COMP6411 COMPARATIVE STUDY OF PROGRAMMING LANGUAGES

Part 2: Programming Paradigms

Learning Objectives

- Learn about different programming paradigms
  - Concepts and particularities
  - Advantages and drawbacks
  - Application domains

Introduction

- A programming paradigm is a fundamental style of computer programming.
- Compare with a software development methodology, which is a style of solving specific software engineering problems.
- Different methodologies are more suitable for solving certain kinds of problems or applications domains.
- Same for programming languages and paradigms.
- Programming paradigms differ in:
  - the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints, etc.)
  - the steps that compose a computation (assignation, evaluation, data flow, control flow, etc.).

Introduction

- Some languages are designed to support one particular paradigm
  - Smalltalk supports object-oriented programming
  - Haskell supports functional programming
- Other programming languages support multiple paradigms
  - Object Pascal, C++, C#, Visual Basic, Common Lisp, Scheme, Perl, Python, Ruby, Oz and F#.
- The design goal of multi-paradigm languages is to allow programmers to use the best tool for a job, admitting that no one paradigm solves all problems in the easiest or most efficient way.

Introduction

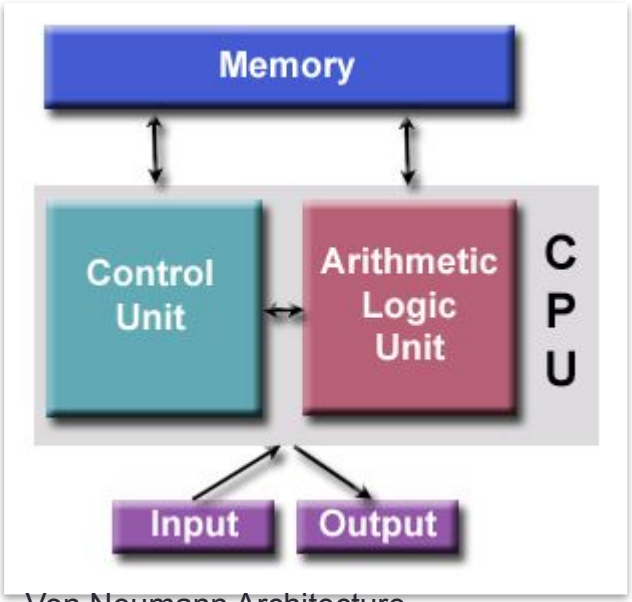
- A programming paradigm can be understood as an **abstraction of a computer system**, for example the von Neumann model used in traditional sequential computers.
- For **parallel computing**, there are many possible models typically reflecting different ways processors can be interconnected to communicate and share information.
- In **object-oriented programming**, programmers can think of a program as a collection of interacting objects, while in **functional programming** a program can be thought of as a sequence of stateless function evaluations.
- In **process-oriented programming,** programmers think about applications as sets of concurrent processes acting upon shared data structures.

PROCESSING PARADIGMS

Processing Paradigms

- A programming paradigm can be understood as an **abstraction** of a computer system, who is based on a certain **processing model** or paradigm.
- Nowadays, the prevalent computer processing model used is the **von Neumann model**, invented by John von Neumann in 1945, influenced by Alan Turing's "Turing machine".
  - Data and program are residing in the **memory**.
  - **Control unit** coordinates the components sequentially following the program's instructions.
  - **Arithmetic Logical Unit** performs the calculations.
  - Input/output provide interfaces to the exterior.
- The **program** and its **data** are what is **abstracted** in a programming language and translated into machine code by the compiler/interpreter.

John von Nuemann



Von Neumann Architecture

Processing Paradigms

- When programming computers or systems with many processors, parallel or process-oriented programming allows programmers to think about applications as sets of concurrent processes acting upon shared data structures.
- There are many possible models typically reflecting different ways processors can be interconnected.
- The most common are based on shared memory, distributed memory with message passing, or a hybrid of the two.
- Most parallel architectures use multiple von Neumann machines as processing units.

Processing Paradigms

- Specialized programming languages have been designed for parallel/concurrent computing.
- Distributed computing relies on several sequential computers interconnected to solve a common problem. Such systems rely on interconnection middleware for communication and information sharing.
- Other processing paradigms were invented that went away from the von Neumann model, for example:
  - LISP machines
  - Dataflow machines

PROGRAMMING PARADIGMS

LOW-LEVEL PROGRAMMING PARADIGM

Low level

- Initially, computers were **hard-wired** or **soft-wired** and then later programmed using **binary code** that represented control sequences fed to the computer CPU.

- This was difficult and error-prone. Programs written in binary are said to be written in machine code, which is a very low-level programming paradigm. Hard-wired, soft-wired, and binary programming are considered **first generation** languages.

Low level

- To make programming easier, assembly languages were developed.
- These replaced machine code functions with **mnemonics** and memory addresses with **symbolic labels**.
- Assembly language programming is considered a low-level paradigm although it is a '**second generation**' paradigm.
- Assembly languages of the 1960s eventually supported **libraries** and quite sophisticated conditional **macro** generation and **pre-processing** capabilities.
- They also supported **modular programming** features such as subroutines, external variables and common sections (globals), enabling significant **code re-use** and isolation from hardware specifics via use of logical operators.

Low level

- Assembly was, and still is, used for **time-critical** systems and frequently in **embedded systems**.
- Assembly programming can directly take advantage of a specific computer architecture and, when written properly, leads to **highly optimized code**.
- However, it is bound to this architecture or processor and thus suffers from **lack of portability**.
- Assembly languages have **limited abstraction capabilities**, which makes them unsuitable to develop large/complex software.

PROCEDURAL PROGRAMMING PARADIGM

Procedural programming

- Often thought as a synonym for **imperative programming**.
- Specifying the **steps** the program must take to reach the desired **state**.
- Based upon the concept of the **procedure call**.
- Procedures, also known as routines, subroutines, methods, or functions that contain a series of computational steps to be carried out.
- Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

- A procedural programming language provides a programmer a means to define precisely each step in the performance of a task. The programmer knows what is to be accomplished and provides through the language step-by-step instructions on how the task is to be done.
- Using a procedural language, the programmer specifies **language statements** to perform a **sequence of algorithmic steps**.

