Logic and Functional Programming

Lecture 1: Introduction

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

Content of Lecture 1

- Organizatorial items
- Programming styles in software engineering
  - Main features
  - Comparison via an illustated example
- Declarative versus imperative programming styles
  - Characteristics of declarative programming
  - Characteristics of functional programming
- Functional programming languages
  - Short history
  - Racket and Haskell

Organization

Weekly lecture.

Topics:

- Introduction to Functional Programming (7 weeks):
  - theoretical aspects: lambda calculus, etc.,
  - practical programming in Racket and Haskell
- Introduction to logic programming (7 weeks):
  - logical foundations, computational model
  - practical programming in Prolog: programming techniques; selected examples; efficiency issues.

Grading:

- in-class quizzes, individual assignments: 20%
- two partial exams: 25% FP; 25% LP
- Written exam: 30%

For Functional Programming:

- H. Abelson, G. J. Sussman, J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press Ltd. 1996.
- M. Marin, V. Negru, I. Drămnesc. *Principles and Practice of Functional Programming*. Editura UVT. 2016.
- S. Thompson. *Haskell: The Craft of Functional Programming*. Second Edition. Pearson Education Ltd. 1999.
- Haskell tutorials

For Logic Programming:

- W.F. Clocksin and C.S. Mellish. Programming in Prolog. Fifth edition. Springer. 2003.
- M.A. Covington *et al.* Coding Guidelines for Prolog. Theory and Practice of Logic Programming. 12(6): 889-927 (2012) (Highly recommended).
- P. Gloess. Constraint Logic Programming (PowerPoint format).

For Functional Programming:

- Racket
- Haskell

For Logic Programming

- SWI-Prolog. For Windows users, there is a convenient SWI-Prolog editor.

In IMPERATIVE PROGRAMMING

- the programmer describes **how** to solve a problem with a set of instructions that change a program's state.
  - The execution of every instruction is sensitive to the state of the program, and can change it
- **State** = data stored in variables, data structures, or accessible from external sources
- **Program** = collection of instructions: assignments, changes of mutable data (lists, arrays, etc.), etc.

In DECLARATIVE PROGRAMMING

- the programmer describes **what** relationships hold between various entities.
- **Program** = collection of definitions of functions or relations.

Main programming styles

In Software Engineering, there are 4 main programming styles:

- Two imperative programming styles:
  1. Procedural Programming
  2. Object-Oriented Programming:
     - A refinement of procedural programming
- Two declarative programming styles:
  3. Functional Programming
     - Programming and computing with functions
  4. Logic Programming
     - Programming and computing with relations

Procedural Programming

- **Program** = collection of procedure definitions.
- **Procedure** = parameterized group of instructions, that can be called and executed as a single instruction.
- **Computation** = execution of a sequence of instructions.
- There is **one** program state, that can be changed by all instructions.

Languages that can be used for imperative programming: Fortran, C.

> C is heavily used for systems programming: implementation of operating systems, and applications for specific computer architectures (supercomputers, embedded systems, etc.)

OBJECT-ORIENTED PROGRAMMING (OOP)

- Program state is distributed among objects, which are either
  - instances of **classes** (in class-based OOP: Java, C++, etc.)
  - clones of existing objects, called **prototypes** (in prototyped-based OOP: JavaScript, Lua, etc.)
- Objects encapsulate
  - **data fields** = attributes that characterize their state
  - procedures, known as **methods**.
- Other features of OOP: encapsulation, dynamic feedback, inheritance, etc.

