Fundamental Concepts of Programming Languages

PL families

Lecture 02

conf. dr. ing. Ciprian-Bogdan Chirila

University Politehnica Timisoara
Department of Computing and Information Technology

October 4, 2022

Lecture outline

- Imperative PLs
- Functional PLs
- Declarative PLs
- Sequential programming vs. concurrent programming
- Parallel processes vs. concurent processes
- Concurent programming languages
- Distributed systems programming
- Short history of PLs development

The three PL paradigms

- There are several criteria of PL classification...
- Imperative
- Functional
- Declarative
- Inside each family there is a diversity of languages
- They have the same basic principles

Imperative PLs

- Imperative means that are based on instructions
- Most widespread
    - Fortan, Cobol, Snobol4, Basic, Pascal, Ada, Modula-2, C, C++, C#, Java
- Their conception is based on the traditional von Neumann architecture
- The computer is made out of
    - Memory (holding data and instructions)
    - Command unit
    - Execution unit

Imperative PLs

- Are based on 2 concepts:
  - Sequential (step by step) execution of instructions
  - Keeping a modifiable set of values during program execution
    - Those values define the state of the system

Imperative PLs

- The 3 essential components:
  - Variables
    - Major component in imperative PLs
    - Memory cells with names assigned and values stored
  - Assignment instruction
    - Memorizing the computed value
  - Iteration
    - Typical way to do complex computation
    - To execute repeatedly a set of instruction

Example of a C imperative language
Prime number testing

```
#include <stdio.h>
#include <math.h>
int prime(unsigned long n)
{
 unsigned long i;
 if(n<=1) return 0;
 for(i=2;i<sqrt(n);i++)
  if(n%i==0) return 0;
 return 1;
}
int main()
{
 unsigned long n;
 printf("N=");
 scanf("%ld", &n);
 if(prime(n)) printf("The number %ld is prime!", n);
 else printf("The number %ld is not prime!",n);
}
```

Functional PLs

- Are based on mathematical concepts of
  - function
  - function application
- applicative languages
- are free from the von Neumann concept
- LISP, SML, Miranda, F#

Functional PLs 4 essential components

- The set of predefined primitive functions
- The set of functional forms
  - Mechanisms that allow combining functions in order to create new ones
- The apply operation
  - Allows applying a function on arguments and producing as a result new values
- The data set (objects)
  - The set of arguments and function values

Example of Lisp functional language
List atom counting

```
(defun count(x)
 (COND ((NULL x) 0)
  ((ATOM x) 1)
   (T (+ count (CAR x))
      (count (CDR x))))))
```

Example of F# functional language
Fibonacci Number formula

```
let fib n =
    let rec g n f0 f1 =
        match n with
        | 0 -> f0
        | 1 -> f1
        | _ -> g (n - 1) f1 (f0 + f1)
    g n 0 1
```

Declarative PLs

- in the development process of a software system
  - in the requests and specifications phase
    - we say WHAT must the system do
  - in the design and implementation phase
    - we implement HOW the system works

What's new in declarative PLs?

- we stop at the specification phase
- we describe what we expect from a system
- we do not to define the implementation of the system
- we specify only
  - problem properties
  - problem conditions
- the system will automatically find the answers

Declarative PLs

- To focus the effort and creativity in the request definition phase
- Very high level languages
- SQL
  - Structured Query Language
  - is a domain specific language
  - is used for database interrogation
- LINQ
  - Language Integrated Query
  - Microsoft .NET Framework component
- Prolog
  - both declarative and logic
  - problem conditions are expressed through predicate calculus

Example of declarative program in LINQ

```
var results =
    from c in SomeCollection
    where c.SomeProperty < 10
    select new {c.SomeProperty, c.OtherProperty};

foreach (var result in results)
{
    Console.WriteLine(result);
}
---
var results =
    SomeCollection
        .Where(c => c.SomeProperty < 10)
        .Select(c => new {c.SomeProperty, c.OtherProperty});

results.ForEach(x => {Console.WriteLine(x.ToString());})
```

