Programming Technologies
Master in Informatics
2023–2024

Maria João Varanda Pereira

Polytechnic Institute of Bragança, Bragança, Portugal

February, 2024

Outline

Programming Technologies

Learning Objectives:

- increase the capacity of lay programming solutions
- improve the ability to choose an appropriate programming language
- increase the ability to learn new languages
- better understand the meaning of implementations
- improve the use of programming languages already known
- gain knowledge about the logical and functional paradigm
- increase knowledge of computer science area

Languages

Languages are used to communicate:

- Communication is achieved when the receiver understands the words, phrases and knows its meaning in a certain context .
- The success of communication depends then on several factors:
  - adequacy of the type of language to participants,
  - mutual agreement on the language to use,
  - the issuer's ability to express himself with the right words and well-constructed sentences,
  - the receiver's ability to process the information received and react

Natural Languages versus Programming Languages

- In natural language, used among humans, the same sentence may have very different meanings depending on the intonation, the region or the context in which they are spoken.
- A programming language is a standardized method of defining data structures and express instructions (GRAMMAR).
- The source code (text or drawing program) is then translated into machine code that can be executed.
- To make this translation (program execution) is necessary to build a program that runs in the computer and check and validate the sentence (source code) and execute the action (PROCESSORS).

Programming Languages History

- Pseudocodigos (1940-1950).
- Fortran I,II,III,IV; Fortran 77; Fortran 90; Fortran 95; e 2003 (OO) Imperative Language
- LISP - first funcional language
- ALGOL 60 + COBOL = BASIC
- ALGOL + Fortran + COBOL = PL/I
- APL e SNOBOL: dynamic languages
- ALGOL = Pascal,C,Perl
- Prolog, logic programming (1970)
- Ada: some OO notions
- SmallTalk : first OO language (and visual)
- C++, imperative OO
- Java, simple, safe but less powerful

Modern Programming Languages

- Javascript, PHP, Python e Ruby: scripting languages
- Java e C++ = C#
- XSLT, JSP, HTML, XML, Latex: markup languages
- Coconut, Julia (python)
- Go, Oden
- Swift
- kotlin
- Erlang (Haskell)
- Dart

Low-code Platforms

- Cronapp
- Outsystems
- Mendix
- Appian
- Zoho Creator
- Wordpress

Programming Languages Generations

- First generation: machine language and binary language
- Second Generation: Assembly language
- Third generation: Procedural and Structured languages (Pascal, C).
- Fourth generation: languages that generate programs in other languages (Java, C ++), query languages (SQL).
- Fifth generation: languages based on problem solving using models and constraints (Prolog)

How to measure how easy is to use a new language?

How easy is to learn, to develop, to understand, to evolve, ...

- **abstraction gradient** - Is it possible to code at different abstraction levels;
- **consistency**- When some of the language has been learnt, how much of the rest can be inferred?
- **error-proneness**- Does the design of the notation induce mistakes?
- **visible dependencies**- Are dependencies easy to detect?
- **imposed guess-ahead**- Do programmers have to make decisions before they have the information they need?
- **role expressiveness**- Do programmers able to express everything they need?

How to measure how easy is to use a new language?

and ...

- **viscosity**- How much effort is needed to perform changes?
- **dispersion code level**- Is all the code visible or it is dispersed? How difficult is to connect the parts?
- **closeness of mapping**- How closely does the notation correspond to the problem world?
- **diffuseness**- How many symbols and space does the notation require to produce something or express a meaning?
- **hard mental operations**- How much hard mental processing is at the notation level?
- **secondary notation**- Can the notation carry extra information not related to the syntax such as layout, color and so on?

Programming Language Classification

- the degree of abstraction (low, medium or high);
- type structure (strongly or loosely; statically or dynamically);
- code organization (Monolithic or modular - code division into blocks);
- style of programming (Imperative or Declarative);
- way of run (sequential or concurrent);
- way of expression (textual or visual);
- purpose (general purpose language or domain specific language).

