Programming Language Families

Procedural, Object Oriented, Logic, and Functional Languages

Jobelle Firme, Nicolas Valera, Yunus Canemre, Stephen Burchill, Beenish Khurshid
Department of Electrical and Computer Engineering
University of Calgary
Calgary, Canada
{jsfirme, nvalera, ycanemre, sburchil, bkhurshi}@ucalgary.ca

Abstract— *There are four major programming language families, also known as a programming paradigm: procedural, object oriented, logic, functional. These programming languages present radically different approaches to automated problem solving and are each best suited for a specific subset of problems. In this paper, we discuss the history of programming language families, the four different programming paradigms, languages that belong to each and how they are used, and what each programming language paradigm is best used for.*

Index Terms—*Procedural Language, Object Oriented Language, Logical Language, Functional Language, Programming Paradigm, Language Families*

I. Introduction

Languages that follow a similar programming paradigm are classified into the same programming language families. In this paper, we discuss the history of programming languages, and discuss at length the four major language families. A brief history of programming languages can be found in section II. Procedural, object oriented, logic, and functional languages are described in sections III to VI.

II. History of Programming Language Families

Charles Baggage's Difference and Analytical Engines, and the Scheutz Difference Engine are some of the earliest computing devices. These devices used mostly physical objects and motion to perform basic arithmetic operations [1]. In a sense, the physical machine represented the first programming languages. The University of Pennsylvania, funded by the United States Army, built ENIAC between 1942 and 1946. The ENAIC was a technological marvel of the time since it had no mechanical parts, nor any moving ones and consisted of only vacuum tubes and wiring [2].

John Von Neumann developed the "conditional control transfer" in 1945 to allow for easier re-programming and code blocks that could be jumped to and from. This innovation allowed for IF THEN, and FOR statements, as well as subroutines [3]. The first compiler, A-o, was written in 1949 by Admiral Dr. Grace Murray Hopper to convert mathematical symbols into machine code [4]. With the advent compilers began the era of modern programming languages.

FORTRAN, less commonly known as The IBM Mathematical Formula Translating System, was one of the first major modern programming languages. See table 1 for a list of major languages and their date of conception. Originally, FORTRAN was an imperative language and looked at computation in terms of the steps that must be taken to reach a certain state. It eventually evolved into a procedural language to support structures like procedures. FORTRAN was used extensively for over half a century and still exists as legacy code in many programs.

The logic programming paradigm, based on first-order logic, shaped languages such as Prolog (1972), which was originally a procedural language. It originated around the same time as functional programming. Hope was the pioneering functional language, developed in the 1977's at Edinburgh University.

Table 1-Major Programming Languages and their Years' of Inception [5, p.38]

Year	**Language**
1952-53	SHORTCODE, Speedcoding, Laning & Zieler
1954	FORTRAN
1955	FLOW-MATIC
1957	COMIT APT
1958	ALGOL 58 (IAL)
1959-60	COBOL JOVIAL LISP
1960	ALGOL 60
1961	GPSS, SIMSCRIPT
1962	SNOBOL
1964	JOSS BASIC PL/I FORMAC
1966	APL\360
1967	SIMULA 67
1968	ALGOL 68
1970	Pascal
1975	C PROLOG
1976	Ada
1980	Smalltalk-80

Functional programming deemphasizes the states that procedural languages focus on and treats computation in a more mathematical fashion. Haskell (1990), Common Lisp (1994), and Scala (2003) are some prominent functional programming languages.

Object Oriented Programming was formalized as a programming concept in the 1960's. Simula 67 was one of the first languages to use object oriented concepts such as classes and instances. C++ (1988), Java (1995), VB.NET (2002), and C# (2005) are some of the commonly used object oriented programming languages today. Object Oriented is considered by some to be the family that absorbs all other paradigms as it incorporates functional, logical, and procedural paradigms.