FUNCTIONAL PROGRAMMING (FP)

- **Program** = collection of definitions of functions, datatypes, macros, etc.
- **Computation** = evaluation of an expression using a fixed and predictable **evaluation strategy.**
  - Computation is **stateless**: it does not depend on a program state
  - The result of a function call depends only on the values of input arguments

Most functional programming languages are either

Strict: they implement the **call-by-value** evaluation strategy: Common Lisp, Racket, Scala

Lazy: they implement the **call-by-need** evaluation strategy: Haskell

LOGIC PROGRAMMING (LP)

- **Program** = collection of definitions of predicates using **facts** and **rules.**
- **Computation** = answering a a question (a.k.a. query) using a fixed and predictable **search strategy**.
  A **query** is a conjunction of atomic queries.
  - Typical strategy: **SLDNF resolution**

  Language that can be used for LP: Prolog

Logic Programming (LP)

- **Program** = collection of definitions of predicates using **facts** and **rules.**
- **Computation** = answering a a question (a.k.a. query) using a fixed and predictable **search strategy**.
  A **query** is a conjunction of atomic queries.
  - Typical strategy: **SLDNF resolution**

  Language that can be used for LP: Prolog

Logic programming 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 the built-in search strategy of the language.

- In declarative programming, changing the the value of a variable is disallowed $\Rightarrow$ **no assignment.**

$\Rightarrow$ all repetitive computations are simulated by recursion.

Example (Computing the factorial)

Imperative style

```
int fact(int n) {
 int r=1,i=n;
 while(i>=1) {
   r=r*i;
   i=i-1;
 }
 return r;
}
```

Functional style

```
int fact(int n) {
  if(n==0)
     return 1;
   else
     return n*fact(n-1);
}
```

REMARK: All iterative computations can be simulated by recursion.

Characteristics of declarative programming
2. Data is immutable

Mutable data can be modified after its initial construction, immutable data can not be modified.

- Declarative programming uses immutable data.
  - There is no assignment to change the value of a variable, and there are no operations to modify the content of a data structure (list, array, etc.)
    - Assignment is used only to define an initial value for a variable.
    - Attempts to change the initial value are prohibited.
- Imperative programming uses mutable data.

All declarative programming languages (FP or LP) implement a fixed and **predictable strategy** of computation

- evaluation strategy, in FP
- search for an answer strategy, in LP

Remarks:

- In declarative programming, the **programmer** should focus on writing a program with correct definitions about **what** he knows. He should not care *too much* about **how** the result is found – this is the task of the language designer.
- The **language designer** must implement a strategy that is correct (computes the right answers) and efficient
  - To **improve** the efficiency of computation we should know how the strategy works.

Compare with imperative programming: the programmer must write programs that describe **how** to find the result.

We will illustrate how different programming styles can be used to solve the same, following problem:

Compute/Find the minimum element of a list of numbers, using the following knowledge:

- Fact: The minimum element of a singleton list made of number m is m.
- Rules:
  - The minimum of x and y is x if $x \leq y$.
  - The minimum of x and y is y if $y < x$.
  - 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.

```
minList(L, n)
  r = L[0];
  while i < n do
     r = min(r, L[i]);
  end while
  return r;
```

```
min(a, b)
  if a < b then
     m = a;
  else
     m = b;
  endif
  i = i + 1;
  return m;
```

- **Program** = collection of two procedure definitions:
  - `minList` is not a function in the mathematical sense: It depends on the value of the program variable i.
  - `min` changes the program state (the value of variable `i`)
- **Computation** = the sequence of instructions

$\underbrace{\texttt{i} = \texttt{1}}_{\text{assignment}}$; $\underbrace{\texttt{minList}([4, 2, 5, 1, 6], 5)}_{\text{procedure call}} \Rightarrow$ result 1.

We encode problem-specific knowledge with definitions of functions, in a program

- ```
  minList (x:[]) = x
  minList (x:y)  = minim x (minList y)
  minim x y
     | x <= y = x
     | x > y = y
  ```
- **Computation** = evaluation of the expression `minList [4,2,5,1,6]` ⇒ value 1.
  - The evaluation of the expression is **stateless**: the result of function calls depends only on the values of input arguments.
  ```

We use facts and rules to encode problem-specific knowledge with facts and rules in a program

- ```
  % min(A,B,C) is defined to hold if
  % C is the minimum of A and B
  min(X,Y,X) :- X =< Y.
  min(X,Y,Y) :- Y < X.

