Topic 11:
Programming Language Families

Zaman Zamanli, Ryan Harvey, Pedro Restrepo Martinez, John Ma, Jeyhun Gurbanov, Jeremy Mah, and Christian Prando

Friday, April-11-14

Introduction

Two Distinct Groups

Languages are generally separated by one specific factor - the handling of state. Languages can either be stateful, which is called imperative; or stateless, which is called declarative. Imperative languages explicitly state the order and control flow of the program (exactly how something must be done), while declarative languages explicitly state what must be true in all states of the program (exactly what must be done) [1]. Underneath the general paradigm of imperative, you have procedural languages, which extend the basic paradigm with the concept of modular functions. Under the declarative paradigm, you have logical programming (where the program is defined by logical mathematical statements) [2], functional programming (where the entire program is defined by mathematical functions) [3], and dataflow programming (where the program is defined in terms of data flow graphs) [4]. Imperative languages are common, easy to understand, provide much finer control over the hardware and usually exhibit much higher single-threaded performance than declarative languages while using less memory. On the other hand, declarative languages don't have to deal with problems of state, meaning they parallelise much easier and tend to be much more stable in threaded operation. [1]

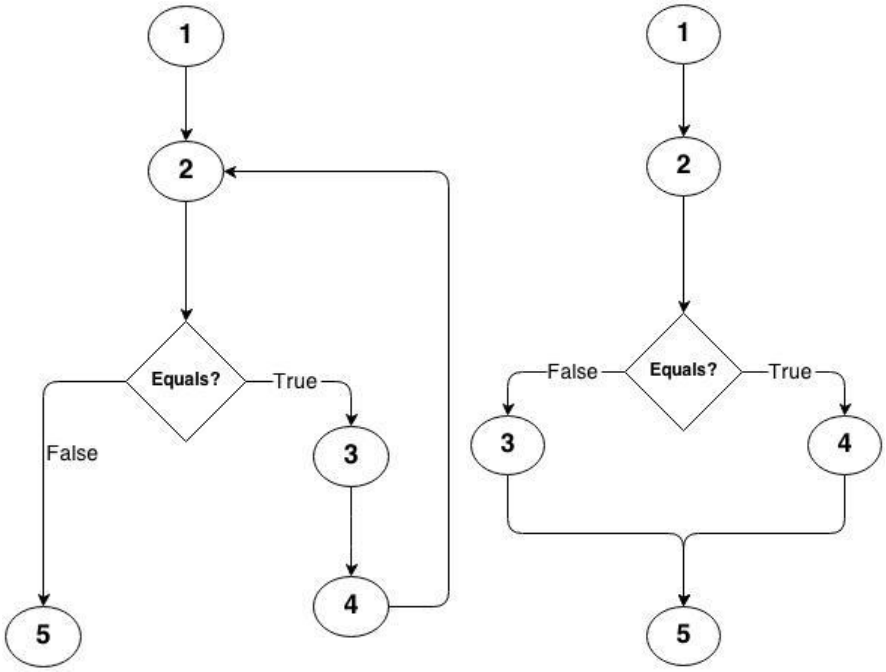
Four Families

Procedural

Overview

Procedural programming languages have existed for a long time, it was one of the first high level programming language paradigms used. Procedural programming is a subclass of Imperative programming adding modular subroutines. Imperative means 'to give orders, or instruction' and a program that is written in a procedural programming language is specified with step by step algorithms. As it is illustrated in the example 1:

Example 1:



All procedural languages have ways to express different types of data, control statements (including looping and branching) and ways to input and output data- languages support this paradigm by providing facilities for the ways to pass arguments, ways to distinguish different kinds of arguments, different kinds of functions (such as procedures, routines, and macros), and so on.

Procedural programming languages are usually the ones people start coding with. Writing code step by step helps beginners to get familiar with the fundamental control flow on a computer, as the languages are usually much lower level than other paradigms discussed.

In a paper on Learning and Teaching Programming written at University of Otago [5], section compares the procedural languages with Object-Oriented languages. Wiedenbeck et al. states "The distributed nature of control flow and function in an OO program may make it more difficult for novices to form a mental representation of the function and control flow of an OO program than of a corresponding procedural program…" [6]. As shown the majority of novices, or beginners in our case, performed worse in OO languages rather than in imperative programming languages. It is also mentioned that the problem domain was specifically designed for both paradigms thus making it equivalent.