III. Procedural Programming Languages

Procedural programming is the traditional programming paradigm that is based on the concept of functions and functions calls. These functions often help to divide the system into various modules, and thus create a structure for the said program [6]. Procedural languages will use a defined set of instructions to accomplish a task, and often use a "main" function that then can call a variety of other functions in order to complete a task. As a result of this languages such as C and Pascal are considered procedural [6]. However, even object-oriented languages such as C++ and Java can be considered procedural, since they can still be used to follow the traditional function based structure of procedural languages.

The structure of procedural programs has several key features. Procedures in these programs are completely independent of each other [6, 7]. Any data that a procedure requires in order to complete its operations must be passed to it at the time it is called [7]. Due to this, data used by these procedures must be ready for use before the procedures can be called, as there is no way to pass data to a procedure once it has started running. The example in Figure 1 is of a simple C program that illustrates these concepts [8].

```
#include <stdio.h>
int mult ( int x, int y );
int main()
{
      int x;
      int y;
      printf( "Please input two numbers
to be multiplied: " );
      scanf( "%d", &x );
      scanf( "%d", &y );
      printf( "The product of your two
numbers is %d\n", mult( x, y ) );
      getchar();
}
int mult (int x, int y)
{
      return x * y;
}
```

Figure 1-C code Example [8]

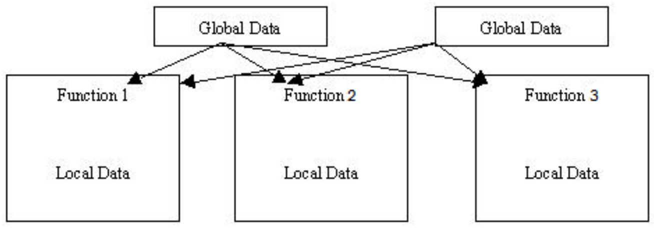


Figure 2-Data Use Diagram [9]

In the example in Figure 1, the program commences by running the "main" function, which sets up variables that contain data required by the "mult" function later in the program. When "mult" is called, the variables "x" and "y" are passed as arguments into the function, which then uses them to compute the product of the two. In both the function prototype and implementation, we can see that it explicitly states the need for two integer parameters in order for the function to be called. Sometimes, data can be stored globally so as to circumvent the need to pass parameters, as illustrated in the Figure 2 [9].

Another key feature of procedural programs is the use of procedures to create modularity within the program [6]. The diagram in Figure 3 illustrates this modularity [9].

The example in Figure 3 illustrates clearly how each function in a program, while potentially invoked by another function, is independent of the other functions. Each function acts as a separate module in the program, which can interact with the other procedures and call them at various points in the program.

In the past, procedural programming and the use of procedural languages has been the standard for developing software in general. Those first learning to program often begin by learning procedural programming. However, studies have shown the clear disadvantages of procedural programming over object oriented programming [5]. In one such study, a set of software science measures and metrics is used to compare the difference between programs implemented in Java and C [10]. This experiment quantitatively shows the clear strengths of the object oriented language Java over the traditional and conventional procedural language C. The procedural nature of C gives it a higher difficulty compared to Java for programming and problem solving, requires more effort to understand a program, and is an overall lower level language [10].

Currently, procedural programming languages are still widely used, as many successful and still used applications have

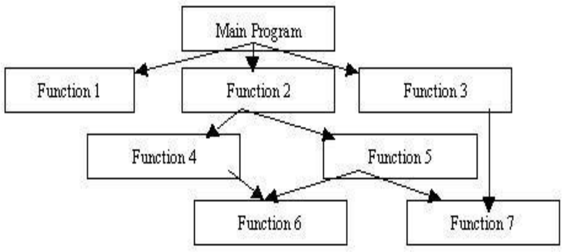
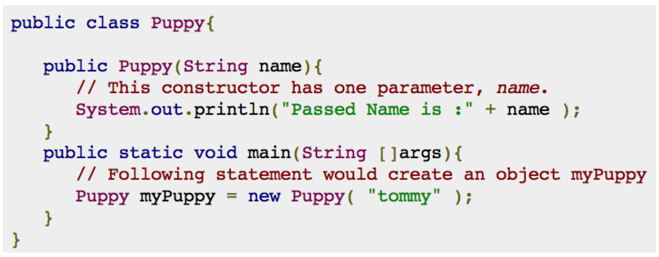