Procedural programming

- Possible benefits:
  - Often a better choice than simple sequential or unstructured programming in many situations which involve moderate complexity or require significant ease of maintainability.
  - The ability to re-use the same code at different places in the program without copying it.
  - An easier way to keep track of program flow than a collection of "GOTO" or "JUMP" statements (which can turn a large, complicated program into spaghetti code).
  - The ability to be strongly modular or structured.
- The main benefit of procedural programming over first- and second-generation languages is that it allows for **modularity**, which is generally desirable, especially in large, complicated programs.
- Modularity was one of the earliest **abstraction** features identified as desirable for a programming language.

Procedural programming

- **Scoping** is another abstraction technique that helps to keep procedures strongly modular.
- It prevents a procedure from accessing the variables of other procedures (and vice-versa), including previous instances of itself such as in recursion.
- Procedures are convenient for making pieces of code written by different people or different groups, including through programming **libraries**.
  - specify a simple interface
  - self-contained information and algorithmics
  - reusable piece of code

Procedural programming

- The focus of procedural programming is to break down a programming task into a collection of **variables**, **data structures**, and **subroutines**, whereas in object-oriented programming it is to break down a programming task into objects with each "object" encapsulating its own data and methods (subroutines).

- The most important distinction is whereas procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together so an "object" operates on its "own" data structure.

Procedural programming

- The earliest imperative languages were the machine languages of the original computers. In these languages, instructions were very simple, which made hardware implementation easier, but hindered the creation of complex programs.
- FORTRAN (1954) was the first major programming language to remove through **abstraction** the obstacles presented by machine code in the creation of complex programs.
- FORTRAN was a compiled language that allowed named variables, complex expressions, subprograms, and many other features now common in imperative languages.
- In the late 1950s and 1960s, ALGOL was developed in order to allow mathematical algorithms to be more easily expressed.
- In the 1970s, Pascal was developed by Niklaus Wirth, and C was created by Dennis Ritchie.
- For the needs of the United States Department of Defense, Jean Ichbiah and a team at Honeywell began designing Ada in 1978.

OBJECT-ORIENTED PROGRAMMING PARADIGM

Object-oriented programming

- Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures encapsulating data fields and procedures together with their interactions – to design applications and computer programs.
- Associated programming techniques may include features such as data **abstraction**, **encapsulation**, **modularity**, **polymorphism**, and **inheritance**.
- Though it was invented with the creation of the **Simula** language in 1965, and further developed in **Smalltalk** in the 1970s, it was not commonly used in mainstream software application development until the early 1990s.
- Many modern programming languages now support OOP.

OOP concepts: class

- A class defines the abstract characteristics of a thing (object), including that thing's **characteristics** (its attributes, fields or properties) and the thing's **behaviors** (the operations it can do, or methods, operations or functionalities).
- One might say that a class is a blueprint or factory that describes the nature of something.
- Classes provide **modularity** and **structure** in an object-oriented computer program.
- A class should typically be recognizable to a non-programmer familiar with the **problem domain**, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation).
- Collectively, the properties and methods defined by a class are called its **members**.

OOP concepts: object

- An object is an individual of a class created at run-time trough object **instantiation** from a class.
- The set of values of the attributes of a particular object forms its **state**. The object consists of the **state** and the **behavior** that's defined in the object's class.
- The object is instantiated by implicitly calling its constructor, which is one of its member functions responsible for the creation of instances of that class.

OOP concepts: attributes

- An **attribute**, also called data member or member variable, is the data encapsulated within a class or object.
- In the case of a regular field (also called **instance variable**), for each instance of the object there is an instance variable.
- A static field (also called **class variable**) is one variable, which is shared by all instances.
- Attributes are an object's variables that, upon being given values at instantiation (using a **constructor**) and further execution, will represent the state of the object.
- A class is in fact a data structure that may contain different fields, which is defined to contain the procedures that act upon it. As such, it represents an **abstract data type**.
- In pure object-oriented programming, the attributes of an object are local and cannot be seen from the outside. In many object-oriented programming languages, however, the attributes may be accessible, though it is generally considered bad design to make data members of a class as externally visible.

OOP concepts: method

- A **method** is a subroutine that is exclusively associated either with a class (in which case it is called a **class method** or a static method) or with an object (in which case it is an **instance method**).
- Like a subroutine in procedural programming languages, a method usually consists of a sequence of programming statements to perform an action, a set of input parameters to customize those actions, and possibly an output value (called the return value).
- Methods provide a mechanism for accessing and manipulating the encapsulated state of an object.
- Encapsulating methods inside of objects is what distinguishes object-oriented programming from procedural programming.

OOP concepts: method

- ***instance*** methods are associated with an object
- ***class*** or ***static*** methods are associated with a class.
- The object-oriented programming paradigm intentionally favors the use of methods for each and every means of access and change to the underlying data:
  - **Constructors:** Creation and initialization of the state of an object. Constructors are called automatically by the run-time system whenever an object declaration is encountered in the code.
  - **Retrieval and modification of state:** accessor methods are used to access the value of a particular attribute of an object. Mutator methods are used to explicitly change the value of a particular attribute of an object. Since an object's state should be as hidden as possible, accessors and mutators are made available or not depending on the information hiding involved and defined at the class level
  - **Service-providing:** A class exposes some "service-providing" methods to the exterior, who are allowing other objects to use the object's functionalities. A class may also define private methods who are only visible from the internal perspective of the object.
  - **Destructor:** When an object goes out of scope, or is explicitly destroyed, its destructor is called by the run-time system. This method explicitly frees the memory and resources used during its execution.