Visual Languages

What is a visual language?

- Medicines Visual Language (A textual grammar of a visual language)
- Labview
- VisualLISA (a visual grammar of a textual language)

DSL vs GPL

- Medicines Visual Language (DSL)
- Labview (GPL)
- VisualLISA (DSL)

GPL vs DSL

- general purpose languages (GPL) C , Java, Fortran
  - Programmers have more experience in this type of languages;
  - There are support manuals and most comprehensive development tools;
  - They are usually lower-level languages;
  - They have more syntax details;
  - And therefore, more prone to error;
- domain-specific languages (DSL) Dot, XAML, FDL
  - usually are more declarative languages, more descriptive, the highest level;
  - the users more easily create correspondence between the program and the problem that meant to solve.
  - They are usually smaller grammar, syntax few details
  - is not as prone to error
  - Easier to learn and understand
  - Improved productivity
  - Easier for maintenance tasks

Examples of DSL- Formating

- HTML
- Dot Language - GraphViz (Att 1)
  - Webgraphviz
  - Exercise1 (Dot code) (Att 2)
  - Exercise2 (Dot code) (Att 3)
  - Dot - GPL implementation
  - Dot Grammar (https://graphviz.org/doc/info/lang.html)
  - Multiple choice exercises (Att 4)
  - More exercises (Last example)

Examples of DSL- Formating

- Latex
  - First Tex,First pdf
  - Article tex,Article pdf
  - Tables tex,Tables pdf
  - Bib File

Examples of DSLs- Knowledge Representation

- FDL - Feature Description Language (Att 5)
  - FDL - GPL implementation
  - Examples (Att 6)
- Ontologies - ONTODL (Att 7)
  - For what: Knowledge extraction; Natural Language Processing; Knowledge management (formalization); Semantic Web; Education; Comunication (Human-Human and Humam-Machine)
  - ONTODL - template (Att 8)
  - ONTODL - music example
  - web development ontology
  - ONTODL - web development example with instances (Att 9)
  - WebOntoDL

Ontology - Example 1

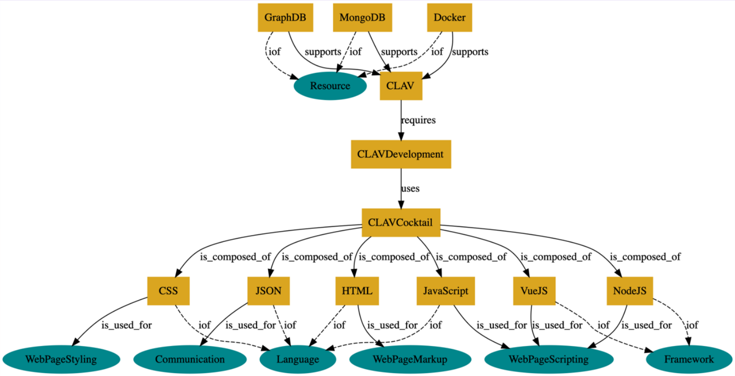


Figure: CLAV Ontology

Ontology - Example 2

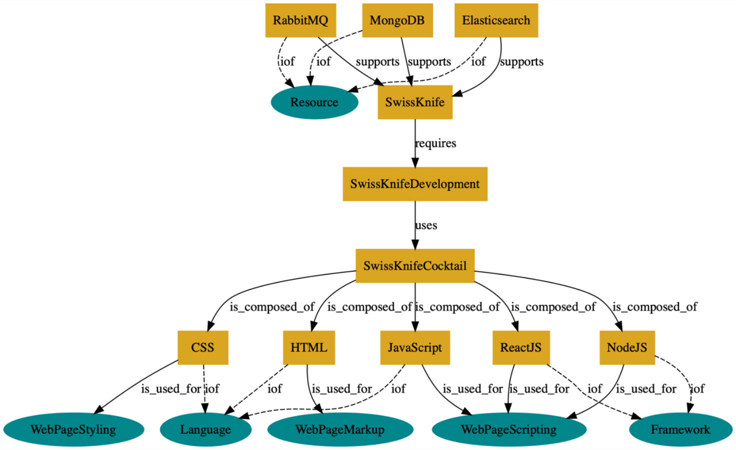


Figure: SwissKnife Ontology

Exercises

- FDL -> dot (GraphViz) (solution-Att 10)
- ONTODL -> dot (GraphViz) (WebOntoDL) (solution-Att 11)
- ClassDG -> Mermaid (Mermaid Syntax + Mermaid Live Editor)

Lets create a new DSL called ClassDG to describe textually class diagrams ...