Figure 3-Function Call Diagram [9]


```
public class Puppy{

   public Puppy(String name){
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }
   public static void main(String []args){
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

Figure 4-Example of a class and object in OOP [15]

been written using the procedural paradigm. Procedural languages have a wide variety of applications, such as for the control of production systems [11]. In cases such as these, their simple structure and logical ordering has been very useful for creating solutions to complex problems.

Although their use is sometimes advantageous, procedural languages are in general inefficient, and often inadequate, for the creation of large scale applications in comparison to the newer, more powerful object oriented languages. Though the concept of object orientation cannot be used to solve every problem, objected oriented languages themselves do not have to necessarily be used in an object oriented way, which makes them significantly more versatile than procedural languages. This does, however, show that procedural languages still have their uses, and when used correctly, can still be an effective tool for solving a problem.

IV. Object Oriented Programming Languages

While Procedural Programming Languages focus more on step-by-step procedures and functions, Object-Oriented Languages (OOLs) focus on combining procedures and data structures to create working objects. It is the techniques and structure that a programmer abides by that makes languages and applications object-oriented [12]. Examples of OOLs include Java, C++, and object-based languages, such as BETA and JavaScript. OOLs support objects, classes, and techniques such as data abstraction and encapsulation, inheritance, and polymorphism [13].

One of the largest misunderstandings by people outside of the development world when discussing Object-Oriented Programming (OOP) is that programmers always need to be working with objects in a visual context. However, the word "object' in OOP can refer to any component that does certain actions or activities. A class can be seen as the blank template that describes the structure of a type of object, each class consisting of data variables and behavioral functions [13]. Several unique instances can be made of any class, and the instances are what we refer to as objects. Figure 4 shows an example of a Java class. The variable myPuppy in the main method[1] is an instance of the class Puppy, which is called an object of type Puppy. The easiest way to differentiate classes and objects is to think of classes as the blueprint that can be implemented as unique objects.

Object-Oriented Languages benefit greatly from the use of certain techniques. Data abstraction and encapsulation, inheritance, and polymorphism are some of the techniques that are inherent in these languages and help make OOP useful, easier and more appropriate for certain applications. As stated before, a class consists of data variables and behavioral methods.

Data abstraction refers to figuring out which variables and behaviors are necessary for a class to completely and correctly define the desired type of object [13]. Encapsulation refers to hiding an object's data from being directly viewed from outside of the object. The data can only be viewed or altered by requesting one of the object's own defining methods to do so [13]. Through this technique, programmers can avoid object data from being accidentally changed.

Although data abstraction and encapsulation are fundamentals of OOP, inheritance and polymorphism are more powerful concepts that programmers can use to their advantage to simplify problems and make much more efficient programs. The basic idea of inheritance is the ability of a class to inherit or extend the variables and methods of an existing class [13]. In the Java code in Figure 5, the Basketball class inherits or extends from the Sports class.

On the other hand, polymorphism means assuming multiple forms (type) of an object [12]. It may have a different concept than inheritance but they take ideas from each other [13]. In Java, polymorphism is commonly used to create objects of different types that can all appropriately respond to methods of the same name. The methods are coded with generic object types in mind, and the types can be specified at run-time. The more basic usage of polymorphism refers to being able to use a subclass object in any situation where you could use a parent class object. In Java, the correct methods

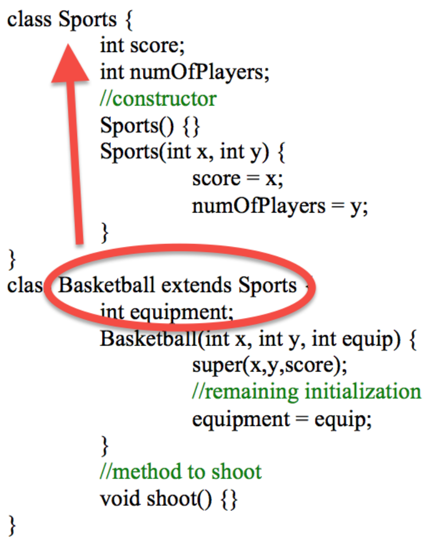
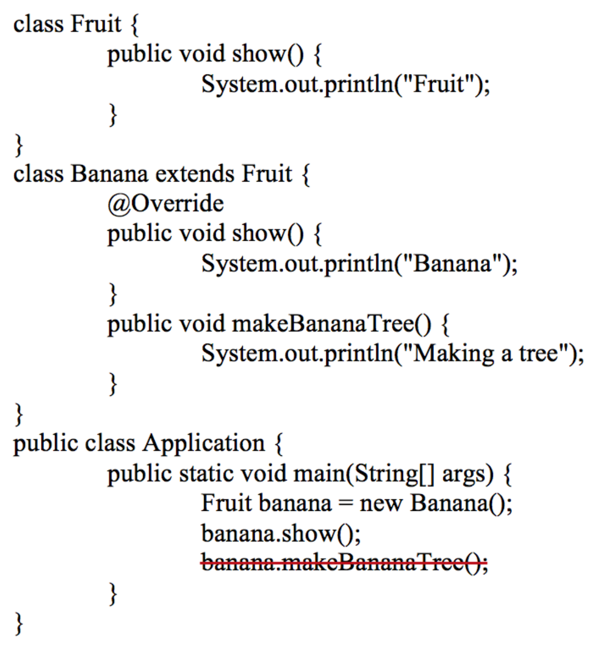