Example of declarative program in Prolog

```
parent(helen,ralph).
parent(peter,ralph).
parent(peter,marry).
parent(ralph,anna).
parent(ralph,dan).
```

```
? - parent(peter,mary).
yes

? - parent(peter,x).
x=ralph

? - parent(peter,x).
x=ralph;
x=mary;
no

? - parent(y,anna), parrent(x,y).
x=helen;
x= peter;
y=ralph;
no
```

Generally, PLs

- are not pure
  - imperative or functional or declarative
- ML
  - functional with imperative facilities
- C
  - programs defining and using functions intensively
- F#
  - functional with imperative facilities

PLs and machines

- Imperative languages
  - work optimal on actual computers
- Functional and declarative languages have
  - solid theoretical foundations
  - automatic checking
  - high level programming

Functional and declarative PL domains

- Artificial intelligence
- List processing
- Databases
- Symbolic calculus
- Natural language processing
- Knowledge bases
- Program checking
- Theorem provers

Sequential programming vs. concurrent programming

- Imperative program
  - actions
  - data
- If the next action is initiated when the current action has finished
- then he program becomes a process
- The programming activity is named as sequential programming

Parallel vs. concurrent processes

- usually a process uses computer resources
  - one at a time
- if only one process in a system is using all the resources
  - we got a weak usage performance
  - multiple processors are useless
- multiple processes in memory using multiple CPUs/cores
  - when each process uses one CPU/core
  - creates a physical parallelism
- multiple processes in memory which use one CPU/cores in time division
  - is useful
  - creates logic parallelism
  - virtually the processes are executed in parallel

Multiprogramming operating systems

- are derived from the previous ideas
- multiple programs are present in the memory
- they are executed in parallel
- their physical parallelism depends on
    - The number of CPUs/cores/virtual cores
    - The type of CPUs/cores/virtual cores (hyperthreading)
- multiprogramming is present in modern OSs like: Windows, Unix, MacOS, OS/2
    - they allow multi-programmed process management
    - they share the system's resources
    - they create a great improvement to the resource usage rate

Programs on multiprogramming OSs

- form parallel processes
- are executed independently
    - as if they would ran alone on a mono programmed system
- resource conflicts are
    - handled by the OS
    - opaque to the application programmer

Processes with communication

- Isolated processes are not always a solution
- The solution may be multiple processes
    - asynchronous
    - with message exchange
    - with data transfer
    - sharing in common the system resources
- Concurrent processes
- Sometimes they need synchronization

Synchronization cases

- Mutual exclusion
- Cooperation

Mutual exclusion

- Multiple processes use the same resource
- The access is permitted to one process at a time
- The access requests must be sequenced
  - the synchronization can be based on a condition
  - a process can be delayed until a condition becomes true
- Critical resource
  - is a resource that may be used in a single process at one time
- Critical section
  - is the code section manipulating the critical resource

Mutual exclusion

Definition:

- Mutual exclusion is a synchronization form for concurrent processes allowing that only one process to be in the critical section at one time
- A language construction to solve this issue is the critical region
  - Added by CAR Hoare and P. Brinch Hansen in 1972 and involves:
  - to emphasis program text and variables which denote the critical resource
  - to add new keywords like region, when for the access of such resources
  - to add a synchronization condition to obtain a conditional critical region

Cooperation

- messages or data are exchanged between processes
- they keep a producer/consumer relationship
- the information produced by a process is used/consumed by the other
- describing concurrent processes and their relationship leads to concurrent programming
- resources are
  - shared between authorized processes
  - protected from unauthorized processes
- when the time factor is involved we get real-time processes
- concurrent processes programming languages

Distributed systems

- Concurrent systems
- The most widespread because of the Internet and networking
- Communication based on message transmission