Exercise 1 - Class Diagrams

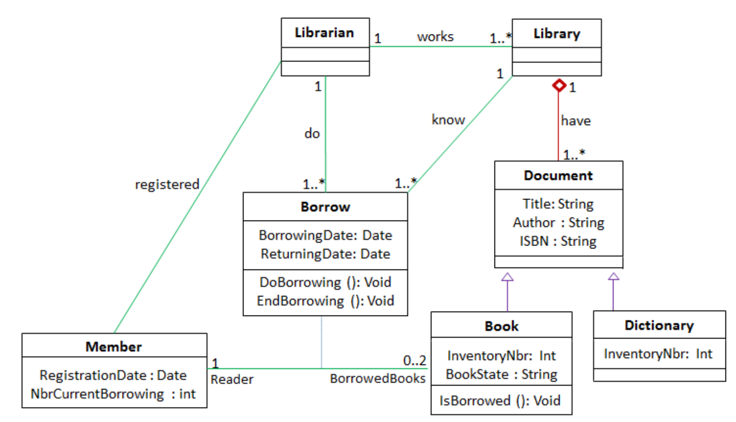


Figure: Library Class Diagram - Att 12

Exercise 2 - Class Diagrams

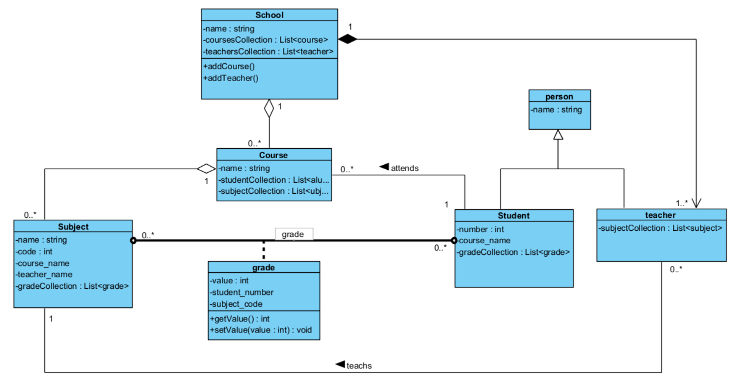


Figure: School Class Diagram - Att 13

End of the first part

Conclusions?

Lessons learned?

Language Processors

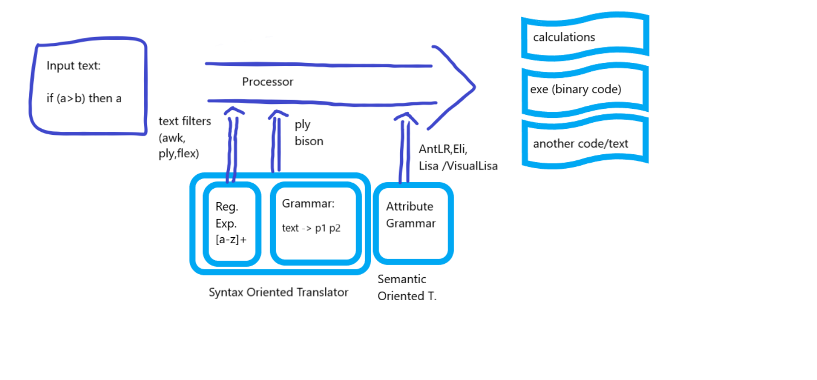


Figure: Language Processors

Basic concepts of Formal Language

- Language - a set of phrases in which each phrase is a sequence (valid) symbols belonging to the vocabulary of the language
- syntactic rules - define possible combinations of symbols
- semantic rules - define the necessary conditions for syntactically correct sentences make sense

