Introduction
Course objectives

Become familiar with the basic notions and principles of functional programming (FP), and use them to solve concrete problems.

- Description of the paradigm of functional programming.
- Acquire functional programming skills using the RACKET programming language

Expected outcome

- Ability to discuss issues of functional programming (measured by course tests)
- Ability to design and implement programs in RACKET that demonstrate the concepts covered in the course (measured by problem sets).
- Design, implement and test short RACKET programs (measured by lab tests and/or mini-projects)
Organizational items

- 13 lectures, 7 labs (weeks 1, 3, 5, ...)
- Lecturer and TA: Mircea Marin

Grading:
- Mini-projects & labwork assignments: 50%
- Colloquium = written exam (week 14): 50%
- Lab attendance: at least 4 labs

Sources of information (lectures, labworks,)
- website: web.info.uvt.ro/~mmarin/lectures/FP/
- Recommended bibliography:
In **Software Engineering**, there are four main **programming paradigms**:

1. Imperative (or Procedural) Programming
2. Object-oriented Programming (OOP)
3. Functional Programming
4. Logic Programming

Every programming paradigm has its own **concepts and notions** which are used to guide our way of thinking and to describe our solutions to problems.

- collection of **languages** with programming constructs for the main concepts and notions
- model of **computation**
1. Procedural (or imperative) programming

- Oldest paradigm: **First do this and next do that.**

- Main notions: command, procedure, and program state.
  - **program state**: a collection of variables with values.
  - **command** = execution of an **instruction**; it **changes** the **program state** as a function of time.
  - **procedure** = name given to a collection of instructions which can be called as a single command.
  - **program** = collection of procedure definitions and instructions to be executed; similar to descriptions of everyday routines, such as food recipes and car repair.
  - **computation** = stepwise execution of **instructions** in an order governed by **control structures** (blocks, conditionals, loops, . . .)

Representative languages: Fortran, Algol, Pascal, Basic, C
1. Imperative programming
Example of an imperative program (pseudocode)

Compute the value of the smallest element from a list \( L \) with \( n \) elements:

\[
\begin{align*}
\text{minList}(L, n) & \quad \text{min}(a, b) \\
& \quad \quad \text{if } a < b \text{ then} \\
& \quad \quad \quad \quad m \leftarrow a \\
& \quad \quad \text{else} \\
& \quad \quad \quad \quad m \leftarrow b \\
& \quad \quad \text{endif} \\
& \quad i \leftarrow i + 1 \\
& \quad \text{return } m
\end{align*}
\]

- program state: variables \( L, r, i, n \)
- basic instruction: assignment (e.g., \( m \leftarrow a \))
  - assignments change the program state
- control structures: while, if, ...
- procedures: minList, min
Programming paradigms
2. Object-oriented programming (OOP)

- world = collection of objects that communicate by sending messages to each other.
- object: description of a concept or notion with
  - data fields = the state of the object
  - methods = operations that can act on the fields of the object

Thus, OOP = generalisation of imperative programming, where the program state is distributed over many objects.

- message passing = applying a method on an object ⇒ the object state may change.

Usually, the structure of an object (=data fields + methods) is described by a class ⇒ object is an instance of a class.

⇒ program = collection of class definitions.

- classes are organised in inheritance hierarchies

Representative languages: Simula, Java, C++, SmallTalk, JavaScript
Answer a question by searching the answer according to a fixed, predictable strategy.

Main notions: fact, rule, query.

Way of thinking which is useful for solving problems related to the extraction of knowledge from basic facts and relations.

- The programmer must describe what he knows as facts and rules collected in a program.
- The compiler (or interpreter) of the programming language finds the answers to all questions we may ask afterwards, using a built-in search strategy.

Representative language: Prolog

- search strategy of Prolog: SLDNF-resolution
3. Logic programming (LP)
Typical example: find the minimum element of a list

- **Fact:** The minimum element of a singleton list made of number \( m \) is \( m \).

- **Rules of inference:**
  1. The minimum of \( x \) and \( y \) is \( x \) if \( x \leq y \).
  2. The minimum of \( x \) and \( y \) is \( y \) if \( y < x \).
  3. The minimum element of a list starting with \( x, y \) followed by sublist \( t \) is \( m \) if \( m \) is the minimum of \( x \) and \( n \), where \( n \) is the minimum element of the list with first element \( y \) followed by sublist \( t \).