```
class Sports {
        int score;
        int numOfPlayers;
        //constructor
        Sports() {}
        Sports(int x, int y) {
                score = x;
                numOfPlayers = y;
        }
}
class Basketball extends Sports {
        int equipment;
        Basketball(int x, int y, int equip) {
                super(x,y,score);
                //remaining initialization
                equipment = equip;
        }
        //method to shoot
        void shoot() {}
}
```

Figure 5-Java Code Example

[1] Procedures or functions in a class



```
class Fruit {
        public void show() {
                System.out.println("Fruit");
        }
}
class Banana extends Fruit {
        @Override
        public void show() {
                System.out.println("Banana");
        }
        public void makeBananaTree() {
                System.out.println("Making a tree");
        }
}
public class Application {
        public static void main(String[] args) {
                Fruit banana = new Banana();
                banana.show();
                ~~banana.makeBananaTree();~~
        }
}
```

Figure 6-Java Code Example [16]

should be invoked regardless of whether a subclass or parent class object is used [16]. The Java code example in Figure 6 [16] shows how polymorphism works.

In the example in Figure 6, an object named "banana" was instantiated with class type Fruit, but using the constructor of the subclass Banana. The output of the code should be the word "Banana" since Java knows to invoke the correct show() method for the Banana subclass instead of the show() method for the Fruit parent class. However, since the Fruit parent class does not contain a makeBananaTree() method, the code cannot call that method for a Fruit object, even if the object was given the properties of the subclass to begin with.

There are many advantages of using object-oriented languages over any other languages with respect to software engineering projects. These advantages include shorter development times, easier code sharing, and flexibility [17]. Through OOL, building complicated software applications becomes easier and faster. Since OOL works using real world concepts, programmers would find it easier to find solutions in coding using OOL. Code can be reusable (Polymorphism), which would eliminate redundancy and would make development times shorter. It would also be easier to maintain an application because of the encapsulation technique.

Although OOL has many advantages, there are some drawbacks that programmers still need to consider with respect to software engineering projects. The use of OOL is very easy to work with once a programmer gets used to it.

However, learning and getting used to how OOL works may not be as easy as it seems. It takes more time to understand and implement object-oriented code. Not all software engineering problems can be solved or made easier using OOL. An experienced programmer would be required to determine if this programming paradigm is appropriate for a specific project.

V. Logic Programming Languages

Logic programming falls under the category of declarative programming, which is one the four main programming paradigms. A programming paradigm is the fundamental style of how a code is written in a language. Declarative programming is based on describing *what* a code/program should accomplish rather than *how* it accomplishes it. First order logic is used as the high level programming language in logic programming and the beginning were at the language called Planner. The PLANNER, a language developed by Carl Hewitt. Carl Hewitt described it as; "Planner was kind of a hybrid between procedural and logical paradigms because it combined programmability with logical reasoning." [18]. Planner was an influential development that set the path for Logic Programming. Prolog was one of the first languages to use the logic programming paradigm, influencing Datalog, Mercury, Oz and many others.