OOP concepts: method

- The difference between procedures in general and an object's method is that the method, being associated with a particular object, may access or modify the data private to that object in a way consistent with the intended behavior of the object.
- So rather than thinking "a procedure is just a sequence of commands", a programmer using an object-oriented language will consider a method to be "**an object's way of providing a service**". A method call is thus considered to be a request to an object to perform some task.
- Method calls are often modeled as a means of passing a message to an object. Rather than directly performing an operation on an object, a message is sent to the object telling it what it should do. The object either complies or raises an exception describing why it cannot do so.
- Smalltalk used a real "**message passing**" scheme, whereas most object-oriented languages use a standard "function call" scheme for message passing.
- The message passing scheme allows for **asynchronous** function calls and thus **concurrency**.

OOP concepts: inheritance

- Inheritance is a way to compartmentalize and **reuse** code by creating collections of attributes and behaviors (classes) which can be based on previously created classes.
- The new classes, known as **subclasses** (or derived classes), inherit attributes and behavior of the pre-existing classes, which are referred to as superclasses (or ancestor classes). The inheritance relationships of classes gives rise to a hierarchy.
- **Multiple inheritance** can be defined whereas a class can inherit from more than one superclass. This leads to a much more complicated definition and implementation, as a single class can then inherit from two classes that have members bearing the same names, but yet have different meanings.
- **Abstract inheritance** can be defined whereas abstract classes can declare member functions that have no definitions and are expected to be defined in all of its subclasses.

OOP concepts: abstraction

- **Abstraction** is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem.

- For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

- Object-oriented programming provides **abstraction** through **composition** and **inheritance**.

OOP concepts: encapsulation and information hiding

- **Encapsulation** refers to the bundling of data members and member functions inside of a common “box”, thus creating the notion that an object contains its state as well as its functionalities
- **Information hiding** refers to the notion of choosing to either expose or hide some of the members of a class.
- These two concepts are often misidentified. Encapsulation is often understood as including the notion of information hiding.
- Encapsulation is achieved by specifying which classes may use the members of an object. The result is that each object exposes to any class a certain interface — those members accessible to that class.
- The reason for encapsulation is to prevent clients of an interface from depending on those parts of the implementation that are likely to change in the future, thereby allowing those changes to be made more easily, that is, without changes to clients.
- It also aims at preventing unauthorized objects to change the state of an object.

OOP concepts: encapsulation and information hiding

- Members are often specified as **public**, **protected** or **private**, determining whether they are available to all classes, sub-classes or only the defining class.

- Some languages go further:

  - Java uses the default access modifier to restrict access also to classes in the same package

  - C# and VB.NET reserve some members to classes in the same assembly using keywords **internal** (C#) or **friend** (VB.NET)

  - Eiffel and C++ allow one to specify which classes may access any member of another class (C++ friends)

  - Such features are basically overriding the basic information hiding principle, greatly complexify its implementation, and create confusion when used improperly

OOP concepts: polymorphism

- Polymorphism is the ability of objects belonging to **different type**s to respond to method, field, or property calls of the **same name**, each one according to an appropriate **type-specific behavior**.
- The programmer (and the program) does not have to know the exact type of the object at compile time. The exact behavior is determined at run-time using a run-time system behavior known as **dynamic binding**.
- Such polymorphism allows the programmer to treat derived class members just like their parent class' members.
- The different objects involved only need to present a **compatible interface** to the clients. That is, there must be public or internal methods, fields, events, and properties with the same name and the same parameter sets in all the superclasses, subclasses and interfaces.
- In principle, the object types may be unrelated, but since they share a common interface, they are often implemented as subclasses of the same superclass.

OOP concepts: polymorphism

- A method or operator can be abstractly applied in many different situations. If a Dog is commanded to speak(), this may elicit a bark(). However, if a Pig is commanded to speak(), this may elicit an oink(). They both inherit speak() from Animal, but their derived class methods override the methods of the parent class. This is **overriding polymorphism**.

- **Overloading polymorphism** is the use of one method signature, or one operator such as "+", to perform several different functions depending on the implementation. The "+" operator, for example, may be used to perform integer addition, float addition, list concatenation, or string concatenation. Any two subclasses of Number, such as Integer and Double, are expected to add together properly in an OOP language. The language must therefore overload the addition operator, "+", to work this way. This helps improve code readability. How this is implemented varies from language to language, but most OOP languages support at least some level of overloading polymorphism.

OOP concepts: polymorphism

- Many OOP languages also support **parametric polymorphism**, where code is written without mention of any specific type and thus can be used transparently with any number of new types. C++ templates and Java Generics are examples of such parameteric polymorphism.

- The use of pointers to a superclass type later instantiated to an object of a subclass is a simple yet powerful form of polymorhism, such as used un C++.

OOP: Languages

- Simula (1967) is generally accepted as the first language to have the primary features of an object-oriented language. It was created for making simulation programs, in which what came to be called objects were the most important information representation.

- Smalltalk (1972 to 1980) is arguably the canonical example, and the one with which much of the theory of object-oriented programming was developed.

OOP: Languages

- Concerning the degree of object orientation, following distinction can be made:
  - Languages called "pure" OO languages, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc. They were designed specifically to facilitate, even enforce, OO methods. Examples: Smalltalk, Eiffel, Ruby, JADE.
  - Languages designed mainly for OO programming, but with some procedural elements. Examples: C++, C#, Java, Scala, Python.
  - Languages that are historically procedural languages, but have been extended with some OO features. Examples: VB.NET (derived from VB), Fortran 2003, Perl, COBOL 2002, PHP.
  - Languages with most of the features of objects (classes, methods, inheritance, reusability), but in a distinctly original form. Examples: Oberon (Oberon-1 or Oberon-2).
  - Languages with abstract data type support, but not all features of object-orientation, sometimes called object-based languages. Examples: Modula-2, Pliant, CLU.

OOP: Variations

- There are different ways to view/implement/instantiate objects:

- **Prototype-based**
  - objects - classes + delegation

  - no classes
  - objects are a set of members
  - create ex nihilo or using a prototype object (“cloning”)

- Hierarchy is a "containment" based on how the objects were created using prototyping. This hierarchy is defined using the **delegation principle** can be changed as the program executes prototyping operations.