Language Processors

- They take the input text, verify if the syntax and semantic is correct and produce an output
- They can be automatically generated if the language is formally specified.
- One of the best known formalisms are regular expressions and grammars

Compilation Process

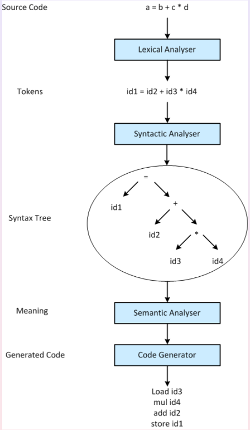


Figure: Compilation Process

Text Filters

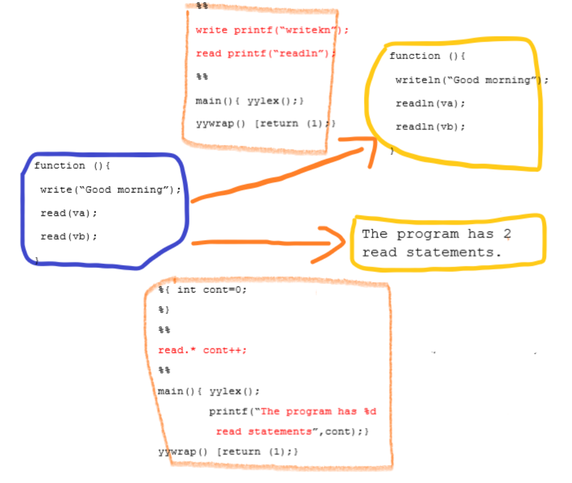


Figure: Text Filters

Syntatic Analysis

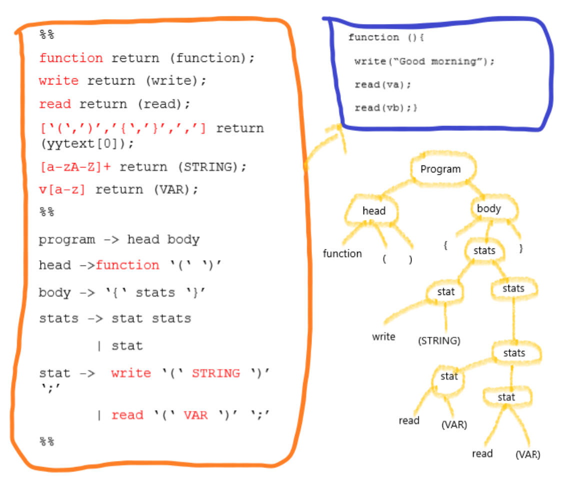


Figure: Syntatic Analysis

Semantic Actions

```
%%
function return (function);
write return (write);
read return (read);
['(',')','{','}',','] return (yytext[0]);
[a-zA-Z]+ return (STRING);
v[a-z] return (VAR);
%%
program -> head body
head ->function '(' ')'
body -> '{' stats '}' {printf("%d\n",cont);}
stats -> stat stats {cont ++;}
       | stat {cont ++;}
stat ->  write '(' STRING ')' ';'
       | read '(' VAR ')' ';'
%%
```

