Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Theme 1. Introduction (Part 2)

Programming Languages, Technologies and Paradigms (LTP)

DSIC, ETSInf

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

etsinf
Escuela Técnica Superior de Ingeniería Informática

Motivation
Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management
Programming Paradigms
Imperative
Declarative
OO
Concurrent
Other paradigms
Interaction-based
References

Programming Paradigms

Key factors for a successful PL

- Expressive power: write clear, compact and maintainable source code
- Easy to learn
- Portable and safe
- Multi-platform and furnished with appropriate development tools and environments
- Financial support
- The migration from applications written in other languages is not difficult (e.g., C++ $\rightarrow$ Java)
- Multiple libraries are available for a variety of applications
- Downloading open code written in the language is possible

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Programming Paradigms

Definition of programming paradigm

Basic model for designing and developing programs which provides methods and techniques for producing programs according to specific guidelines (style and approach to solve a given problem)

Main paradigms:

- Imperative
- Declarative
  - functional
  - logic
- Object-oriented
- Concurrent

There also are the so-called *emerging* paradigms

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Imperative Paradigm

A program is considered as a sequence of commands that changes the state of a machine

- It establishes how to proceed $\rightarrow$ algorithm
- The main concept is the machine state, which is given by the values of the variables stored in the memory
- Instructions are sequencially processed and the program builds the sequence of machine states leading to the solution
- This model strongly follows the usual machine architecture (*Von Neumann*'s)
- Programs are structured in blocks and modules.
- Efficient, difficult to modify and verify, with side effects

Theme 1
LTP
Motivation
Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management
Programming Paradigms
Imperative
Declarative
OO
Concurrent
Other paradigms
Interaction-based
References

Imperative Paradigm

Example: Pascal

Function length in Pascal:

```
function length (l : list): integer
var
   b : boolean;
   aux : list;
begin
   b := is_empty(l);
   case b of
     true : length := 0;
     false : begin
                 aux := tail(l);
                 length := 1+length(aux);
               end;
   end;
end
```

Motivation

Concepts
- Types and type systems
- Polymorphism
  - Overloading
  - Coercion
  - Genericity
  - Inclusion
- Reflection
- Procedures and control flow
  - Parameter passing
  - Variable scope
- Memory management

Programming Paradigms
- **Imperative**
- Declarative
- OO
- Concurrent

Other paradigms
- Interaction-based

References

Imperative Paradigm

Side effects

Two calls to the same function with the same arguments may return different results

```
program test;
var
   flag : boolean;
function f (x : integer) : integer;
   begin
     flag := not flag;
     if flag then f := x else f := x+1;
   end;
begin
   flag := false;
   write(f(1));
   write(f(1));
end
```

global variable

f changes the value of the global variable

Imperative Paradigm

Side effects

Two calls to the same function with the same arguments may return different results

```
program test;
var
   flag : boolean;
function f (x : integer) : integer;
   begin
     flag := not flag;
     if flag then f := x else f := x+1;
   end;
begin
   flag := false;
   write(f(1));
   write(f(1));
end
```

Program outcome:

```
> test
1
2
```

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Imperative Programming

Features

- The main point is how to solve a problem
- The execution order crucially depends on the sequence of program statements
- Destructive assignment (new values given to a variable destroy any previously associated value) $\rightarrow$ understanding the code is harder due to these side effects
- The programmer is responsible for all control issues
- More complex than usually admitted (as witnessed by the complex semantic definitions or the difficulty of the associated techniques, e.g., formal verification techniques)
- Parallelization is difficult
- Programmers often prefer to neglect some advanced and good features in exchange for a faster execution

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Declarative Paradigm

A program describes the properties of the desired solution. The algorithm (set of instructions) which is used to find a solution is not specified

- Kowalski's insight:

PROGRAM = LOGIC + CONTROL

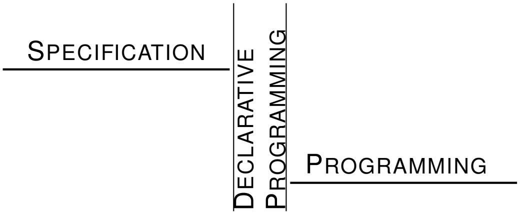
  - Logic: is about the what's
  - Control: is about the how's
- The programmer focuses on the logic aspects of the solution. Control aspects are left to the compiler/system
- Easy to verify and modify; clear and concise

Motivation
Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management
Programming Paradigms
Imperative
Declarative
OO
Concurrent
Other paradigms
Interaction-based
References

Declarative Paradigm

Declarative programs can be thought of as **executable specificacions**.