  % minList(T,X) is defined to hold if
  % X is the smallest number in list T
  minList([X],X).
  minList([X,Y|T],M) :-
    minList([Y|T],M1), min(X,M1,M).
  ```
- **Computation** = answering the query "who is the minimum element X of list $[4, 2, 5, 1, 6]$?"
  ```
  ?- minList([4,2,5,1,6],X).
  X = 1.
  ```
  ```

What will we learn?

This lecture is intended to familiarize you with the declarative programming styles, by practicing programming with

1. Racket: a strict functional programming language
2. Haskell: a lazy functional programming language
3. SWI-Prolog: a popular implementation of Prolog, for logic programming.

All languages are freely available on all major platforms: Windows, Unix, Mac OS

What will we learn?

This lecture is intended to familiarize you with the declarative programming styles, by practicing programming with

1. Racket: a strict functional programming language
2. Haskell: a lazy functional programming language
3. SWI-Prolog: a popular implementation of Prolog, for logic programming.

All languages are freely available on all major platforms: Windows, Unix, Mac OS

- **We will start practicing functional programming.**

In Functional Programming, functions are pure: there is no program state that is changed by function calls or operations on mutable data.

- ⇒ the same function call always produces the same result.
- ⇒ referential transparency: we can replace "equals by equals" without changing the result of computation.
- ⇒ programs can be verified for correctness, optimized, or parallelized by clever compilers

In Procedural Programming, procedure calls can change program state (e.g., values of program variables)

- ⇒ the same procedure call always can produce different results.
- ⇒ referential opaqity: we can not perform equational reasoning to understand program behavior.

Every (functional) language implements a particular evaluation strategy. Most FP languages implement one of the following two evaluation strategies:

1. **Strict evaluation:** we always reduce the arguments of function calls to values before calling the function. Racket is a strict functional programming language.
2. **Lazy evaluation:** we reduce the arguments of function calls only when they are really needed. Haskell is a lazy functional programming language.

Other evaluation strategies may exist, e.g., **parallel evaluation**.

Consider the function definition $\texttt{double}(\texttt{x}) = \texttt{x} + \texttt{x}$. Also, lazy FP languages know that $0 * \texttt{x} = 0$ for every number $\texttt{x}$.

- Examples of strict evaluations:

  $\texttt{double}(\underline{1+2})=\underline{\texttt{double}(3)}=\underline{3+3}=6$

  $0*(1+\underline{2*3})=0*(\underline{1+6})=\underline{0*7}=0$

  Strict evaluation proceeds bottom-up, from left to right.

- Examples of lazy evaluations:

  $\underline{\texttt{double}(1+2)}=\underline{(1+2)}+(1+2)=3+\underline{(1+2)}=\underline{3+3}=6$

  $\underline{0*(1+2*3)}=0$

  Lazy evaluation proceeds top-down, from left to right.

Values can be

- named,
- passed as argument to a function,
- returned as result of a function call,
- stored in a data structure (e.g., as element of a list)

Values can belong to various datatypes: integers, booleans, strings, lists, pairs, etc.

- In FP, functions are values that belong to the function type $\Rightarrow$ we can define a function that takes function(s) as argument(s) and/or returns a function as value.
  - Such functions are called higher-order functions

Declarative	Imperative
Focus on ”what”	Focus on “how”
Stateless	Uses state
Functions without side effects	Functions with side effects (can change program state)
Uses recursion to iterate	Uses loops and assignment to iterate
Functions are values	Functions are not values.