Figure: Semantic Actions

Semantic calculations

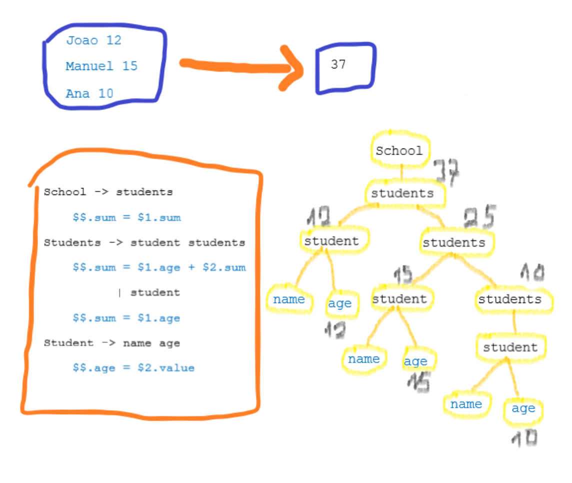


Figure: Semantic calculations

Regular Expressions

Special Character	Meaning
.	every character except newline
\n	newline
*	zero or more copies
+	one or more copies
?	zero or one copy
^	begining of the line
$	end of the line
a/b	a followed by b
a|b	a or b
(ab)+	one or more copies of the group ab
"a+b"	string a+b
[ ]	class of characters

Table: Special characters to descrive patterns

Regular Expressions

Expression	Text
a.	aa, ab, ac, ...
abc	abc
abc*	ab, abc, abcc, abccc, ...
abc+	abc, abcc, abccc, ...
a(bc)+	abc, abcbc, abcbcbc, ...
a(bc)?	a, abc
[abc]	a, b, c
[+-]?	, +, -
[a-z]	any letter from **a** to **z**
[a\-z]	a, -, z
[-az]	-, a, z
[A-Za-z0-9]+	one or more characters (letter or number)
[\t\n]+	blank spaces
[^ab]	anything except **a** or **b**
[a^b]	a, ^, b
[a|b]	a, |, b
a|b	**a** ou **b**
a{5}	aaaaa
a{3,5}	aaa, aaaa, aaaaa

Table: Examples of regular expressions

Exercises with regular expressions

- a regular expression is implemented using a finite automaton that specifies a state machine;
- a regular expression is recognized by the lexer following that automaton;
- Exercises with Regular Expressions(Att14) can be made in order to understand the words that can be represented in each RE and its automaton based implementation.

Grammars

Backus Naur Form

p: A −> BbC

production p: nonterminal A derives into nonterminal B followed by terminal b followed by nonterminal C.

p1: A −> BbC

p2: | C

Nonterminal A derives either in B followed by b followed by C or only in C. This construction has an optional rule for the same symbol.

Grammars

BackusNaur Form

p1: A− >bA
p2: |b

Nonterminal A derives into either b followed by A (right recursion) or only b (stopping case).

This construction means a non−empty list of b's.

p1:A − > bA
p2: |

Nonterminal A derives either in b followed by A or in empty. This construction means an empty list of b's.

Grammars

Extended BackusNaur Form

p: A $->b?$

Nonterminal A derives into either b or empty. This construction means optional existence of a symbol.

p: A $->b+$

Nonterminal A derives into a list of b's. This construction means a non−empty list of b's.

p: A $->b*$

Nonterminal A derives into a list of b's. This construction means an empty list of b's.

Grammars - Example

Non empty list of numbers and words (NEList)

T = num, wrd, '[', ']', ','

num = [0..9]+

wrd = [a..zA..Z]+

N = List, Content, Item

P1: List − > '[' Content ']'

P2: Content − > Item

P3: | Item ',' Content

P4: Item − > num

P5: | wrd

Example of sentences

Valid sequences:

[1,blue,45]

[blue,red,green]

Invalid sequences:

[,23,blue, 34,]

[23,,]

Construct the parsing trees.

Exercises with grammars

- a grammar follows a BNF notation and is implemented using a parsing table;
- the parsing table gives the production that should be used to recognize a valid sentence;
- it is possible to convert ER into grammars and sometimes grammars into RE.
- Exercises with Grammars (Att15) can be made in order to understand the sentences that are valid, the derivation tree that proves that they are valid and a possible conversion to ER can be tested.