Another study was done between procedural programming and OO programming at University of Jammu. [7] This study is focused on the performance of both programming styles in real-time embedded system's environment. It is shown in this study that imperative programming is more suitable to real-time systems. Firstly, OO languages like Java have background processes as

garbage collector which is resource consuming. However, in procedural programming languages this type of background process is optional, it isn't built-in into main libraries but rather could be found in its own independent library, thus focusing all their resources on executing main code. Second of all, procedural languages are more memory efficient. Because it doesn't have objects and encapsulations, it performs better in data memory access time and has a fewer number of instructions in setting or getting a value from variable than OO (see example 2). A major difference between procedural languages and Object Oriented languages is that procedure languages tend to separate the procedures from the data, whereas Object Oriented languages combine them in entities.

Example 2:

As you can see, in OO variables and functions are encapsulated into entities (classes):

```
int Sum(int a, int b)
	{
		return a+b;
	}


	int main()
	{
		int sum = Sum(5,7);
		return 0;
	}


class Summation()
	{
		private sum;


		public static void main (String [] args)
		{ int sum=Sum(5,7); }

		public int Sum(int a, int b)
		{ return a+b; }
	}
```

Since procedural languages are relatively low level compared to other languages, they are also used to write programs that work more closely to the operating system and hardware. Compilers and interpreters are written in procedural programming languages because of it. Some of these languages are even used for writing operating systems as they can call assembly code to operate hardware, or even operate it directly. As well, because of its memory efficiency, and speed this type of programming languages allow more resources to be spent performing other tasks. [8]

Examples

- Pascal, C, and COBOL are examples of procedural languages.

Benefits

- Excellent for general purpose programming
- Good level of control without having to know precise target CPU details- unlike low level languages (E.g. Assembly).
- Portable source code- uses a different compiler to target a different CPU. It means that converting code is done by CPU specific compiler providing us with code that doesn't need to change unlike low-level programming languages.
- Easier to debug as imperative paradigm is simple step by step programming. When debugging such code that doesn't make a lot of jumps to unsequenced lines makes it more easy to understand

Drawbacks

- Functions within programs are not flexible, making them difficult to reuse in other programs without modifications.
- Difficult implementation in fuzzy conditions as found in Artificial Intelligence applications.
- Since procedural languages don't have encapsulation, implementation of a complex program gets more challenging as complexity of its solution grows.
- Extremely difficult to solve high complexity problems compared to OO.

Object Oriented

Overview

Object Oriented languages are the result of an attempt to create a programming paradigm that resembles systems as they occur in the world in which we live. In better words, object orientation aims to facilitate understanding of code, make it very maintainable and easy to modify. The main feature of this language type is deliberately proposed in its name, objects and classes. A class is a defined type of object, for example "animal." The corresponding object to the "animal" class will be an instance of that class, for example lion, or tiger. Classes encapsulate states (usually through the use of saved variables) and methods that can be used to modify these states. Objects can interact between themselves, changing each other's states and accomplishing a task as the result of one or various state (variable) changes. [9] The major elements present in object oriented programming, as observed by Grady Brooch in his book "Object-Oriented Analysis and Design", are abstraction, encapsulation, modularity and hierarchy.

Abstraction: abstraction refers to the ability of denoting an object only by characteristics that distinguish it from other objects, according to the perspective of the viewer. Therefore if an object needs to use a specific different object to accomplish a task, the first object will know the second as a mere means to accomplish that task. At the same time the second objects could

have a different purpose in a third object's perspective. Both the first and third objects are using the second, but they have different perspectives on how the same object can be used.

This can be exemplified in the real world in a lot of ways. Suppose a computer is used by a family of three in a household. While this computer can carry out a number of tasks, each viewer (or user) will view it as a tool to accomplish their own tasks. For the father it is only useful for reading emails; from the mother's perspective, it is a tool for managing digital pictures, while from the daughter's perspective it is a means to access her Facebook account.