Logic programming comes with certain advantages. Some of these advantages include: (1) Easy to represent knowledge through the code, (2) natural support for non-determinism, (3) natural support for pattern-matching, (4) natural support for meta-programming [19] (Natural support meaning easily achievable with the expressions you can use in code).

Note that the first advantage takes into account, you are familiar with Prolog and logic programming in general. The programming paradigm terminology are defined as follows;

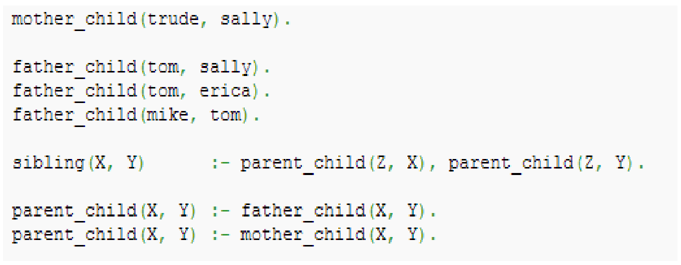
<u>Non-Determinism</u>: A non-deterministic language is one that can support, at various points in a program, different alternatives for program flow. These are not just if-else statements but rather choices that would be made at run time. Prolog can provide this through unification.

<u>Pattern-Matching</u>: As the name suggests, pattern-matching is the act of checking for a sequence.

<u>Meta-Programming</u>: Meta-programming is writing computer code that can manipulate, or write other programs, or itself. This means developer could potentially express a solution in shorter LOC and save time.

So how do these advantages help? Well, non-determinism, meta-programming and pattern matching are major parts in developing AIs, and thus Prolog is a favored language for this task. For example, if you wanted to build the AI for a game, you have to keep track of the state of the program, and check through the states to look for a pattern. In an OO language, or a procedural language, the databases to store states, pattern checking by backtracking previous choices and such would have to be implemented by hand whereas in Prolog and other logic languages these features are built in. This paper will focus on examples from Prolog.

There are three basic constructs in Prolog; facts, rules and queries. A collection of facts and rules make the database. We use facts and rules to create a database that represents a relation of interest. Queries are like questions, and they are asked on the database. This is how we use a prolog program; run queries on the database. Facts are always true, and they are used to evaluate rules, which are conditionally true like 'if' statements.



```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

Figure 7-A Small Knowledge Base [21]

The execution of a Prolog program starts with the user posing a question, which Prolog engine logically tries to evaluate the query to true or false. You can think of the function calls in procedural languages as the clauses in Prolog however multiple clause heads may match one function call. In this case the Prolog engine creates a choice point and continues execution from one alternative. If the goal(the user query) fails during that execution, all changes are undone all the way to the last choice point and the other alternative is chosen. This is called chronological backtracking. This forms the basics of many of the advantages that Prolog provides.

Possible example query: ?- sibling(sally, Erica). To evaluate this, Prolog first looks for a clause matching this query. The closest match is sibling(X,Y) :- parent_child(Z,X), parent_child(Z,Y). Binds X,Y to the corresponding values, and to evaluate sibling, it must evaluate the right hand side, first parent_child(Z,X) and then parent_child(Z,Y). parent_child(Z,X) -> parent_child(Z,sally). Now Prolog looks through the knowledge base for parent_child(Z,sally). There are two matching clauses, with different bodies, so Prolog makes a choice point, and picks the first one. Continuing the logic of binding the variables, the goal evaluates to true for both parent_child(Z,X), parent_child(Z,Y) and there is no need to return to choice point.