Processor Examples

- Text Filters in AWK (Exercises (Att16))
- Text Filters using RE (Python)(Exercises (Att17))
- Text Filters in Ply (Python)(Exercises (Att18))
- Text filters and parsers in Flex/Bison (Lex and Yacc (Att19))
- Parsers in Ply (Python) (Exercises (Att20))

Basic Programming Concepts

- Names
- Variables (memory address, variable value, type, life cycle and scope)
- Binding (static vs dynamic; explicit vs implicit)
- Type verification
- Scope (local, global; static vs dynamic)
- Constants
- Data Types (primitive or compound (strings, arrays, structures, unions and pointers))
- Assignments and expressions
- Commands (conditional, cyclic, jump)
- Subprograms
- Abstraction
- Encapsulation

Imperative Paradigm

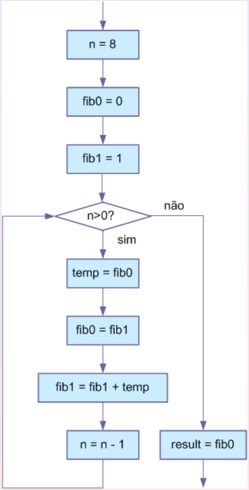
- John von Neumann and others recognized that the program and the data could reside in memory of a computer - Turing machine (1936);
- the machine's memory contains program instructions and data values and the heart of its architecture is the idea of assignment - change the value of a memory location and destroy their previous value;
- In addition, supports variable declarations, expressions, and cyclic conditional instructions and procedural abstraction;
- The commands are executed in the order they are in memory but conditional and cyclic instructions can disrupt this normal flow;

Imperative Paradigm

Imperative program example:

```
fibonacci (int n){
        int fib0, fib1;
        int temp, result;
        fib0 = 0;
       fib1 = 1;
        while(n>0) {temp = fib0;
              fib0 = fib1;
               fib1 = fib1 + temp;
               n = n - 1;
        }
        result=fib0;
}
```

Imperative Paradigm



Object Oriented Programming

- In object oriented programming, the specification is based on the problem domain and the objects can be viewed as a form of abstract modeling of the real world;
- Since the gap between the code and objects controlled by this code is smaller it will be easier coding and understanding programs.
- This type of programming can arise in the context of the imperative paradigm but also in declarative paradigm.
- Initially the objects are identified and subsequently involved operability is based on the exchange of messages between these objects.
- Each object class has a set of possible states (attributes) and behavior (methods).

Object Oriented Programming

The most important concepts involved in this paradigm are:

- Object - distinguishable entity - unique set of attributes and methods (encapsulation)
- Classification - objects with the same data structure (attributes) and the same behavior (methods) are grouped in a class.
- Class - is an abstraction depicting the important properties common to a group of objects; An object is an instance of class with instance variables and methods associated; Each method must have an object as an argument and return an object as a result.
- Attribute - a value of an attribute gives information about the state of that object.
- Method - defines the specific behavior of each object.
- Message - define the object dependencies, indicating the need of services that each object has from other objects.

Object Oriented Programming

```
typedef struct nodo {
INFO inf;
struct nodo *next;
} NODO;

Status Push( NODO **pilha, INFO *inf) {
NODO *novo;
if (!(novo = (NODO *) malloc(sizeof(NODO))))
return INSUCESSO;

novo->inf = *inf;
novo->next = *pilha;
*pilha = novo;
return SUCESSO;
}

Status Pop(NODO **pilha, INFO *inf) {
... }


Status Top(NODO *pilha, INFO *inf) {

... }
```

```
class MyStack{
class Node{
Object val;
Node next;
Node(Object v, Node n)val= v; next=n;
}
Node theStack;

Mystack(){ theStack = NULL;}

boolean empty(){ return theStack == NULL;}

Object pop(){
Object result = theStack.val;
theStack = theStack.next;
return result;
}
Object top(){ return theStack.val;}

void push(Object v){
theStack = new Node(v, theStack);
}
}
```

Object Oriented Programming

- Foto Machine
- Vending Machine

Other modular ways of programming

- Web programming - the code is executed embedded in an html file and is divided in front–end and back–end;
- Event Oriented Programming - based on methods init() and action()
- Aspect Oriented Programming - concept of point cut that defines join points and advice that sets the code to run on each join point.
- Concurrent Programming - parallel execution of code (start(), stop() and run())
- Scripting Programming - execution of code inside other code (<script>)