Declarative language = (executable) SPECIFICATION language
(high-level) PROGRAMMING language



Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Declarative Paradigm

Specification vs. programming

Specification: Definition of a mathematical function

```
fib(0)  = 1
fib(1)  = 1
fib(n)  = fib(n-1)+fib(n-2)
```

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Declarative Paradigm

Specification vs. programming

Specification: Definition of a mathematical function

```
fib(0)  = 1
fib(1)  = 1
fib(n)  = fib(n-1)+fib(n-2)
```

Program (two versions):

The specification!

```
fib(0)  = 1
fib(1)  = 1
fib(n)  = fib(n-1)+
          fib(n-2)
```

Optimized (with accumulator)

```
fib(0)  = 1
fib(1)  = 1
fib(n)  = fib_aux(1,1,n)
fib_aux(x,y,0)  = x
fib_aux(x,y,n)  =
        fib_aux(y,x+y,n-1)
```

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Declarative Paradigm

- **Functional Paradigm** (based on λ-calculus)
  - Data structures and functions for manipulating them are defined by means of equations (s=t)
  - polymorphism
  - higher-order
- **Logic Paradigm** (based on first-order logic)
  - relations among objects are defined by means of rules:

    If C1 and C2 and $\cdots$ and Cn, then A,
    written $A \leftarrow C1, C2, \ldots, Cn$

  - logic variables
  - indeterminism

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Declarative Paradigm

Example: Haskell and Prolog

Function `length` on lists:

Haskell

```
data list a = [] | a:list a

length [] = 0
length (x:xs) = (length xs) + 1
```

Prolog

```
length([],0).
length([X|Xs],N) :- length(Xs,M), N is M + 1.
```

Declarative Paradigm

Features

- Specification of what is a solution to a given problem
- The order of program sentences does not change the program semantics
- Expressions denote values that do not depend on the program context (referential transparency)
- High level programming:
  - simpler semantics
  - automatic control
  - amenable for parallelization
  - simpler maintenance
  - more expressive
  - smaller code
  - more productivity
- Efficiency: comparable to imperative languages like Java
- Faster acquisition of programming skills
- Some features of real systems are difficult to model in the declarative setting

Declarative *vs* imperative paradigms

Imperative Paradigm

PROGRAM	Transcript of an algorithm
INSTRUCTIONS	Control of an underlying machine
COMPUTATIONAL MODEL	State machine
VARIABLES	References to machine memory

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Declarative *vs* imperative paradigms

Declarative Paradigm

LOGIC
as a programming language

PROGRAM	Specification of a problem
INSTRUCTIONS	Logic Formulas
COMPUTATIONAL MODEL	Inference machine
VARIABLES	Logic variables

Motivation

Concepts
- Types and type systems
- Polymorphism
  - Overloading
  - Coercion
  - Genericity
  - Inclusion
- Reflection
- Procedures and control flow
  - Parameter passing
  - Variable scope
- Memory management

Programming Paradigms
- Imperative
- **Declarative**
- OO
- Concurrent

Other paradigms
- Interaction-based

References

Declarative *vs* imperative paradigms

Example

What is the purpose of this imperative program?

```
void f(int a[], int lo, hi){
  int h, l, p, t;

  if (lo<hi) {
    l = lo;
    h = hi;
    p = a[hi];
    do {
      while ((l<h)&&
             (a[l] <= p))
        l = l+1;
      while ((h>l)&&
             (a[h] >= p))
        h = h-1;
      if (l<h) {
        t = a[l];
        a[l] = a[h];
        a[h] = t;
      }
    }
    
    a[hi] = a[l];
    a[l] = p;

    f(a, lo, l-1);
    f(a, l+1, hi);
}
```