Encapsulation: encapsulation goes together with abstraction; they complement each other. In fact, because of their similarity, they are often mistakenly interpreted to be interchangeable. While abstraction is a deliberate ignorance of the whole behaviour of an object, encapsulation is the deliberate ignorance of how an object carries out its behaviour, or how it works. Therefore, because of encapsulation, one object does not need to be familiar with another object's internal system in order to use it.

Continuing with the real world family example, the girl who uses the computer to access Facebook does not need to be familiar with the intricacies of computer network communications in order to use a browser to access the internet. She knows that the computer can handle the complexity to carry out her task; therefore she doesn't need to understand how it works.

Modularity: modularity refers to the ability to separate a system into smaller, more understandable parts. By subdividing complex systems into smaller and less-complex groups of similar tasks and/or objects, a system is more easily developed, used, understood, maintained and modified. This is actually human behavior that was employed way before software was invented; whenever humans are tackling a difficult, complex task, their intuition forces them to work in small parts which will eventually completely accomplish the complex task. Decreased coupling is one of the main benefits from the use of modularity, meaning that the interdependence between different components is minimal.

Hierarchy: hierarchy works very closely with abstraction; it is actually a way to rank, or order, abstractions. This is mainly achieved through single inheritance, multiple inheritance and aggregation. Single and multiple inheritance are the redefinition or augmentation of an existing object by another object. An object in a first level of hierarchy might have very little complexity, but it might have other objects inheriting its behaviour or states and adding functionality to it. Single inheritance defines inheriting functionality from only one object, while multiple inheritance allows functionality to be merged from two unrelated objects into one.

This can be exemplified by levels of permissions in an online forum. A regular user can only post comments, while a moderator can post comments AND delete comments. The moderator inherits the functionality of the regular users and augments it.

Aggregation is the possibility of assigning objects as being part of another object that represents a collection. For example, a car can have a motor, four wheels and two doors. The latter objects are all aggregated to the car, which represents the collective.

Although these four features are the main characteristics of object-oriented programming, other features can also be found in object-orientation, such as: typing, concurrence, persistence, open recursion, dynamic dispatch, message passing and polymorphism. [10]

Examples

Java and C# are examples of object oriented languages.

Benefits

The main benefits of Object-Oriented languages are:

Understandability: it is easy to program with object-oriented languages, since they make use of real world concepts.

Maintainability: with understandability comes easiness to make modifications to existing software.

Changeability: Modularity and low coupling make it very easy to extend or modify functionality, compared to other languages.

Design Patterns: Since relationships between objects are very easily understood, patterns in relationships can often be found. These patterns can be reused to solve recurring problems in software development. [11]

Drawbacks

Lacks Normalization Rules: OO has nothing equivalent to relational Normal Form rules and guidelines to produce consistency and reduce known duplication.

Grouping functions and data: some people argue that because functions and data are extremely different in nature, they should not be encapsulated together. Keeping data grouped with functions inside classes fails the possibility of having cross-language sharing of protocols and data. Lastly, it also fails divide and conquer of data operations.

Logic

Overview

Logic programming, as its name may suggests, uses logic! Logic (specifically formal logic) was created to aid in human thought; to provide a structured means to reason and argue. When it came to developing languages for computers (which themselves are great with logic..), using logic as a basis was a natural adaption.

Logical programs contain a set of statements that define rules. A rule is either a fact or an inference over entities and relationships. Facts are assumed to be true, and inferences are conditionally true (logical if, then statements).

Consider the entities ‘0’ and ‘2’ with the rules:

(1) 0 is even

(2) (X + 2) is even if (X is even)

Here, **(1)** is a fact and **(2)** is an inference.

Programs are then typically used in interactive mode to answer queries (questions) to obtain results. A result is provided when a matching fact is found or when the query is proven inconsistent with the knowledge base (there is no set of rules in the knowledge base that supports the query).

Logic programs determine if a query is true or false by either:

- **goal-directed**: starting with the query and work backwards using inference rules to arrive at the facts
- **forward-reasoning**: start with the facts and work forwards by applying inference rules until arriving at the query

Logic programming was conceived with goal-directed search in mind, and has mostly stayed that way. The most popular logic language is Prolog which uses goal-directed search. Forward reasoning has other applications that involve finding new inferences when provided data, which can increase efficiencies.