- examples: ActionScript, JavaScript, JScript, Self, Object Lisp

OOP: Variations

- **object-based**
  - objects + classes - inheritance
  - classes are declared and objects are instantiated
  - no inheritance is defined between classes
  - No polymorphism is possible
- example: VisualBasic

OOP: Variations

- **object-oriented**
  - objects + classes + inheritance + polymorphism
  - This is recognized as true object-orientation
  - examples: Simula, Smalltalk, Eiffel, Python, Ruby, Java, C++, C#, etc...

DECLARATIVE PROGRAMMING PARADIGM

Declarative Programming

- General programming paradigm in which programs express the logic of a computation without describing its control flow.
- Programs describe **what** the computation should accomplish, rather than **how** it should accomplish it.
- Typically avoids the notion of variable holding **state**, and function **side-effects**.
- Contrary to imperative programming, where a program is a series of steps and state changes describing how the computation is achieved.
- Includes diverse languages/subparadigms such as:
  - Database query languages (e.g. SQL, Xquery)
  - XSLT
  - Makefiles
  - Constraint programming
  - Logic programming
  - Functional programming

FUNCTIONAL PROGRAMMING PARADIGM

Functional Programming

- Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and **avoids state changes** and **mutable data**.
- It emphasizes the **application of functions**, in contrast to the imperative programming style, which emphasizes **changes in state**.
- Programs written using the functional programming paradigm are much more easily representable using mathematical concepts, and thus it is much more easy to mathematically reason about functional programs than it is to reason about programs written in any other paradigm.

Functional Programming: History

- Functional programming has its roots in the **lambda calculus**, a formal system developed in the 1930s to investigate function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.

- LISP was the first operational functional programming language.

- Up to this day, functional programming has not been very popular except for a restricted number of application areas, such as artificial intelligence.

- John Backus presented the FP programming language in his 1977 Turing Award lecture "*Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs*".

Functional Programming: History

- In the 1970s the **ML** programming language was created by **Robin Milner** at the University of Edinburgh, and **David Turner** developed initially the language **SASL** at the University of St. Andrews and later the language **Miranda** at the University of Kent.

- ML eventually developed into several dialects, the most common of which are now **Objective Caml**, **Standard ML**, and **F#**.

- Also in the 1970s, the development of the **Scheme** programming language (a partly-functional dialect of Lisp), as described in the influential "*Lambda Papers*" and the 1985 textbook "*Structure and Interpretation of Computer Programs*", brought awareness of the power of functional programming to the wider programming-languages community.

- The **Haskell** programming language was released in the late 1980s in an attempt to gather together many ideas in functional programming research.

Functional Programming

- Functional programming languages, especially purely functional ones, have largely been emphasized in academia rather than in commercial software development.
- However, prominent functional programming languages such as **Scheme**, **Erlang**, **Objective Caml**, and **Haskell** have been used in industrial and commercial applications by a wide variety of organizations.
- Functional programming also finds use in industry through domain-specific programming languages like **R** (statistics), **Mathematica** (symbolic math), **J** and **K** (financial analysis), **F#** in Microsoft .NET and **XSLT** (XML).
- Widespread declarative domain-specific languages like **SQL** and **Lex/Yacc**, use some elements of functional programming, especially in eschewing mutable values. **Spreadsheets** can also be viewed as functional programming languages.

Functional Programming

- In practice, the difference between a mathematical function and the notion of a "function" used in imperative programming is that imperative functions can have **side effects**, changing the value of already calculated **variables**.
- Because of this they lack **referential transparency**, i.e. the same language expression can result in different values at different times depending on the state of the executing program.
- Conversely, in functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument x will produce the same result f(x) both times.
- Eliminating side-effects can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional Programming: Higher-Order Functions

- Most functional programming languages use **higher-order functions**, which are functions that can either take other functions as arguments or return functions as results.
- The differential operator *d/dx* that produces the derivative of a function f is an example of this in calculus.
- Higher-order functions are closely related to **functions as first-class citizen**, in that higher-order functions and first-class functions both allow functions as arguments and results of other functions.
- The distinction between the two is subtle: "higher-order" describes a mathematical concept of functions that operate on other functions, while "first-class" is a computer science term that describes programming language entities that have no restriction on their use (thus first-class functions can appear anywhere in the program that other first-class entities like numbers can, including as arguments to other functions and as their return values).

Functional Programming: Pure Functions

- Purely functional functions (or expressions) have **no memory** or **side effects**. They represent a function whose valuation depends only on the value of the parameters they are given. This means that pure functions have several useful properties, many of which can be used to optimize the code:
  - If the result of a pure expression is not used, it can be removed without affecting other expressions.
  - If a pure function is called with parameters that cause no side-effects, the result is constant with respect to that parameter list (**referential transparency**), i.e. if the pure function is again called with the same parameters, the same result will be returned (this can enable **caching** optimizations).
  - If there is no data dependency between two pure expressions, then they can be evaluated in any order, or they can be **performed in parallel** and they cannot interfere with one another (in other terms, the evaluation of any pure expression is thread-safe and enables parallel execution).

Functional Programming: Pure Functions

- If the entire language does not allow side-effects, then any evaluation strategy can be used; this gives the compiler freedom to **reorder** or **combine** the evaluation of expressions in a program. This allows for much more **freedom in optimizing the evaluation**.

- The notion of pure function is central to **code optimization** in compilers, even for procedural programming languages.

- While most compilers for imperative programming languages can detect pure functions, and perform common-subexpression elimination for pure function calls, they cannot always do this for pre-compiled libraries, which generally do not expose this information, thus preventing optimizations that involve those external functions.

- Some compilers, such as gcc, add extra keywords for a programmer to explicitly mark external functions as pure, to enable such optimizations. Fortran 95 allows functions to be designated "pure" in order to allow such optimizations.