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Declarative *vs* imperative paradigms

Example

What is the purpose of this declarative program?

```
f :: Ord a => [a] -> [a]

f [] = []
f (p:xs) = (f lesser) ++ [p] ++ (f greater)
           where
             lesser = filter (< p) xs
             greater = filter (>= p) xs
```

Declarative *vs* imperative paradigms

Example

What is the purpose of this declarative program?

```
f :: Ord a => [a] -> [a]

f [] = []
f (p:xs) = (f lesser) ++ [p] ++ (f greater)
           where
             lesser = filter (< p) xs
             greater = filter (>= p) xs
```

- No variable assignment
- No indices to an array
- No memory management

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Object-Oriented Paradigm

Embed state and operations into objects

- Object: state + operations
- Important concepts: *class*, *instance*, *subclass*, inheritance
- Essential elements:
  - abstraction
  - encapsulation
  - modularity
  - hierarchy

Object-Oriented Paradigm

Example: Java

Class *Circle* is an abstraction of the notion of *circle*:

```
public class Circle {      // class name
  double radius;           // variables (state)
  String color;

  double getRadius() {...} // methods (operations)
  double getArea() {...}
}
```

Classes are used to define instances representing specific objects (circles with a particular radius and color)

```
Circle c1, c2;
c1 = new Circle(2.0, "blue");
c2 = new Circle(3.0, "red");

Circle c3 = new Circle(1.5, "red");
```

Concurrent Paradigm

- Concurrent programming languages are used to program the simultaneous execution of multiple interactive tasks
- Such tasks consist of a set of processes created by a single program:

> Concurrent access to databases, use of the resources provided by an operating system, etc.

- Concurrent programming began with the introduction of interruptions in the late fifties.
  - Interruption: a hardware mechanism to break the execution flow of a program in such a way that the CPU transfers the control to a given address, where a special routine (or handler) performs the appropriate actions that are associated to the interruption

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Concurrent Paradigm

Problems associated to concurrency

- Corruption of shared data

  For instance, if two programs concurrently write on the same printer, the output can be unreadable

- Deadlock among processes sharing resources

  Process A demands shared resources R1 and R2. In order to avoid the aforementioned problem, A tries to lock them by first requiring R1 and then R2. Simultaneously, process B tries to lock R2 and then R1. Both processes get only one of the resources and wait forever for the other

- Starvation of processes never obtaining a given resource.

  The OS enqueues the processes trying to gain access to a shared resource according to their priority. Less prioritary processes may fail to obtain resources demanded by high-priority processes.

- Indeterminism in the coordination of actions from different processes.

  Debugging concurrent programs can be very difficult due to such dependencies

Motivation
Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management
Programming Paradigms
Imperative
Declarative
OO
Concurrent
Other paradigms
Interaction-based
References

Concurrent Paradigm

Main concepts: First abstractions (1/2)

- First attempts to define concurrent languages just added OS-supported primitives to launch processes (coroutines) as part of the execution of programs written in a sequential language (Simula).
  - Problem: low level and lack of portability
- Dijkstra introduced the first abstractions (1965-71).
  - **Concurrent Program:** a set of asynchronous sequential processes making no assumption about the relative progress of other processes
  - semaphores were introduced as a synchronization mechanism

Motivation
Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management
Programming Paradigms
Imperative
Declarative
OO
Concurrent
Other paradigms
Interaction-based
References

Concurrent Paradigm

Main concepts: First abstractions (2/2)

- Hoare additionally introduced the notion of critical region to avoid deadlocks
  - managing critical region was costly and modularity was difficult to achieve
- In 1974 a new approach to encapsulate shared resources was introduced: monitors, inspired by sequential programming ADTs.
  - The first high-level concurrent language with monitors was concurrent Pascal (1975), subsequently incorporated into Modula-2.
- New architecture independent models (CSP, CCS, π-calculus, Petri nets, PVM) were introduced for the analysis of concurrent programs
  - New constructs inspired in such models were introduced in a number of languages. For instance, CSP inspired Occam's channels and Ada's remote calls.