**Corresponding Prolog program**

\[
\begin{align*}
\text{min}(X,Y,X) & :\:- X =< Y. \\
\text{min}(X,Y,Y) & :\:- Y < X. \\
\text{minList}([M],M). \\
\text{minList}([X,Y|T],M) & :\:- \text{minList}([Y|T],N), \text{min}(X,N,M).
\end{align*}
\]
Describe every computation as a request to **evaluate an expression**, and use the resulting value for something.

Main notions: function, value, expression.

- **program**: collection of function definitions in the *mathematical* sense
  - The value of a function call depends only on the values of the function arguments.
- **expression** = (typically) a combination of nested function calls.
- **computation** = evaluation of an expression → a **value**.
- **value** = element of a datatype (string, integer, list, etc.) that can be
  - named
  - stored in a composite data (e.g., element of a list or vector)
  - passed as argument to a function call
  - returned as result of a function call
4. Functional Programming

Characteristic features

- **Variables are just names given to values**
- **There is no assignment** ⇒ we can not change the value of a variable
  - ⇒ we can not define repetitive computations by iteration.
  - ⇒ we define repetitive computations by recursion
- **Functions are values** ⇒ we can have
  - functions that take function arguments
  - functions that return functions as results
  - lists of functions, etc.
Example 1: Computation of $n!$ by recursion

We can not define $\text{fact}$ by iteration

```c
fact (int n)
    i=1; fact:=1;
    if(i<n)
        i:=i+1;
        fact:=fact*i;
    endif
    return fact
```

because there is no assignment in functional programming. But we can define $\text{fact}$ by recursion:

```c
fact (int n)
    if (n==1)
        1
    else
        n*fact (n-1)
```
Example 2: A function with a function argument

\[
\text{map}(f, L) \text{ takes as inputs a function } f : A \rightarrow B \text{ and a list } \\
L = (a_1, a_2, \ldots, a_n) \text{ of elements from } A \\
\text{and returns the list } \\
(f(a_1), f(a_2), \ldots, f(a_n)) \text{ of elements from } B
\]

Assume that

- \text{empty}(L) \text{ recognises if } L \text{ is empty list}
- \text{first}(L) \text{ returns the first element of a list}
- \text{rest}(L) \text{ returns the list produced by removing the first element of } L
- \text{prepend}(e, L) \text{ adds element } e \text{ in front of list } L

\[
\text{map}(f, L) \\
\text{ if (empty(L)) } L \\
\text{ else prepend(f(first(L)), map(f, rest(L)))}
\]
Why learn functional programming?

- It has a very simple model of computation:
  - Programs consist (mainly) of function definitions
  - Computation = evaluation of (nested) function calls
- We can define higher-order functions (functions that take functions as arguments and/or return functions as results)
  \[ \Rightarrow \text{we can write highly modular and reusable code} \]
  [Hughes:1989]
- According to [Thompson:1999]:
  \[ \text{“Functional languages provide a framework in which the crucial ideas of modern programming are presented in the clearest possible way. This accounts for their widespread use in teaching computing science and also for their influence on the design of other languages.”} \]
What will you learn?

How to use DrRacket to write and run functional programs

- DrRacket is an integrated development environment (IDE) for Racket, the current dialect of Scheme
- DrRacket is freely available to be installed on Windows, Linux, MacOS, etc.
  
  https://racket-lang.org

When started, DrRacket shows a window with two panels:

1. the definitions area, where you can start typing your own programs, save them for later use and run them.
2. the interactions area, where you can interact directly with the interpreter of RACKET.
The first high-level programming language was Fortran (1957). Fortran is an imperative programming language.

The second high-level programming language was Lisp (1958).

- Designed by people interested in AI ("the science and engineering of making intelligent machines"); Lisp became the favoured language for AI research
- Lisp is acronym for "List Processing": Lists are used to represent both source code and composite data.

Initial Lisp had no standard specification ⇒ many dialects of Lisp appeared ⇒ people were confused, and wanted a standardised and reliable version of Lisp

⇒ **Common Lisp** (ANSI 1994 standard): extensive libraries, ideal for writing commercial software
⇒ **Scheme** (IEEE 1990 standard): wonderful for educational purposes.

**The most recent dialect of Scheme is Racket.**
They are strict programming languages

A language is strict if the evaluation of function calls proceeds as follows: First, we compute the values of the arguments, and next we call the function with the computed values.