Concurrent programming languages

- Are developed in the last 30 years
- Have special facilities to describe
    - parallel and concurrent processes
    - synchronization and communication
- Edison defined by P. Brinch Hansen 1980
- To describe concurrent programs of small and medium sizes for micro and mini computing systems

The "when" instruction

- The processes
  - communicate through common variables
  - synchronize through conditional critical regions

```
when b_1 do instr_list_1
else b_2 do instr_list_2
...
else b_n do instr_list_n
```

The "when" instruction

- the common variable for the critical region is not specified
- Edison solution
  - Mutual exclusion of all critical regions
  - Only one critical sequence is executed at one time
- thus, it results
  - Simplified language implementation
  - Complex restrictions regarding process concurrency

The "when" instruction

- Is executed in two phases
  - Synchronization phase
    - The process is delayed until no other process executes the critical phase of a when instruction
  - Critical phase
    - Logical expressions are evaluated b_1, b_2,...,b_n
    - If one of them is true the corresponding instruction list is executed
    - If all are false the synchronization phase is repeated

The "cobegin" instruction

- describes the concurrent activities

```
cobegin
    const_1 do instr_list_1 also
    const_2 do instr_list_2 also
...
    const_n do instr_list_n
end
```

- the instruction list represents processes to be executed in parallel
- processes start at cobegin
- cobegin ends when all processes end
- each process has a constant attached
- the constant semantic in PL implementation dependent
  - necessary memory space
  - the processor number
  - the priority etc.

The Edison program

- Has the form of a procedure
- Is launched by activating the procedure instructions
- Is formed out of several modules
- The exported identifiers are preceded by the star * symbol

The Philosophers problem

- 5 philosophers spend their life eating and meditating
- When a philosopher is hungry goes to the dining room, sits at the table and eats
- To eat from the spaghetti dish he needs 2 forks
- On the table there are only 5 forks
- There is only one fork between two places
- Each philosopher can access the forks at his right and left hand-side
- After eating (a finite amount of time) the Philosopher puts back the forks and leaves the room

The solution program

- the philosophers behavior is modeled by concurrent processes
- the forks are modeled by the shared resources
- philosophers wait until both forks are free
- the "forks" table stores the number of forks available to a philosopher
- it can occur the starvation situation when the neighbors are eating alternatively
- the 5 philosophers represent the 5 activations of the "philosopherlife" procedure in each cobegin branch
- each branch launches one parallel process

The Philosophers program

```
proc philosophers
module
  array tforks[0...4] (int)
  var forks:tforks; i:int;

  proc philoright(i:int):int
  begin
    val philoright:=(i+1) mod 5
  end

  proc philoleft(i:intr):int
  begin
    if i=0 do val philoleft:=4
    else true val philoleft:=i-1
  end
```

The Philosophers program

```
*proc get(philo:int)
begin
  when forks[philo] = 2 do
    forks[philoright(philo)]:=forks[philoright(philo)]-1;
    forks[philoleft(philo)]:=forks[philoleft(philo)]-1;
  end
end
*proc put(philo:int)
begin
  when true do
    forks[philoright(philo)]:=forks[philoright(philo)]+1;
    forks[philoleft(philo)]:=forks[philoleftt(philo)]+1;
  end
end
```

The Philosophers program

```
begin
  i:=0
  while i<5
    forks[i]:=2
    i:=i+1;
  end
end
```

The Philosophers program

```
proc philosoperlife(i:int)
begin
  while true do
    -think-
    get(i);
    -eat-
    put(i);
  end
end
```

The Philosophers program

```
begin
  cobegin
    1 do philosopherlife(0) also
    2 do philosopherlife(1) also
    3 do philosopherlife(2) also
    4 do philosopherlife(3) also
    5 do philosopherlife(4) also
  end
end
```