Motivation
Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management
Programming Paradigms
Imperative
Declarative
OO
Concurrent
Other paradigms
Interaction-based
References

Concurrent Paradigm

Example: threads in Java (1/2)

Two ways to create threads in Java:

- using inheritance (`extends`)
- using interfaces (`implements`)

Inheritance

Define a subclass `MyThread` of the Java class `Thread`

```
class MyThread extends Thread {
      public void run () {
        // encode here the task to be executed
        // by the thread
      }
}
```

Motivation
Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management
Programming Paradigms
Imperative
Declarative
OO
Concurrent
Other paradigms
Interaction-based
References

Concurrent Paradigm

Example: threads in Java (2/2)

Create and use an instance of `MyThread`

```
MyThread t1 = new MyThread();
t1.setPriority(5)
t1.start();
System.out.println("Now, I can do other things");
// ...
```

- Method `start` initiates the execution of the thread (with a call to `run`)
- The priority assignment is optional
- The message is displayed disregarding the execution of the thread.

Concurrent Paradigm

About concurrency in Java

- Java class Thread provides a built-in support of concurrent programming (without additional libraries).
- Threads are similar to processes, although all resources used by threads belong to a 'root' program.
  - In contrast, processes may have their own memory addresses and execution environment.
- There are specific functions to create (and run, suspend, resume, abort, priorize, sinchronize, etc) such threads
- The JVM is able to organize them; however, avoiding undesirable behaviors (deadlock, starvation, etc.) is left to the programmer.
- The implementation of concurrent communication relies on the use of shared memory. Accordingly, some locking mechanism must be used to coordinate the threads. Actually, each object is implicitly locked if some thread is using it.

Parallel programming

Main goal:

accelerating time-consuming algorithms by splitting the execution time as much as possible, by using several processors to distribute the data and the execution workload.

- After the introduction of microprocessors in 1975, processes began to be concurrently executed in different processors. In this way, the implicit assumption (essential for monitors and semaphores) of a common memory where shared variables would be placed became unfeasible.
  - New process communication approaches were proposed. For instance, message passing approaches among processors (rendez-vous).
- Early parallel lenguajes were sequential languages (Fortran, C) extended with proprietary message passing libraries.

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Parallel *vs* Concurrent Programming

	PARALLEL	CONCURRENT
GOAL	Efficiency: workload *distribution*	Interaction: *simultaneous* processes
PROCESSORS	more than one	one or more
COMMUNICATION	message exchange	shared memory

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Interaction-based Paradigm

- The traditional paradigm follows Von Neumann's *programming as calculation* style.
  - a program describes the sequence of steps that are necessary to yield a result out from the program inputs
- This model does not fit the requirements of some areas: HCI, robotics, software agents, AI, service oriented applications, . . .

  Instead:

  computación as interaction: inputs are 'awaited' or tracked and the outputs are actions that are dynamically raised while the process is executed (there is no *final result*)

Interaction-based Paradigm

Interactive Program

A collection of entities (agents, databases, network services, etc) that interact according to some interaction rules

- The interaction rules can be constrained by interfaces, protocols and quality of service (QoS) requirements (timeouts, confidentiality, etc)
- Instances of this model of interactive programming:
  - Event-driven programming
  - Reactive systems,
  - Embedded systems
  - Client/server architectures
  - Software agents
- This model underlies distributed applications, user-interfaces design, web programming, and the incremental design of programs where parts of a program are refined during its execution.

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Interaction-based Paradigm

Event-driven programming

- the control flow is determined by the occurrence of events

> Hardware events: mouse clicks, mouse mouvements, a key pressed, external signals coming from other devices, etc... Software events: messages issued from other programs or processes, etc.