Overall, Prolog brings many neat features that are not provided naturally by many of the other programming paradigms, such as OO, or procedural. In order to have features of Prolog in these other paradigms, one has no choice but to implement their own engine according to their needs. However, after much experience in OO, and procedural paradigms, getting a good enough grasp of the language in order to be able to do the cool things Prolog can do is difficult.

VI. Functional Programming

Functional programming is a paradigm that emphasizes the use of functions for calculation over mutable data or changes in state, which is the focus of the imperative programming [22]. Functional programming is based upon lambda calculus which is a formal system used to investigate computability, function definition, function application, and recursion. Many of the functional programming languages can be viewed as expansions of lambda calculus [22].

The difference of mathematical functions and the notion of a function used in imperative programming is that imperative functions can have side effects. These side effects may change the value of program state. It is because of side effects that imperative functions lack referential transparency; the same language expression can result in different values at different times depending on the state of the executing program. Functional code on the other hand, has a set output for functions which only depends on the arguments that are input into the system. This means there are no side effects and thus the program is easier to understand and more predictable [22].

Purely functional programming languages are used mostly only in academia. However some prominent languages such as Lisp, Haskell, Erlang, Scala, and F# have been used in commercial and industrial applications [23]. However, programming in a functional style can be accomplished in most languages that aren't designed as functional languages.

There are a number of concepts which are the main corner stones of functional programming, such as first-class and higher-order functions, pure functions, recursion, strict versus non-strict evaluation, and type systems.

Higher-order functions can take other functions as arguments or return them as results. An example of a higher-order functions is an integrator or differential operator as it returns a function.

Pure functions have no side effects (memory or I/O). This means that pure functions are very useful to optimize code. For example, if the result of a pure expression is not used, it can be removed without affecting the other expressions. If there is no data dependency between two pure expressions, then their order can be reversed or they can be performed in parallel and not interfere with each other. This means pure expressions are thread safe. To allow compilers to optimize code more easily in other languages, there is usually keywords you can add to tell the compiler the function is pure.

Recursion functions invoke themselves, performing an operation multiple times unit the base case is reached. Common patterns of recursion can be re-factored using higher order functions. Some recursions require maintaining a stack but tail recursions can be optimized into the same code used to implement iteration in imperative languages. Functional programming that is limited to well-founded recursion with a few other constraints is called total functional programming [24].

Strict versus non-strict evaluation is a concept that divides functional languages by whether they use strict (eager) or non-strict (lazy) evaluation. Figure 8 presents an example.

Under strict evaluation the entire term and function and evaluated, meaning if any term of the expression would fail, the whole expression would fail. Under non-strict evaluation, the length function will return the value 4 since evaluation it will not attempt to evaluate the terms making p the list. The usual implementation strategy for non-strict evaluation in functional languages is graph reduction [25].

Type systems in functional programming include type inference which allows the programmer to not need to declare types to the compiler. However the use of algebraic data types and pattern matching makes programs more reliable. Well

```
print length([2+1, 3*2, 1/0, 5-4])
```

Figure 8-Evaluation Style

```
#include <iostream>
// Fibonacci numbers, imperative style
int fibonacci(int iterations)
{
    // seed values
    int first = 0, second = 1;

    for(int i = 0; i < iterations - 1; ++i)
    {
        int sum = first + second;
        first = second;
        second = sum;
    }

    return first;
}

int main()
{
    std::cout << fibonacci(10) << "\n";
    return 0;
}
```

Figure 9-Fibonacci in C++

typed programs become a means of writing formal mathematical proofs from which a compiler can generate certified code [26].