Examples of logical programming languages

Prolog, Datalog.

eg. The above problem can be solved in **Prolog**, consisting of two rules, which identify if a given number is a positive, even integer:

```
even(0).                                    //(1) Fact
even(X) :- X > 0, Y is X-2, even(Y).   //(2) Inference
```

Given the following query:

```
?- even(4).
```

Prolog uses goal-directed search, so computations will start with even(4) and applies inference rules to arrive at a fact, namely even(0) and return true:

```
        ?-even(4).
call    even(4) ?       //initial call to knowledge base
call    even(2) ?       //recursive call with X = 2
call    even(0) ?       //recursive call with X = 0, matches even(0), return true.
```

Then because zero is even, even(2) is also even, so the initial query, even(4) is true.
QED.

If the query of even(3) is provided, then recursive calls end with even(-1), which does not match any rules, and false will be passed up through even(1) and even(3).

Utility

Declarative languages are open to being manipulated and transformed to be more efficient at solving the same problem, or with slight modification to solve a different problem.

Most known for its application in AI, logical programming solutions have found places in complex scheduling algorithms, warehouse problems, and database problems. [13]

Benefits

Logic programs can be used to express knowledge without caring about underlying implementation or hardware. Programs are understandable by people with non-computational backgrounds provided they have knowledge of first-order predicate calculus.

As mentioned above, logic programming can be used in scheduling algorithms, which are ‘timetabling’ problems; timetabling is known to be NP-complete, ie. no known algorithm solves the problem in polynomial time. [15]

Drawbacks

Including its limited uses, people don’t typically think of problems that give way to logical programming solutions. And if they do, a logical programming solution may not be thought of or not practical for the situation. (Think of how most of the problems people can think of are in NP, but the majority of problems that exist lie outside of NP)

Logic programs are generally slower than counterparts programmed in C or Java, but are easier to code.

Functional

Overview

Functional languages are a set of languages that concentrate their structure on creating “elegant yet powerful and general libraries of functions” [17]. Functional languages try to reflect the way people think mathematically. The main concept of functional programming is that functions are treated as first class object. A function can act like any other object, can be

passed to a function and can be returned from a function. What really makes functional programming exceptional from all others is the possibility of defining and manipulating the function from within other functions.

Another key concept in functional programming, functional languages rarely use side effects. What do we mean by “rarely use side effect”? It means that your functional, in addition to return value, does not change the state of the system. You programs behaviour becomes really predictable, which makes functional programming really attractive for software developers.

All functional languages can be divided in two groups: Pure and Impure. Pure functional languages are those that concentrate on creating function that has no side effects, output, and does not depend on any state except it is own. Some examples of pure functional programming languages are Haskell, Lisp. Impure functional languages are languages that do not follow strictly the functional paradigm. Some of them have features like state, exception handling, etc.

Functional programming languages support lazy evaluation, which means lazy evaluator will only evaluate an argument to a function if that argument’s value is needed to compute the overall result. [17]

Looping in functional programming is not possible, but it is compensated by large use of recursion. Not having side effects in functional programming makes the use of loop counter almost impossible, as they require constant change of the state. The main advantage of having recursion in pure functional language is that it can be implemented more efficiently.

Examples

This is the implementation of quicksort in Haskell. As you can see this code is fairly simple and easy to understand. It would take only 3 lines of code.

```
qsort :: [a] -> [a]
qsort [] = []
qsort (h:t) = qsort [y | y <- t, y < h ] ++ [h] ++ qsort [y | y <- t, y >= h]
```

The same code in, for example, object-oriented language would be a lot more complicated, it would require a lot more functions and more jumping in the code.