Functional Programming: Recursion

- Iteration in functional languages is usually accomplished via **recursion**.
- Recursion may require maintaining a stack, and thus may lead to **inefficient memory consumption**, but tail recursion can be recognized and optimized by a compiler into the same code used to implement iteration in imperative languages.
- The Scheme programming language standard requires implementations to recognize and optimize tail recursion.
- Tail recursion optimization can be implemented by transforming the program into **continuation passing style** during compilation, among other approaches.
- Common patterns of recursion can be factored out using higher order functions, catamorphisms and anamorphisms, which "folds" and "unfolds" a recursive function call nest.
- Using such advanced techniques, recursion can be implemented in an efficient manner in functional programming languages.

Functional Programming: Eager vs. Lazy Evaluation

- Functional languages can be categorized by whether they use strict (**eager**) or non-strict (**lazy**) evaluation, concepts that refer to how function arguments are processed when an expression is being evaluated. Under strict evaluation, the evaluation of any term containing a failing subterm will itself fail. For example, the expression

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

- will fail under eager evaluation because of the division by zero in the third element of the list. Under lazy evaluation, the length function will return the value 4 (the length of the list), since evaluating it will not attempt to evaluate the terms making up the list.

- Eager evaluation fully evaluates function arguments before invoking the function. Lazy evaluation does not evaluate function arguments unless their values are required to evaluate the function call itself.

- The usual implementation strategy for lazy evaluation in functional languages is **graph reduction**. Lazy evaluation is used by default in several pure functional languages, including **Miranda**, **Clean** and **Haskell**.

Functional Programming: Type Inference

- Especially since the development of **Hindley–Milner type inference** in the 1970s, functional programming languages have tended to use typed lambda calculus, as opposed to the untyped lambda calculus used in Lisp and its variants (such as Scheme).

- **Type inference**, or implicit typing, refers to the ability to deduce automatically the type of the values manipulated by a program. It is a feature present in some strongly statically typed languages.

- The presence of strong compile-time type checking makes programs more **reliable**, while type inference frees the programmer from the need to **manually declare** types to the compiler.

- Type inference is often characteristic of — but not limited to — functional programming languages in general. Many imperative programming languages have adopted type inference in order to improve type safety.

Functional Programming: In Non-functional Languages

- It is possible to employ a functional style of programming in languages that are not traditionally considered functional languages.
- Some non-functional languages have borrowed features such as **higher-order functions**, and **list comprehensions** from functional programming languages. This makes it easier to adopt a functional style when using these languages.
- Functional constructs such as higher-order functions and lazy lists can be obtained in C++ via libraries, such as in FC++.
- In C, function pointers can be used to get some of the effects of higher-order functions.
- Many object-oriented design patterns are expressible in functional programming terms: for example, the **Strategy** pattern dictates use of a higher-order function, and the **Visitor** pattern roughly corresponds to a catamorphism, or fold.

REFLECTIVE PROGRAMMING PARADIGM

Reflective Programming

- Reflection is the process by which a computer program can **observe** and **modify** its own structure and behavior at runtime.
- In most computer architectures, program instructions are stored as data - hence the distinction between instruction and data is merely a matter of how the information is treated by the computer and programming language.
- Normally, instructions are executed and data is processed; however, in some languages, programs can also treat instructions as data and therefore make reflective modifications.
- Reflection is most commonly used in high-level virtual machine programming languages like Smalltalk and scripting languages, and less commonly used in manifestly typed and/or statically typed programming languages such as Java, C, ML or Haskell.

Reflective Programming

- Reflection-oriented programming includes **self-examination**, **self-modification**, and **self-replication**.
- Ultimately, reflection-oriented paradigm aims at **dynamic program modification**, which can be determined and executed at runtime.
- Some imperative approaches, such as procedural and object-oriented programming paradigms, specify that there is an exact predetermined sequence of operations with which to process data.
- The reflection-oriented programming paradigm, however, adds that program instructions can be modified dynamically at runtime and invoked in their modified state.
- That is, the program architecture itself can be decided at runtime based upon the data, services, and specific operations that are applicable at runtime.

Reflective Programming

- Reflection can be used for observing and/or modifying program execution at runtime. A reflection-oriented program component can monitor the execution of an enclosure of code and can modify itself according to a desired goal related to that enclosure. This is typically accomplished by dynamically assigning program code at runtime.
- Reflection can thus be used to **adapt a given program to different situations dynamically**.
- Reflection-oriented programming almost always requires additional knowledge, framework, relational mapping, and object relevance in order to take advantage of this much more generic code execution mode.
- It thus requires the translation process to retain in the executable code much of the higher-level information present in the source code, thus leading to more bloated executables.
- However, in cases where the language is interpreted, much of this information is already kept for the interpreter to function, so not much overhead is required in these cases.

Reflective Programming

- A language supporting reflection provides a number of features available at runtime that would otherwise be very obscure or impossible to accomplish in a lower-level language. Some of these features are the abilities to:
  - Discover and modify source code constructions (such as code blocks, classes, methods, protocols, etc.) as a first-class object at runtime.
  - Convert a string matching the symbolic name of a class or function into a reference to or invocation of that class or function.
  - Evaluate a string as if it were a source code statement at runtime.

Reflective Programming

- Compiled languages rely on their runtime system to provide information about the source code.
- A compiled **Objective-C** executable, for example, records the names of all methods in a block of the executable, providing a table to correspond these with the underlying methods (or selectors for these methods) compiled into the program.
- In a compiled language that supports runtime creation of functions, such as Common Lisp, the runtime environment must include a compiler or an interpreter.
- Programming languages that support reflection typically include dynamically typed languages such as Smalltalk; scripting languages such as Perl, PHP, Python, VBScript, and JavaScript.