Distributed systems programming

- Distributed system
  - a set of computers capable of information exchange
  - computers are called nodes
  - can be programmed to solve problems involving
    - concurrency
    - parallelism

Typical algorithmic problems

- Synchronization on condition
- Message broadcasting to all nodes
- Process selection for fulfilling special actions
- Termination detection
    - A node performing an action must be capable of detecting its ending moment
- Mutual exclusion
    - Using resources by mutual exclusion
    - Files, printers, etc
- Deadlock detection and prevention
- Distributed file system management
- a PL for distributed systems must have all facilities: Java
- example: a chat system

The client/server model

- Server processes
  - managing resources
- Client processes
  - accessing resources managed by servers
- The message is limited to only one text line
- The server
  - must be started first
  - developed in the compilation unit Server.java
- The client
  - sens a message
  - waits for an answer
  - send the STOP command
  - developed in the compilation unit Client.java

Client.java

```
import java.net.*; import java.io.*;
class Client
{
  public static void main(String[] args) throws IOException
  {
    Socket cs=null;
    BufferedReader is=null; DataOutputStream os=null;
    try
    {
      cs=new Socket("localhost",5678);
      is=new BufferedReader(new InputStreamReader(cs.getInputStream()));
      os=new DataOutputStream(cs.getOutputStream());
    }
    catch(UnknownHostException e)
    {
      System.out.println("No such host");
    }
```

Client.java

```
    BufferedReader stdin=
      new BufferedReader(new InputStreamReader(System.in));
    String line;
    for(;;)
    {
      line=stdin.readLine()+"\n";
      os.writeBytes(line);
      System.out.println("Transmission:\t"+line);
      if(line.equals("STOP\n")) break;
      line=is.readLine();
      System.out.println("Receiving:\t"+line);
    }
    System.out.println("READY");
    cs.close(); is.close(); os.close();
  }
}
```

Server.java

```
import java.net.*;
import java.io.*;
class Server
{
  public static void main(String[] args) throws IOException
  {
    ServerSocket ss=null; Socket cs=null;
    BufferedReader is=null; DataOutputStream os=null;
    try
    {
      ss=new ServerSocket(5678);
      System.out.println("The server is running!");
      cs=ss.accept();
      is=new BufferedReader(new InputStreamReader(cs.getInputStream()));
      os=new DataOutputStream(cs.getOutputStream());
```

Server.java

```
      BufferedReader stdin=
        new BufferedReader(new InputStreamReader(System.in));
      String line;
      for(;;)
      {
        line=is.readLine();
        System.out.println("Receiving:\t"+line);
        if(line.equals("STOP")) break;
        line=stdin.readLine()+"\n";
        os.writeBytes(line);
      }
    }
    finally
    {
      cs.close(); ss.close();
      is.close(); os.close();
    }
  }
}
```

The socket

- An IP address identifies a computer in Internet
- A port number identifies a program running on a computer
- A combination between an IP address and a port is a final point for a communication path
- Two communicating applications must find themselves in the Internet
- Typically the client must find the server

The socket

- The client connects to the server by initiating a socket connection
- The first client message to the server contains the client socket
- The server transmits its socket address to the client in the first reply message
- Data transmission is done through socket input/output streams
- The streams can be accessed through getInputStream() and getOutputStream() from class Socket

Short history of PL development

- First high level PL were created in 1950
- In this period the efficiency was the main goal
- Fortran
  - Designed by a group from IBM lead by John Bachus 1954
- Algol 60
  - 1958-1960
  - Block structures
  - Recursive procedures

Short history of PL development

- Cobol
  - Financed by Department of Defense in 1959
  - Economical applications
  - Files
  - Data description facilities
    - record
    - struct
  - Used in current days in an evolved version

Short history of PL development
Late 50s and early 60s

- Functional PLs
  - Lisp
    - John McCarthy MIT 1955
    - The main PL in artificial intelligence
  - APL
    - Iverson IBM 1962
- Imperative PL
  - Snobol
    - Bell Laboratories 1964