- “No state, no side-effects” in functional programming help to write bugs-free code or less error-prone code.
- Functional code is compact, easier to maintain, reuse, and test.
- Functional programs consist of independent blocks that can run concurrently $\Rightarrow$ improved efficiency.
- They are close to mathematics, which is advantageous when proving their properties.
- Functions as values are a very powerful programming feature.

Recommended reading:

- J. Hughes. Why Functional Programming Matters. 1984.

The absence of state requires to create new objects whenever we perform actions

$\Rightarrow$ Functional Programming requires large memory space.

$\Rightarrow$ Garbage collection must be used to reclaim memory occupied by objects that become inaccessible.

- Historical note: Garbage collection was invented in 1959 by John McCarthy, the inventor of Lisp, the second-oldest high-level programming language.

- Recursion is usually slower than iteration.
  - This is not so bad: modern languages have efficient garbage collectors $\Rightarrow$ some recursive computations can be as fast as iterative computations.

A word of warning

Functional Programming (FP) and Logic Programming (LP) are **declarative** programming styles:

- Programming = encode “what” you know in a program, without caring too much how the result/answer is computed
  - trust the built-in strategy of the language (FP: evaluation strategy; LP: resolution strategy). which always finds the right result/answer
- **Good thing:** Declarative programs are referentially transparent: they are easy to understand and verify if they are correct (with equational reasoning tools)
- **Bad thing:** Declarative programs can become very inefficient (See next slides)

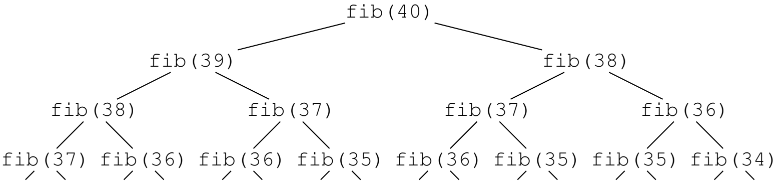
1. Easy to understand recursive definition, but awfully inefficient:

$$\texttt{fib}(\texttt{n}) = \begin{cases} 1 & \text{if } \texttt{n} = \texttt{1} \text{ or } \texttt{n} = \texttt{2}, \\ \texttt{fib}(\texttt{n} - \texttt{1}) + \texttt{fib}(\texttt{n} - \texttt{2}) & \text{if } \texttt{n} > \texttt{2}. \end{cases}$$

2. Less readable recursive definition, but very efficient:

$$\texttt{fib}(n) = \texttt{fibA}(n, 1, 1) \text{ where}$$

$$\texttt{fibA}(n, a_1, a_2) = \begin{cases} a_2 & \text{if } n = 1, \\ \texttt{fibA}(n - 1, a_2, a_1 + a_2) & \text{if } n > 1. \end{cases}$$



⇒ `fib`(40) performs 331 160 281 recursive function calls!

$$\texttt{fibA}(40,1,1) = \texttt{fibA}(39,1,2) = \texttt{fibA}(38,2,3) = \ldots$$
$$= \texttt{fibA}(1,102334155,165580141) = 165580141$$

⇒ `fibA`(40, 1, 1) performs only 39 recursive function calls.

1955: John McCarthy (MIT): proposed the study of Artificial Intelligence (AI): “the science and engineering of making intelligent machines.”

- Inventor of Lisp (1958) = first language with notable functional programming capabilities
  - Second oldest high-level programming language–only Fortran is 1 year older (from 1957)
  - Both Fortran and Lisp are in widespread use today
  - **Lisp** stands for **List** **p**rocessing: linked lists are the main data structure, used to represent both source code (programs) and data.
  - Lists were used mainly for algebraic processing in AI
  - Other data types (besides lists): numbers and symbols
- Initially, Lisp was not standardized: many people developed their own versions of Lisp (a.k.a.Lisp dialects) $\Rightarrow$ standardization became necessary.