Imperative programs emphasize the series of steps taken by a program in carrying out an action while functional programs emphasize the composition and arrangement of functions, often without specifying explicit steps. The following is an example showing how functional programming can convert a 10 line imperative function into a 3 line functional function.

```
-- Fibonacci numbers, functional
style

-- describe an infinite list
based on the recurrence relation
for Fibonacci numbers
fibRecurrence first second =
first : fibRecurrence second
(first + second)

-- describe fibonacci list as
fibRecurrence with initial values
0 and 1
fibonacci = fibRecurrence 0 1
-- describe action to print the
10th element of the fibonacci
list
main = print (fibonacci !! 10)
```

Figure 10-Fibonacci in Haskell

```
fibonacci2 = 0:1:zipwith (+)
fibonacci2 (tail fibonacci2)
```

Figure 11-Fibonacci in Haskell in one line

Pure functional programming as stated does not use I/O but it can perform these tasks, such as accepting user input and printing to the screen, by simulating state. Languages like them implement them using monads, derived from category theory. Moads are a way to abstract computational patterns with mutable state without losing purity [27].

Functional programming does have a downside. The languages of functional programming are typically less efficient in their use of CPU and memory than imperative languages [28].

VII. Conclusion

The history of programming languages has been short, with the first compiler written only in 1949. In the 60 or so years since, many advancements have been made in programming language theory. One such advancement was the advent of programming language paradigms such as procedural, object oriented, logic, and functional. These paradigms were close temporally as they were invented around the same time, but conceptually differ significantly.

Procedural Languages are the simplest programming paradigm and focus on call and return structure and give us the basic principle of modularity. Procedural languages are good for teaching beginners the basics of programming as well as for problems that require lower level solutions such as embedded systems.

Object Oriented Languages, considered by some as the paradigm that absorbs all other paradigms, are widely used in academia and industry. They focus on the principles of data abstraction, encapsulation, inheritance, and polymorphism. These principles combine to make object oriented languages a relatively effective tool to solve most programming problems.

Logic languages are based on first-order logic. It uses the principles of non-determinism, pattern-matching, and meta-programming. Logic programs consist of facts, and rules which used in conjunction answer queries. Though not as commonly used, logic programming is an effective solution for applications such as artificial intelligence.

Functional programming takes a very different approach from imperative techniques similar to the procedural paradigm as it de-emphasizes changes in state and focuses on calculation. Some of the major concepts in functional programming include pure functions, recursion, strict versus non-strict evaluation, type systems, and higher-order functions. Functional programming is a relatively difficult topic to understand and as such is mostly used in academia.

All in all, all four paradigms have made their contribution to the world of programming. Each language provides a solution for a specific niche of problems and as such care should be taken to consider which solution is best for the problem under consideration.s

Acknowledgment

Thank you to all the genius individuals who came before us and brought us our sophisticated world of software engineering. And of course a special thanks to Dr. Kremer!

References

[1] Bromley, Allan G., "Charles Babbage's Analytical Engine, 1838," *Annals of the History of Computing, IEEE*, vol.20, no.4, pp.29,45, Oct-Dec 1998 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=728228&isnumber=15706

[2] Winegrad, D., "Celebrating the birth of modern computing: the fiftieth anniversary of a discovery at the Moore School of Engineering of the University of Pennsylvania," *Annals of the History of Computing, IEEE*, vol.18, no.1, pp.5,9, Spring 1996 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=476556&isnumber=10202

[3] Ferguson, A. (2000), "A History of Computer Programming Languages" Available: http://cs.brown.edu/~adf/programming_languages.html

[4] "Who Developed the First Compiler?", *FairCom eNewsletter*, vol. 23, URL: http://www.faircom.com/ace/enl_23_s08_t.php

[5] Sammet, Jean E., "Some Approaches to, and Illustrations of, Programming Language History," *Annals of the History of Computing* , vol.13, no.1, pp.33, 50, Jan.-March 1991 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4638280&isnumber=4638277

[6] Khan, E.H.; Al-A'ali, M.; Girgis, M.R., "Object-oriented programming for structured procedural programmers," *Computer*, vol.28, no.10, pp.48,57, Oct 1995 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=467579&isnumber=9841

[7] Stannard, K. (2009), "Procedural vs Object Oriented Programming", *Object Oriented ColdFusion*, URL: http://objectorientedcoldfusion.org/procedural-vs-object-oriented.html