Utility

Functional programming is popular in academia but less popular in industry. However, recently functional programs have been used in industrial systems. As an example, the Erlang programming language which was developed by Swedish company, Ericsson, in 1980s and was

originally used to implement fault-tolerant telecommunications systems. Later it became popular in other companies including Nortel. [19] [20]

Haskell was first developed as a research language. [21] Nowadays, Haskell is widely used in companies like Google and Barclays Capital Quantitative Analytics Group. Google uses Haskell in some internal projects for internal IT infrastructure support. [22] Barclays Capital's Quantitative Analytics group is using Haskell to develop an embedded domain-specific functional language (called FPF) which is used to specify exotic equity derivatives. These derivatives, which are naturally best described in terms of mathematical functions, and constructed compositionally, map well to being expressed in an embedded functional language. [23]

Benefits

The advantages of functional programming are often summarized as follows. As functional programs don't contain assignment statements, so variables, once given a value, never change. Because of this, they do not have any side-effects at all. A function call will not do anything except computing the result. This eliminates a major source of bugs. In addition to this, order of execution is not important - since no side-effect can change the value of an expression, thus it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa - that is, programs are "referentially transparent". This freedom helps make functional programs more tractable mathematically than their conventional counterparts. [18]

In order to call two functions one can do g (f input), where the input for the g will be output of the function f. On the other hand, this might be implemented as storing the output of the function f into the temporary variable and use this temporary variable as the input to the function g. The problem with this is that the temporary file might occupy so much memory that it might be impractical to attach these two functions in this way. Functional programs have a solution to this problem. The two functions f and g are run together in strict synchronization. The function f will start once g tries to read some input, and only runs for long enough to deliver the output g is trying to read. Next f is suspended and g is run until it tries to read another input. If g terminates without reading all of f's output then f is aborted. Function f can even be a non-terminating program, producing an infinite amount of output, since it will be terminated forcibly as soon as g is finished. This allows termination conditions to be separated from loop bodies. Because, this method of evaluation runs function f as little as possible, it is called "lazy evaluation". While some other systems allow programs to be run together in this manner, only functional languages use lazy evaluation uniformly for every function call, allowing any part of a program to be modularised in this way. Lazy evaluation is perhaps the most powerful tool for modularisation in the functional programmer's repertoire.[18] Because of using lazy evaluation in functional programming performance increases by avoiding unnecessary calculations and error conditions in evaluating compound expressions.

We can add the example stated above to the advantages of functional programming. Compared to object-oriented or procedural programming languages, the implementation of quicksort in functional programming is more elegant and shorter.

How can we be sure that our program works correctly? Traditional way to show a program works correctly is to test the program. But the famous computer scientist Edsger Dijkstra said “Testing shows the presence, not the absence of bugs”. The other way to show the program works correctly is formal proof. We can use proofs for most programming languages, but it is a lot easier for functional languages because of statelessness, and recursion as the primary tool; proofs for recursion are simpler.

Drawbacks

As pure functional programming does not allow any side effects, it is hard to write useful software without I/O. Therefore most people would never get further than just calculation of single output from single input.

Functional programming is also really hard to read. Comparing to Object Oriented languages, functional programming is not similar to English or any other languages.

Conclusion

On the highest order, languages may be divided into imperative and declarative, and further into procedural and objected oriented, and functional and logical, respectively. Many modern languages are comprised of multiple paradigms; Java and Oz being notable examples of languages supporting almost every paradigm mentioned in this paper.

Imperative languages tend to be sought for their control over the base hardware, and their intrinsic understandability for learning the syntax and semantics. Declarative languages, while more difficult read and learn, requiring much more complex compilers or interpreters, and generally being less efficient in the use of system resources in sequential operations, are often shorter in length due to compressed complexity and safety in parallel operation due to the lack of explicit state and side effects.

Compilers/Interpreters for more esoteric languages such as Oz only exist for a single platform and operating system; limiting their use on different platforms versus more widely used languages like C that have a compiler for most hardware systems and operating systems.

Ultimately, the choice of paradigm depends on the problem and what languages a programmer is comfortable with. But there are situations where the problem might favour a paradigm X, and well written code in a paradigm Y would be a superior solution to a poorly written solution in paradigm X. Vice versa; the primary importance in any software project is that the software meets the requirements; beyond that the readability, maintainability and modifiability must take precedence over any idea of the “superior solution”.