SCRIPTING PROGRAMMING PARADIGM

Scripting Languages

- A scripting language, historically, was a language that allowed control of software applications.
- "Scripts" are distinct from the core code of the application, as they are usually written in a different language and are often created by the end-user.
- Scripts are most often interpreted from source code, whereas application software is typically first compiled to a native machine code or to an intermediate code.
- Early mainframe computers (in the 1950s) were non-interactive and instead used batch processing. IBM's Job Control Language (JCL) is the archetype of scripting language used to control batch processing.
- The first interactive operating systems **shells** were developed in the 1960s to enable remote operation of the first time-sharing systems, and these used **shell scripts**, which controlled running computer programs within a computer program, the shell.

Scripting Languages

- Historically, there was a clear distinction between "real" high speed programs written in **compiled** languages such as C, and simple, slow scripts written in **interpreted** languages such as Bourne Shell or Awk.

- But as technology improved, the performance differences shrank and interpreted languages like Java, Lisp, Perl and Python emerged and gained in popularity to the point where they are considered general-purpose programming languages and not just languages that "drive" an interpreter.

- The Common Gateway Interface allowed scripting languages to control web servers, and thus communicate over the web. Scripting languages that made use of CGI early in the evolution of the Web include Perl, ASP, and PHP.

- Modern web browsers typically provide a language for writing extensions to the browser itself, and several standard embedded languages for controlling the browser, including JavaScript and CSS, or ActionScript.

Scripting Languages: Types of Scripting Languages

- **Job control languages and shells**
  - A major class of scripting languages has grown out of the automation of job control, which relates to starting and controlling the behavior of system programs. (In this sense, one might think of shells as being descendants of IBM's JCL, or Job Control Language, which was used for exactly this purpose.)
  - Many of these languages' interpreters double as command-line interpreters such as the Unix shell or the MS-DOS COMMAND.
  - Others, such as AppleScript offer the use of English-like commands to build scripts. This combined with Mac OS X's Cocoa framework allows user to build entire applications using AppleScript & Cocoa objects.

Scripting Languages: Types of Scripting Languages

- **GUI scripting**
  - With the advent of graphical user interfaces a specialized kind of scripting language emerged for controlling a computer. These languages interact with the same graphic windows, menus, buttons, and so on that a system generates.
  - They do this by simulating the actions of a human user. These languages are typically used to automate user actions or configure a standard state. Such languages are also called "macros" when control is through simulated key presses or mouse clicks.
  - They can be used to automate the execution of complex tasks in GUI-controlled applications.

Scripting Languages: Types of Scripting Languages

- **Application-specific scripting languages**
  - Many large application programs include an idiomatic scripting language tailored to the needs of the application user.
  - Likewise, many computer game systems use a custom scripting language to express the game components' programmed actions.
  - Languages of this sort are designed for a single application; and, while they may superficially resemble a specific general-purpose language (e.g. QuakeC, modeled after C), they have custom features that distinguish them.
  - Emacs Lisp, a dialect of Lisp, contains many special features that make it useful for extending the editing functions of the Emacs text editor.
  - An application-specific scripting language can be viewed as a domain-specific programming language specialized to a single application.

Scripting Languages: Types of Scripting Languages

- **Web scripting languages (server-side, client-side)**
  - A host of special-purpose languages has developed to control web browsers' operation. These include JavaScript, VBScript (Microsoft - Explorer), XUL (Mozilla – Firefox), and XSLT, a presentation language that transforms XML content.
  - Client-side scripting generally refers to the class of computer programs on the web that are executed by the user's web browser, instead of server-side (on the web server). This type of computer programming is an important part of the Dynamic HTML (DHTML) concept, enabling web pages to be scripted; that is, to have different and changing content depending on user input, environmental conditions (such as the time of day), or other variables.
  - Web authors write client-side scripts in languages such as JavaScript (Client-side JavaScript) and VBScript.
  - Techniques involving the combination of XML and JavaScript scripting to improve the user's impression of responsiveness have become significant enough to acquire a name, such as AJAX.

Scripting Languages: Types of Scripting Languages

- Client-side scripts are often embedded within an HTML document (hence known as an "embedded script"), but they may also be contained in a separate file, which is referenced by the document that use it (hence known as an "external script").

- Upon request, the necessary files are sent to the user's computer by the web server on which they reside. The user's web browser executes the script using an **embedded interpreter**, then displays the document, including any visible output from the script. Client-side scripts may also contain instructions for the browser to follow in response to certain user actions, (e.g., clicking a button). Often, these instructions can be followed without further communication with the server.

- In contrast, server-side scripts, written in languages such as Perl, PHP, and server-side VBScript, are executed by the web server when the user requests a document. They produce output in a format understandable by web browsers (usually HTML), which is then sent to the user's computer. Documents produced by server-side scripts may, in turn, contain or refer to client-side scripts.

Scripting Languages: Types of Scripting Languages

- Client-side scripts have greater access to the information and functions available on the user's browser, whereas server-side scripts have greater access to the information and functions available on the server.
- Server-side scripts **require that their language's interpreter be installed on the server**, and **produce the same output regardless of the client's browser**, operating system, or other system details.
- Client-side scripts do not require additional software on the server (making them popular with authors who lack administrative access to their servers). However, they do **require that the user's web browser understands the scripting language** in which they are written. It is therefore impractical for an author to write scripts in a language that is not supported by popular web browsers.
- Unfortunately, even languages that are supported by a wide variety of browsers may not be implemented in precisely the same way across all browsers and operating systems.

ASPECT-ORIENTED PROGRAMMING PARADIGM

Aspect-Oriented Programming

- Aspect-oriented programming entails breaking down program logic into distinct parts (so-called **concerns** or cohesive areas of functionality).
- It aims to **increase modularity** by allowing the separation of **cross-cutting concerns**, forming a basis for aspect-oriented software development.
- AOP includes programming methods and tools that support the modularization of concerns at the level of the source code, while "aspect-oriented software development" refers to a whole engineering discipline.

Aspect-Oriented Programming

- All programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing **abstractions** (e.g., procedures, modules, classes, methods) that can be used for implementing, abstracting and composing these concerns.