There are 2 main dialects of Lisp, standardized and in widespread use:

1. Common Lisp: industrial standard developed by the Lisp community to combine the features from earlier Lisp dialects; became an ANSI standard in 1994
    - Huge, multi-paradigm programming language
2. Scheme: a Lisp dialect developed at MIT for instructional use; became an IEEE standard in 1990 (IEEE 1990), and was recently renamed to RACKET
    - Small, modular, easy-to-learn programming language

We will practice functional programming in RACKET

- The first FP languages were strict: Lisp (1958), Common Lisp, Scheme, etc. Also, most FP languages developed afterwards are strict.
- Lazy FP languages emerged much later: SASL (1972), KRC (1981), Miranda (1985), Lazy ML, etc.
- Haskell (1987) emerged from an effort to standardize the lazy FP languages.

We will practice functional programming in HASKELL

Peculiarities of Racket and Haskell

Racket, and all dialects of Lisp, use a weird syntax to write expressions, called fully parenthesised syntax. For example:

- Instead of $f(v_1, \ldots, v_n)$ we write $(f\ v_1\ \ldots\ v_n)$
- Instead of `if` *cond* `then` $branch_1$ `else` $branch_2$ we write
  (`if cond` $branch_1$ $branch_2$)
  etc.

Haskell requires the usage of parentheses only for two purposes: (1) to disambiguate the order of operator application (e.g., in arithmetic expressions), and (2) to build tuples (a composite datatype). For example:

- Instead of $f(v_1, \ldots, v_n)$ we write $f\ v_1\ \ldots\ v_n$

Most functional languages (including Racket) are strict:

- Whenever we evaluate a function call, we first evaluate all function arguments to values, and then call the function with the values of the arguments:

  EXAMPLE:

  $$(+\ \ (/\ \ 4\ \ \underline{(-\ \ 3\ \ 1)})\ \ (*\ \ 2\ \ 5)) = (+\ \ \underline{(/\ \ 4\ \ 2)}\ \ (*\ \ 2\ \ 5)) = (+\ \ 2\ \ \underline{(*\ \ 2\ \ 5)}) = \underline{(+\ \ 2\ \ 10)} = 12$$

Sometimes, argument evaluation is useless:

$$(*\ \ 0\ \ (/\ \ (-\ \ (\mathrm{sqrt}\ \ (-17\ \ 1))\ \ (-\ \ 3\ \ 1)))) = 0$$

The evaluation of red argument is time-consuming and useless

- Lazy functional languages evaluate only needed arguments
  - Representative language: Haskell (standardized in 1990)

What is Racket?

A strict functional programming language: the entire language is built on top of a few primitive operations for list manipulation.

- enormous volume of educational material which created for it.
- Easy to get.

Easiest way to interact with Racket, is via DrRacket = widely used IDE among introductory Computer Science courses that teach Scheme or Racket

- Freely available for all major platforms: Windows, MacOS, UNIX, Linux with X Window system
- Recommended textbooks:
  - “Structure and Interpretation of Computer Programs”, arguably the best textbook about functional programming.
  - “How to Design Programs” (from `http://www.htdp.org`)

- Every expression is evaluates to a value, by a stepwise process called reduction.
- Values are expressions that evaluate to themselves.
  - Can be primitive or composite
  - A **function expression** is evaluated to a function object
- Racket is dynamically typed:
  - We don't have to declare the types of variables, functions, etc.
  - The interpreter computes the types of expressions at runtime.
- Type = set of values with common properties.
  - type checking is performed at runtime, and can raise runtime type errors.

What is Haskell?

A lazy functional language created in the 1980's by a committee of academicians:

Functional Functions are first-class citizens: they are values which can be used as any other sort of value.

Lazy: Computation = evaluation of expressions using lazy evaluation

- expressions are not evaluated until their results are actually needed

Pure: Expressions are referentially transparent:

- no side effects
- calling the function with same inputs produces the same output every time

Statically typed: every expression has a type, which is checked at compile-time. Programs with type errors will not run because they will not even compile.