[8] Alltain, A. (1997), "Lesson 4: Functions in C", *CProgramming*, URL: http://www.cprogramming.com/tutorial/c/lesson4.html

[9] (2010), "Object Oriented Programming", *C++ Home*, http://www.cpp-home.com/archives/206.html,

[10] Ahmad, A.; Talha, M., "A measurement based comparative evaluation of effectiveness of object-oriented versus conventional procedural programming techniques and languages," *Software Engineering Conference, 2002. Ninth Asia-Pacific*, vol., no., pp.517,526, 2002 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1183072&isnumber=26540

[11] Ishida, T.; Sasaki, Y.; Fukuhara, Y., "Use of procedural programming languages for controlling production systems," *Artificial Intelligence Applications, 1991. Proceedings., Seventh IEEE Conference on*, vol. i, no., pp.71,75, 24-28 Feb 1991 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=120848&isnumber=3453

[12] B. Smith, *AdvancED ActionScript 3.0: Design Patterns*, Apress, 2011, p.428

[13] Schulz, W., "Object-oriented programming", *ACM SIGPLAN Fortran Forum*, vol. 7, 1, pp.11,19, April 1998 URL: http://doi.acm.org/10.1145/291709.291711

[14] Greenberg, I., *Processing: Creative Coding and Computational Art.*, Apress, 2007, pp.301-336.

[15] "Java Tutorial", *Tutorials Point*, URL: http://www.tutorialspoint.com/java/java_tutorial.pdf

[16] Purcell, J. (2013). "Java Tutorial 5: Inheritance and Polymorphism", *Cave of Programming, URL:* http://www.caveofprogramming.com/javatutorial/java-tutorial-5-inheritance-and-polymorphism/

[17] D'Andrea, R.J.; Gowda, R.G., "Object-oriented programming: concepts and languages," Aerospace and Electronics Conference, 1990. NAECON 1990., Proceedings of the IEEE 1990 National , vol., no., pp.634,640 vol.2, 21-25 May 1990 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=112840&isnumber=3357

[18] Hewitt, C., "Middle History of Logic Programming: Resolution, Planner, Edinburgh LCF, Prolog, Simula, and the Japanese Fifth Generation Project", *Logic In Computer Science, arXiv*, Jan 2013 URL: http://arxiv.org/abs/0904.3036

[19] Bruscoli, P., "Prolog Programming", University of Bath. United Kingdom, Bath, 2006. Lecture. URL: http://cs.bath.ac.uk/~pb275/CM20019/P/Prolog1.pdf

[20] Blackburn, P., Bos, J., Striegnitz, K. (2012), "Learn Prolog Now!.", URL: http://www.learnprolognow.org/lpnpage.php?pageid=online

[21] (2013) "Prolog." *Wikipedia*. URL: http://en.wikipedia.org/wiki/Prolog.

[22] Hudak, P., "Conception, evolution, and application of functional programming languages" (PDF). ACM Computing Surveys 21 (3): 359–411, Sept 1989. doi:10.1145/72551.72554

[23] Gregory, J. (2009), "State-Based Scripting in Uncharted 2", *Naughty Dog*, URL: http://www.gameenginebook.com/gdc09-statescripting-uncharted2.pdf

[24] Turner, D.A., "Total Functional Programming", *Journal of Universal Computer Science*, 10 (7): 751–768, 2004

URL: http://pdf.aminer.org/001/001/424/total_functional_programming.pdf

[25] Jones, S. L. P., *The Implementation of Functional Programming Languages*, Prentice Hall, 1987

[26] Leroy, X. (2013), "The Compcert verified compiler", URL: http://compcert.inria.fr/doc/

[27] Newbern, J., "All About Monads: A comprehensive guide to the theory and practice of monadic programming in Haskell", URL: http://monads.haskell.cz/html/index.html

[28] Larry C. P., *ML for the Working Programmer*, ed. 2, Cambridge University Press, June 1996
URL: http://www.cl.cam.ac.uk/~lp15/MLbook/