- But some concerns defy these forms of implementation and are called **cross-cutting concerns** because they "cut across" multiple abstractions in a program.

- Logging exemplifies a crosscutting concern because a logging strategy necessarily affects every logged part of the system. Logging thereby **crosscuts** all logged subsystems and modules, and thus many of their classes and methods.

Aspect-Oriented Programming: Terminology

- Cross-cutting concerns: Even though most classes in an OO model will perform a single, specific function, they often share common, secondary requirements with other classes. For example, we may want to add logging to classes within the data-access layer and also to classes in the UI layer whenever a thread enters or exits specific methods. Even though each class has a very different primary functionality, the code needed to perform the secondary (e.g. logging) functionality is often identical.

- Advice: This is the additional code that you want to apply to your existing model. In our example, this is the logging code that we want to apply whenever the thread enters or exits a specific method.

- Pointcut: This is the term given to the point of execution in the application at which the cross-cutting concern needs to be applied. In our example, a pointcut is reached when the thread enters a specific method, and another pointcut is reached when the thread exits the method.

- Aspect: The combination of the pointcut and the advice is termed an aspect. In the example above, we add a logging aspect to our application by defining a correct advice that defines how the cross-cutting concern is to be implemented, and a pointcut that defines where in the base code the advice is to be injected.

Aspect-Oriented Programming

- To sum-up, an aspect can alter the behavior of the base code (the non-aspect part of a program) by applying **advice** (additional behavior) at various **joint points** (points in a program) specified in a quantification or query called a **pointcut** (that detects whether a given join point matches).
- An aspect can also make binary-compatible structural changes to other classes, like adding members or parents.
- The aspects can potentially be applied to different programs, provided that the pointcuts are applicable.

Aspect-Oriented Programming: Implementation

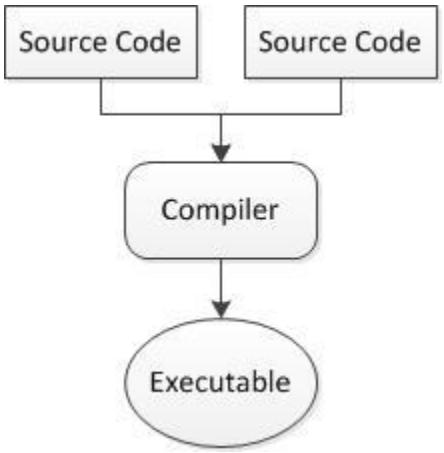
- Most implementations produce programs through a process known as **weaving** - a special case of program transformation.

- An aspect weaver reads the aspect-oriented code and generates appropriate object-oriented code with the aspects integrated.

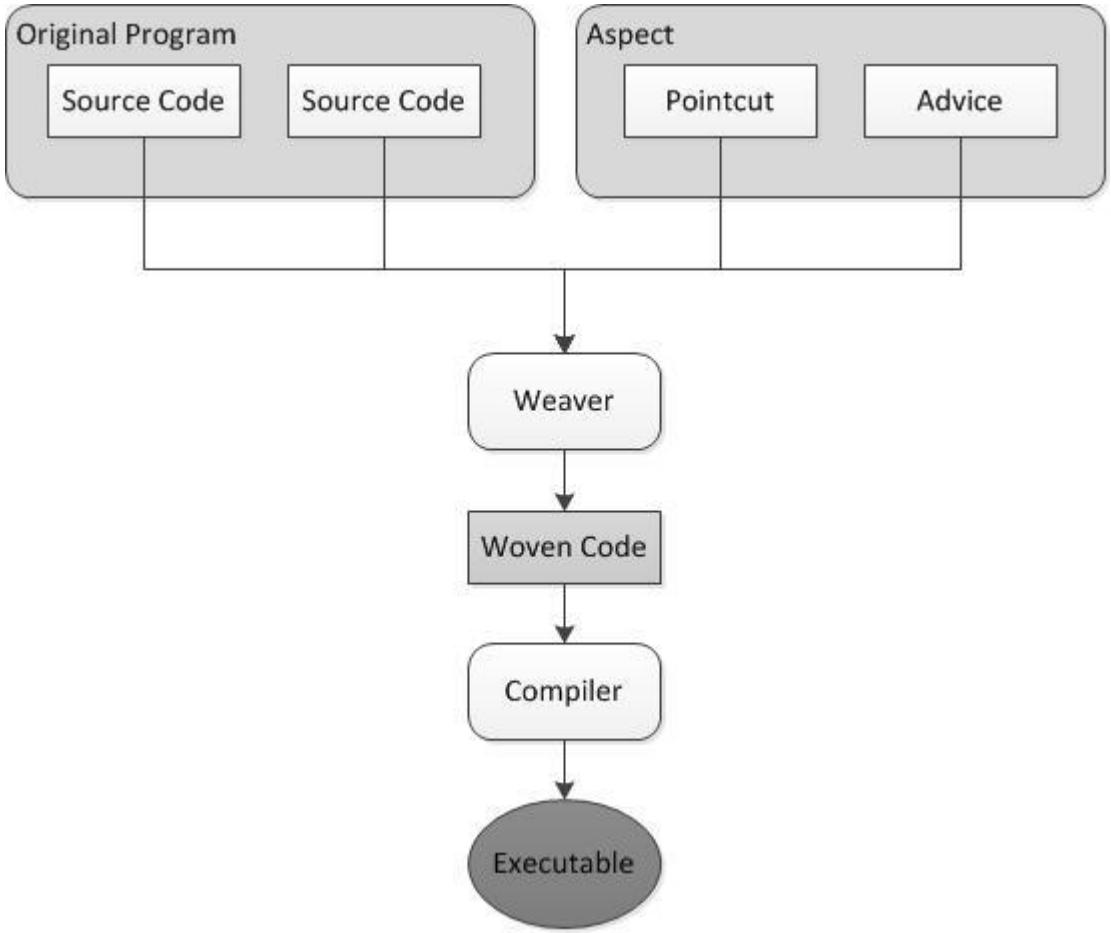
- AOP programs can affect other programs in two different ways, depending on the underlying languages and environments:

    1. a combined program is produced, valid in the original language and indistinguishable from an ordinary program to the ultimate interpreter
    2. the ultimate interpreter or environment is updated to understand and implement AOP features.