References

[1] Edmonds, B., "Declarative and Imperative Computing Paradigms". Retrieved February, 2014 Available: http://cfpm.org/~bruce/logreas/logreas_7.html

[2] Nilsson, U., et all, "LOGIC, PROGRAMMING AND PROLOG (2ED)". Retrieved February, 2013 Available: http://www.cs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/DataFlowProgrammingLanguages.pdf

[3] Ford, N., "Functional thinking: Why functional programming is on the rise". Retrieved February, 2013 Available: http://www.ibm.com/developerworks/java/library/j-ft20/index.html

[4] Johnston, J. M., et al, "Advances in Dataflow Programming Languages". Retrieved February, 2014 Available: http://www.cs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/DataFlowProgrammingLanguages.pdf

[5] Robins et al, "Learning and Teaching Programming: A Review and Discussion". Retrieved February, 2014 Available: http://home.cc.gatech.edu/csed/uploads/2/robins03.pdf

[6] Wiedenbeck, et al, A comparison of the comprehension of object-oriented and procedural programs by novice programmers, Interacting With Computers, vol 3, no 11, p.61 - 83

[7] Dutt, et al, "Object Oriented Vs Procedural Programming in Embedded Systems". Retrieved February, 2014 Available: http://home.cc.gatech.edu/csed/uploads/2/robins03.pdf

[8] Flynn, I. M., "Procedural Languages". Retrieved February, 2014 Available: http://go.galegroup.com.ezproxy.lib.ucalgary.ca/ps/i.do?id=GALE%7CCX3401200261&v=2.1&u=ucalgary&it=r&p=GVRL&sw=w&asid=b1009d3562333372396cf25468c19ecc

[9] Wirth, N, "Good Ideas, Through the Looking Glass". Retrieved February, 2014 Available: http://people.inf.ethz.ch/~wirth/Articles/GoodIdeas_origFig.pdf

[10] Brooch, G , "Objected-Oriented Analysis and Design with Applications". Retrieved February, 2014 Available: http://www.cvauni.edu.vn/imgupload_dinhkem/file/PTTKHT/Object-oriented-analysis-and-design-with-applications-2nd-edition.pdf

[11] Gamma, et al, "Design Patterns: Elements of Reusable Object-Oriented Software". Retrieved February, 2014 Available: http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf

[12] Clocksin, F, Mellish, C. "*Programming in Prolog - Using the ISO Standard, Fifth Edition*". Berlin: Springer-Verla. 1-8. 2003.

[13] Hancox, P., "Logic Programming with Prolog". Retrieved February, 2014 Available: http://www.cs.bham.ac.uk/~pjh/prolog_course/sem223_se.html

[14] Pfenning, F. "Logic Programming". Retrieved February, 2014 Available: http://www.cs.cmu.edu/~fp/courses/lp/lectures/01-lp.pdf

[15] Cooper, TB., and Kingston, JH., "The Complexity of Timetable Construction Problems". in the Practice and Theory of Automated Timetabling, ed. EK Burke and P Ross,

pp. 283-295, Springer-Verlag (Lecture Notes in Computer Science), 1996.

[17] Thompson, S., “*Haskell the craft of functional programming*”. Addison Wesley Professional, 2011.

[18] Hughes, J., “Why Functional Programming Matters”. Retrieved February, 2014 Available: http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf

[19] Armstrong, J., “A History of Erlang”. Retrieved February, 2014 Available: http://webcem01.cem.itesm.mx:8005/erlang/cd/downloads/hopl_erlang.pdf

[20] Larson, J., Erlang for Concurrent Programming, Communications of the ACM, vol 52, p.58 - 52

[21] Hudak, et al. “A History of Haskell”. Retrieved February, 2014 Available: http://www.iro.umontreal.ca/~monnier/2035/history.pdf

[22] Pop, I., “Experience Report: Haskell as a Reagent, Results and Observations on the Use of Haskell in a Python Project”. Retrieved February, 2014 Available: http://k1024.org/~iusty/papers/icfp10-haskell-reagent.pdf

[23] Frankau, et al., “Commercial Uses: Going functional on exotic trades.” Retrieved February, 2014 Available: http://www.dmst.aueb.gr/dds/pubs/jrnl/2008-JFP-ExoticTrades/html/FSNB08.pdf