Example

\[4 + ((2 - 2) \times (4 - 3))\] is the infix notation for the nested function call \[+(4, \times (0, -(4, 3)))\]. The strict evaluation of this expression is:

\[+(4, \times (-(2, 2), -(4, 3)) \rightarrow +(4, \times (0, -(4, 3))) \rightarrow +(4, \times (0, 1)) \rightarrow +(4, 0) \rightarrow 4\]

All expressions are written in a peculiar syntax, called the parenthesised prefix notation (see next slide)
The parenthesised prefix notation

- Every function call $f(e_1, e_2, \ldots, e_n)$ is written as
  
  \[(f \; pe_1 \; pe_2 \; \ldots \; pe_n)\]
  
  where $pe_1, pe_2, \ldots, pe_n$ are the parenthesised prefix notations of $e_1, e_2, \ldots, e_n$

- Every other composite programming construct is of the form
  
  \[(form-id \; \ldots)\]
  
  where $form-id$ is the identifier of the programming construct.

Characteristics of the parenthesised prefix notation

- Every open parenthesis has a corresponding close parenthesis
- Instead of comma, we type whitespace (one or more blanks, newlines, tabs, etc.)
The parenthesised prefix notation

Examples

- \((2+7)/3-1\) \(\times\) \((7-4)\) is written as
  \((\ast (- (/ (+ 2 7) 3) 1) (- 7 4))\)

- The parenthesised prefix notation of
  \(\text{if } (n=1) 1 \text{ else } n\ast\text{fact}(n-1)\)
  is
  \((\text{if } (= n 1) 1 (* n (\text{fact } (- n 1))))\)

Remark

The parenthesised prefix notation of

\(\text{if } \text{cond} \ \text{expr}_1 \ \text{else} \ \text{expr}_2\)

is \((\text{if } \text{cond-pe} \ \text{expr}_1\text{-pe} \ \text{expr}_2\text{-pe})\)

where \(\text{cond-pe} \ \text{expr}_1\text{-pe} \ \text{expr}_2\text{-pe}\) are the parenthesised prefix notations of \(\text{cond}, \ \text{expr}_1, \ \text{expr}_2\).
Values are the simplest expressions: they evaluate to themselves.

A value is either atomic or composite.

Every value belongs to a datatype.

Datatypes with atomic values:

- integer: 1 -703 12345678999999
- floating-point: 1.23 3.14e+87
- string: "abc"
- symbol: ’abc
  (symbol values are preceded by the quote character’)
- boolean: #t (for truth)  #f (for falsehood)
  ...

Some datatypes have composite values: pairs, lists, vectors, hash tables, etc.

- A composite value is a collection of other values.
- Composite values and datatypes will be described in Lecture 2.
In the interactions area, at the input prompt >

- Type in an expression $e$ in parenthesised prefix notation, and press Enter

- $e$ will be read, evaluated, and the resulted value will be printed on the next line in the interactions area.
the nesting of parentheses clarifies the order in which the function applications should be performed

the semicolon ; starts a comment (highlighted with brown) that extends to the end of the line

comments are ignored by the interpreter
Expressions and definitions

The interpreter of RACKET recognises two kinds of input forms:

**Expressions:** An expression \( expr \) is read, evaluated, and its value is printed in the interactions area.

**Definitions:** A definition is of the form

\[
\text{(define } \text{var } \text{expr})
\]

Definitions are interpreted as follows:

- \( expr \) is evaluated, and its value is assigned to variable \( \text{var} \).
- Afterwards, \( \text{var} \) can be used to refer to the value of \( expr \).

Example (Compute \( \sqrt{x^2 + y^2} \) for \( x = 2 \) and \( y = 3 \))

\[
> \text{(define x 2)}
\]

\[
> \text{(define y 3)}
\]

\[
> \text{(sqrt (+ (* x x) (* y y)))}
\]

3.60551275463989
Function definitions

The meaning of an expression

\[
\text{(lambda (x}_1 \ldots x_n) \text{body})
\]

is “the function which, for the values of arguments \(x_1, \ldots, x_n\), computes the value of \(\text{body}\).”

- The evaluation of this expression creates a function, which is a value that can be assigned to a variable.

Example (the factorial function)

```
> (define fact
   (lambda (n)
      (if (or (= n 0) (= n 1))
         1
         (* n (fact (- n 1))))))
> (fact 5) ; compute 5!
120
```
At the end of this lecture+related lab, you should be know:

- What are the main programming paradigms of software engineering
- What are the main features of each programming paradigm
- The parenthesised prefix notation
  - write simple arithmetic and logic expressions in the parenthesised prefix notation
  - write simple functions, using the lambda-construct
  - Use DrRacket to compute the values of arithmetic expressions, define functions, and call them.