Aspect-Oriented Programming



Compilation process



Weaving process

Aspect-Oriented Programming

```
public class Foo {
   public void bar() {
      System.out.println("Executing Foo.bar()");
   }
   public void baz() {
      System.out.println("Executing Foo.baz()");
   }
}
```

base code

aspect code

```
aspect Logger {
   pointcut method() : execution(* *(..));
   before() : method() {
      System.out.println("Entering " +
         thisJoinPoint.getSignature().toString());
   }
   after() : method() {
      System.out.println("Leaving " +
         thisJoinPoint.getSignature().toString());
```

```
public class Foo {
   public void bar() {
      System.out.println("Entering Foo.bar()");
      System.out.println("Executing Foo.bar()");
      System.out.println("Leaving Foo.bar()");
   }
   public void baz() {
      System.out.println("Entering Foo.baz()");
      System.out.println("Executing Foo.baz()");
      System.out.println("Leaving Foo.baz()");
   }
}
```

woven code

Aspect-Oriented Programming: History

- AOP as such has a number of antecedents: the Visitor Design Pattern, CLOS MOP (Common Lisp Object System's MetaObject Protocol).

- **Gregor Kiczales** and colleagues at Xerox PARC developed AspectJ (perhaps the most popular general-purpose AOP package) and made it available in 2001.

Aspect-Oriented Programming: Motivation

- Typically, an aspect is scattered or tangled as code, making it harder to understand and maintain.
- It is scattered by virtue of its code (such as logging) being spread over a number of unrelated functions that might use it, possibly in entirely unrelated systems, different source languages, etc.
- That means to change logging can require modifying all affected modules. Aspects become tangled not only with the mainline function of the systems in which they are expressed but also with each other.
- That means changing one concern entails understanding all the tangled concerns or having some means by which the effect of changes can be inferred.

Aspect-Oriented Programming: Join Point Model

- The advice-related component of an aspect-oriented language defines a join point model (JPM). A JPM defines three things:
  - When the advice can run. These are called join points because they are points in a running program where additional behavior can be usefully joined. A join point needs to be addressable and understandable by an ordinary programmer to be useful. It should also be stable across inconsequential program changes in order for an aspect to be stable across such changes. Many AOP implementations support method executions and field references as join points.
  - A way to specify (or quantify) join points, called pointcuts. Pointcuts determine whether a given join point matches. Most useful pointcut languages use a syntax like the base language (for example, AspectJ uses Java signatures) and allow reuse through naming and combination.
  - A means of specifying code to run at a join point. AspectJ calls this advice, and can run it before, after, and around join points. Some implementations also support things like defining a method in an aspect on another class.
- Join-point models can be compared based on the join points exposed, how join points are specified, the operations permitted at the join points, and the structural enhancements that can be expressed.

Aspect-Oriented Programming: Implementation

- Java's well-defined binary form enables bytecode weavers to work with any Java program in .class-file form. Bytecode weavers can be deployed during the **build** process or, if the weave model is per-class, during class **loading**.

- AspectJ started with source-level weaving in 2001, delivered a per-class bytecode weaver in 2002, and offered advanced load-time support after the integration of AspectWerkz in 2005.

- Deploy-time weaving offers another approach. This basically implies post-processing, but rather than patching the generated code, this weaving approach **subclasses** existing classes so that the modifications are introduced by method-overriding. The existing classes remain untouched, even at runtime, and all existing tools (debuggers, profilers, etc.) can be used during development.

Aspect-Oriented Programming: Problems

- Programmers need to be able to read code and understand what is happening in order to prevent errors.
- Even with proper education, understanding crosscutting concerns can be difficult without proper support for visualizing both static structure and the dynamic flow of a program. Starting in 2010, IDEs such as Eclipse have begun to support the visualizing of crosscutting concerns, as well as aspect code assist and refactoring.
- Given the intrusive power of AOP weaving, if a programmer makes a logical mistake in expressing crosscutting, it can lead to widespread program failure.
- Conversely, another programmer may change the join points in a program – e.g., by renaming or moving methods – in ways that the aspect writer did not anticipate, with unintended consequences.
- One advantage of modularizing crosscutting concerns is enabling one programmer to affect the entire system easily; as a result, such problems present as a conflict over responsibility between two or more developers for a given failure.
- However, the solution for these problems can be much easier in the presence of AOP, since only the aspect need be changed, whereas the corresponding problems without AOP can be much more spread out.

Aspect-Oriented Programming: Implementations

- The following programming languages have implemented AOP, within the language, or as an external library:
  - C / C++ / C#, COBOL, Objective-C frameworks, ColdFusion, Common Lisp, Delphi, Haskell, Java, JavaScript, ML, PHP, Scheme, Perl, Prolog, Python, Ruby, Squeak Smalltalk and XML.

REFERENCES

References

1. John von Neumann. First Draft Report on the EDVAC, 1945.
2. A.M. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, 2 42: 230–65, 1937.
3. J. R Gurd, C. C Kirkham, I. Watson. The Manchester prototype dataflow computer. Communications of the ACM - Special section on computer architecture CACM Homepage archive. Volume 28 Issue 1, Jan. 1985, Pages 34-52, ACM New York, NY, USA.
4. Alan Bawden, Richard Greenblatt, Jack Holloway, Thomas Knight, David Moon, Daniel Weinreb, LISP Machine Progress Report, MIT AI Lab memos, AI-444, 1977.
5. John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. Communications of the ACM . Volume 21 Issue 8, Aug. 1978. Pages 613-641. ACM New York, NY, USA.
6. Harold Abelson, Gerald Jay Sussman. Structure and Interpretation of Computer Programs. The MIT Press. 1996.