Short history of PL development
Mid 60s

- there was a large diversity of programming languages
- the IBM project intended:
  - to gather all concepts in a single PL
  - to replace all other PLs
  - in 1964 it resulted the PL/I language
    - limited success
    - complex
    - heavy

Short history of PL development
In the 60s

- Algol68, 1968
  - perfect orthogonality
  - defined using formal methods
- Simula67, 1967
  - has simulation facilities
  - uses the class concept for
    - modularization
    - abstract data description

Short history of PL development

- Pascal 1971
  - N. Wirth
  - Expressivity
  - Simplicity
- ML 1973
  - University of Edinburgh
  - Functional PL
  - Strongly typed

Short history of PL development

- C 1974
  - one of the most widespread PL
  - invented by Dennis Ritchie at Bell Labs in 1974
  - portable implementation for the Unix operating system
  - programs have good portability

Short history of PL development
In the 70s

- Abstract data types
- Program checking
- Exception handling
- Concurrent programming

Short history of PL development

- Mesa (Terax, 1974)
- Concurrent Pascal (Hansen, 1975)
- CLU (Liskov, MIT 1974)
- Modula2 (Wirth, 1977)
- Ada (DoD, 1979)
- Prolog (Colmeraurer, 1972)
  - Logic programming
  - Artificial intelligence

Short history of PL development
In the 80s

- Common Lisp 1984
  - Was used and consolidated
- Standard ML
  - SML, Milner, Edinburgh, 1984
- Miranda
  - Turner, Kent, 1985
- Haskell
  - Hudak, 1988

Short history of PL development

- Object-oriented programming languages
- SmallTalk
  - PL and IDE altogether
  - created by Xerox in the late 70s
- C++
  - created by Bjarne Stroustrup at Bell Labs in 1988
  - C retrofitted with object-oriented concepts
  - Widely used in present

Bjarne Stroustrup seminar at INRIA, Sophia Antipolis, France, July, 2003

Short history of PL development

- Python
  - developed in late 80's by Guido van Rossum in Netherlands
  - successor of ABC programming language
  - first released in 1991
  - Python 2.0 released in 2000, Python 3.0 released in 2008
  - multiparadigm: object-oriented, structured, support for functional programming, aspect-oriented programming (meta-programming, meta-objects)
- Object Oberon
  - Zurich, 1989
- Eiffel
  - developed by Bertrand Meyer in 1988 and also today
- Java
  - developed by Sun Microsystems Inc. in 1995, now by Oracle
  - object-oriented programming supporting functional programming

Short history of PL development: Java

- industrial strength technology
  - micro edition - cards, devices
  - standard edition - desktop applications, interactivity, animation
  - enterprise edition - distributed applications over the Internet,
  - mobile edition - mobile apps
- in 2022, Java version 19
- has anti C++ philosophy
  - no pointer arithmetic
  - no manually releasing memory
  - no multiple inheritance between classes
- Other object-oriented PLs
  - Object Pascal (Delphi, Borland 1995-2000)
  - CLOS (Common Lisp Object System)
  - OCAML (object-oriented ML)

Short history of PL development: C#

- general purpose, object-oriented, internationalization
- string type checking, array bounds checking
- Alpha release in 2000
- Microsoft team lead by Anders Hejlsberg
- Derived from C, C++ and Java
- Portability taken from Java
- Can be mixed with other PL: F#, C++, LINQ
- designed to build software components deployable in distibuted environments
- Full integration with MS Windows OS

Bibliography

1. Brian Kernighan, Dennis Ritchie, C Programming Language, second edition, Prentice Hall, 1978.
2. Carlo Ghezzi, Mehdi Jarayeri – Programming Languages, John Wiley, 1987.
3. Horia Ciocarlie – Universul limbajelor de programare, editia 2-a, editura Orizonturi Universitare, Timisoara, 2013.