alto, 1999.

[56] Lloyd, J.W: *Foundations of Logic Programming*, Springer-Verlag, New York, 1984.

[57] Lutz, M: *Learning Python*, Third Edition, O'Reilly, Sebastopol, 2008.

[58] Martens, C: *JAVAONE: Sun - The bulk of Java is open sourced*, ITWorld an Open Exchange, 2007, URL: http://www.itworld.com/070508opsjava, Retrieved: 15. 10. 2008.

[59] McDuffie S.T: *JavaScript Concepts & Techniques: Programming Interactive Web Sites*, Franklin, Beedle & Associates, Wilsonville, 2003.

[60] Meyer, E: *CSS: The Definitive Guide*, 3rd Edition, O'Reilly, Sebastopol, 2006.

[61] Meyers, J: *A Short History of Computer*, 2001, URL: http://softlord.com 2.1/comp/TMP992869671.htm, Retrieved: 12.04.2008.

[62] Milner, R; Tofte, M; Harper, R: *Definition of Standard ML*, MIT Press, Cambridge, 1990.

[63] Milner, R; Tofte, M; Harper, R; MacQueen, D. (1997): *The Definition of Standard ML (Revised)*, MIT Press

[64] Mitchell, J. C: *Concepts in Programming Languages*, Cambridge University Press, Cambridge, 2003.

[65] Musciano, C; Kennedy, B: *HTML & XHTML: The Definitive Guide*, 6 ed., O'Relly, Sebastopol, 2006.

[66] Nash, T: *Accelerated C#*, Apress, Springer-Verlag, New York, 2007.

[67] Paulson, L. C: *ML for the Working Programmer*, Cambridge University Press, Cambridge, 1996.

[68] Phoenix, T; Schwartz, R.L: *Learning Perl*, 3rd Edition, O'Reilly, Sebastopol, 2001.

[69] Quam, Q.H; Diffie, W: *LISP 1.6 Manual*, Stanford Artificial Intelligence Laboratory, Stanford University, Palo Alto, 1968.

[70] Quigley, E: *Perl by Example*, Fourth Edition, Prentice Hall, Upper Saddle River, 2007.

[71] Rantell, B: *From Analytical Engine to Electronical Digital Computer. The Contributions of Ludgate, Torres, and Bush*, IEEE Annals of the History of Computing, 4(4), 1982.

[72] Robinson, J.A: *A Machine-Oriented Logic Based on the Resolution Principle*, Journal of the ACM (JACM), Volume 12, Issue 1, pp. 23–41, 1965.

[73] Rudd, D: *Introduction to Software Design and Development With Ada*, Brooks Cole , Florence, 1994.

[74] Salus, P. H: *The Handbook of Programming Languages (HPL): Object Oriented Programming Languages*, Volume 1, Macmillan Technical Publications, Oxford, 1998.

[75] Sammet, J.E; Garfunkel, J: *Summary of Changes in COBOL, 1960-1985*, IEEE Annals of the History of Computing, Vol. 7, No. 4, pp 342-347, 1985.

[76] Schengili-Roberts, K. (2003): *Core CSS: Cascading Style Sheets*, 2nd Edition, Prentice Hall PTR

[77] Sebesta, R. W: *Concepts of Programming Languages*, 6^{th} ed. Addison-Wesley, Boston, 2003.

[78] Siever, E; Spainhour, S; Patwardhan, N: *Perl in a Nutshell*, O'Reilly, Sebastopol, 1998.

[79] Stephens, R: *Visual Basic 2008 Programmer's Reference (Programmer to Programmer)*, Wiley Publishing, Indianapolis, 2008.

[80] Stroustrup, B: *A History of C++: 1979-1991 AT&T Bell Laboratories*, Murray Hill, New Jersey, 1993.

[81] Stroustrup, B: *The Design and Evolution of C++*, Addison-Wesley, Reading, 1994.

[82] Stroustrup, B: *Why C++ is not just an object-oriented programming language*, ACM SIGPLAN Conference on Object-oriented programming systems, languages, and applications, pp 1-13, 1995.

[83] Stroustrup, B: *The C++ Programming Language*, Special Edition, Willey, Indanapolis, 2000.

[84] Swade, D: *The Difference Engine: Charles Babbage and the Quest to Build the First Computer* (reprint), Penguin, New York, 2002.

[85] Thimbleby, H: *The Critique of Java*, 1998, http://web4.cs.ucl.ac.uk/uclic/harold/srf/javaspae.html, Retrieved: 15. 10. 2008.

[86] Thompson, S: *Haskell: The Craft of Functional Programming*, Second Edition, Addison-Wesley, Essex, 1999.

[87] Tomek, I: *Visualworks Smalltalk: An Introduction*, Series on Object-Oriented Programming, Addison-Wesley, Reading, 1999.

[88] Toole, B.A: *Ada, the Enchantress of Numbers: Prophet of the Computer Age*, Strawberry Press, Oxfordshire, 1998.

[89] Weissinger, K: *ASP in a Nutshell: A Desktop Quick Reference*, O'Reilly, Sebastopol, 1999.

[90] Williams, J: *Rails Solutions: Ruby on Rails Made Easy*, Apress, Springer-Verlag, 2007.

[91] Williams, R: *Punch Cards: A Brief Tutorial*, IEEE Annals of the History of Computing, Web Extra, 2002.

[92] Wirth, N: *Programming in Modula-2*, 3rd ed, Springer Verlag, Berlin, 1985.

[93] Wirth, N : *Recollections about the Deveopment of Pascal*, 2nd ACM SIGMOD Concerence on History of Programming Languages, 1993.

[94] Yang, G; Kifer, M; Wan, H; Zhao, C: *Flora-2 User's Manual*, State University of New York at Stony Brook, 2008.

[95] ***: *BASIC*, Dartmouth College, Computer Center, 1964.

[96] ***: *First mention of HTML Tags on the www-talk mailing list*, World Wide Web Consortium, http://lists.w3.org/Archives/Public/www-talk/1991SepOct/0003.html, Retrieved: 17.11.2007.

[97] ***: *HTML Tags*, World Wide Web Consortium, http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/
Tags.html, Retrieved: 15.11.2007

[98] ***: *ISO/IEC 23270:2003*, 2003
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=36768, Retrieved: 8.12.2007

[99] ***: *Standard ECMA-334 C# Language Specification*. 4th edition (June 2006), http://www.ecma-international.org/publications/standards/Ecma-334.htm, Retrieved: 8.12.2007

[100] ***: *The Brief History of the Green Project*, https://duke.dev.java.net/green/, Retrieved: 17.10.2008

[101] ***: *TIOBE Programming Community Index*,
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, Retrieved: 18. 10. 2008.