- A typical architecture for an *event-driven* (or *event-based*) application consists of a main loop having two independent sections
  1. *event-detection*
  2. *event-handling*
- As for *embedded systems*, the first section is actually part of the *hardware* and is managed by means of interruptions

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Interaction-based Paradigm

Event-driven programming

Event-driven programming is not bound to any specific programming paradigm.

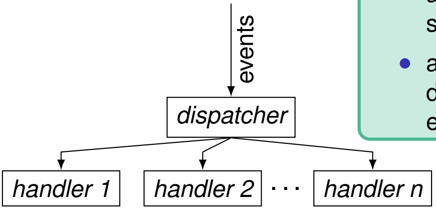
- Event-driven programs can be written in any high-level language, provided that the *event-driven* style is feasible.
- Object-orientation is not necessary
- Concurrency is not mandatory
- Requirements:
  - Catch signals, processor interruptions or, in GUI applications, mouse clicks
  - Managing an event queue to launch the appropriate event-handler

Interaction-based Paradigm

Event-driven programming

- Following the *event-handler* pattern is useful to implement this kind of applications.

The *event-handler* pattern



- a *dispatcher* manages the sequence of events
- a collection of *handlers* dealing with the different events

Interaction-based Paradigm

Event-driven programming: example of *dispatcher*

main loop

exit

handler selection

```
do forever:   // the event loop
  get an event from the input stream

  if event.type == EndOfEventStream :
    quit   // break out of event loop

  if event.type == ...:
    call the appropriate handler, passing it
    event information as an argument

  elseif event.type == ...:
    call the appropriate handler, passing it
    event information as an argument

  else:   // unrecognized event type
    ignore the event, or raise an exception
```

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Interaction-based Paradigm

Final remarks

Event-driven programming is massively used in GUI development. This is because most development environments provide assistants for implementing the event-driven pattern.

- Advantages:
  - It simplifies the programmers' burden by providing a default implementation for the main loop and management of the event queue.
- Disadvantages:
  - It promotes a too simple event-action model
  - It is difficult to extend
  - It is error prone: managing shared resources is difficult

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Emerging paradigms

- BIO-COMPUTATION: there are computational models inspired in biology
  - they use techniques that are inspired by biological systems as a basis for computation and programming
- QUANTUM COMPUTING: replace classical circuits by others that can take benefit from quantum effects (using quantum gates rather than logic gates)

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Matching PLs and Programming Paradigms

There are many multiparadigm PLs:

- **CoffeeScript** (2009): It is an OO functional and imperative language based on prototypes. CoffeeScript compiles into JavaScript.
- **Scala** (2003): Object-oriented, imperative and functional (used in Twitter together with Ruby).
- **Erlang** (1986): functional and concurrent (used by HP, Amazon, Ericsson, Facebook, . . .)
- **Python** (1989): functional (comprehension lists, lambda abstractions, fold, map) and object-oriented (multiple inheritance)

Theme 1

LTP

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

Basic references

- Cortazar, Francisco. *Lenguajes de programación y procesadores*. Editorial Cera, 2012.
- Peña, Ricardo. *De Euclides a Java: historia de algoritmos y lenguajes de programación*, Editorial Nivola, 2006.
- Pratt, T.W.; Zelkowitz, M.V. *Programming Languages: design and implementation*, Prentice-Hall, 2001
- Scott, M.L. *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2008 (revised version).
- Schildt, Herbert. *Java. The Complete Reference*. Eight Edition. The McGraw-Hill eds. 2011

Motivation

Concepts
Types and type systems
Polymorphism
Overloading
Coercion
Genericity
Inclusion
Reflection
Procedures and control flow
Parameter passing
Variable scope
Memory management

Programming Paradigms
Imperative
Declarative
OO
Concurrent

Other paradigms
Interaction-based

References

References

Implementation aspects

- “Programming Language Pragmatics”, M.L. Scott. (chapter 3)
- “Lenguajes de programación y procesadores”, Francisco Cortazar (chapter 1)
- “Programming Languages: design and implementation”, Pratt, T.W.; Zelkowitz, M.V. (chapters 9 and 10)