Aspect-oriented programming

```
main(){ I1; setI2; I3; I4; setI5; I3; I4; I8; setI3; I3; I4;}

f() { I3; I4; }
main(){ I1; setI2; f(); setI5; f(); I8; setI3; f();}

after set.*() : call f();
main(){ I1; setI2; setI5; I8; setI3; }
```

Declarative Paradigm

- minimizes or eliminates side effects by describing what the program must accomplish in terms of the problem domain
- expresses the logic of a computation without describing its control flow
- a declarative program describes what computation should be performed and not how to compute it

Specification Languages

- Domain specific languages that specify structures of knowledge;
- Some examples: FDL, dot, markup languages (Latex, HTML, XML, etc).

Modeling Languages

- Modeling languages are used to design system and some of them allow the generation of code;
- The Unified Modeling Language (UML) is a modeling language that is intended to provide a standard way to develop and visualize the design of a system.

Prolog Language

- A program defines what is real and presents a set of rules that allows the inference of other facts.
- These predicates allow to solve problems in a question format.
- These issues are addressed and answered by a machine that is always based on the same algorithm to search and test and works on the basis of knowledge defined by the program clauses.

Prolog Language

Prolog predicates: close and open atoms form Horn clauses (facts or rules) father(john,mary)
father(john,X)
A Horn clause always begins with an atom which is called head clause. This is separate from the body of the clause for `:-`. The body of the clause is a finite sequence of atoms separated by commas. If the body is empty the clause is a fact (always true). If it is not empty it's a rule.

```
father(john,mary).
grandfather(john,X) :- father(john,Y), son(X,Y).
```

If all the body atoms are true then the head of the atom is true. A logic program is a sequence of terms which may be grouped into packages that have the same atom head.

Prolog Language

- Unification
- Proof Trees
- Search Trees
- Backtracking process
- Data strutures- compound predicates
- Arithmetic expressions
- Cut operator
- write and read operations
- Lists

Prolog Language

```
fibonacci(0,0).
fibonacci(1,1).
fibonacci(N,F) :- N > 1, N1 is N-1, N2 is N-2,
                  fibonacci(N1,F1), fibonacci(N2,F2),
                  F is F1+F2.
```

Prolog Language

```
init :- assert(photo_mach(24, true)).
buy_memory(N) :- retract(photo_mach(Nphoto,_)), Xphoto is Nphoto+N,
                 assert(photo_mach(Xphoto,_)).
take_photo :- photo_mach(Nphoto,_), Nphoto>0, retract(photo_mach(Nphoto,_)),
                 Xphoto is Nphoto-1, assert(photo_mach(Xphoto,_)).
```

Prolog Exercises

- Proposed Exercises-Part 1 (Att21)
- Proposed Exercises-Part 2 (Att22)

Haskell Language

- In this paradigm, the computation is seen as a mathematical function. There is no program status concept (as in the imperative), it is not necessary to make assignments and the effect of cycles is achieved using recursion.
- The programs are seen as a mapping of input values in the output values (functions); functions call other functions and the result of one function can be argument of the other.
- No variable or commands, only expressions, functions and declarations.

Haskell Language

```
fib 0 = 0
fib 1 = 1
fib n | n>=2 = fib(n-2) + fib(n-1)
fib(n) = if(n==0) then 0
                   else if(n==1) then 1
                             else fib(n-1)+fib(n-2)
```

Haskell Language

```
type Photo_mach = (Number, Battery)
type Number = Integer
type Battery =Bool
my_mach :: Photo_mach
my_mach = (24, True)
buy_memory (n,b) x = (n+x,b)
take_photo (n,b) = (n-1,b)
```

Haskell Language

- Nonrecursive functions
- Recursion
- Lists
- Accumulators
- Higher Order Functions
- Data Structures
- Classes and Polymorphism
- Monads

Haskell Exercises

- Proposed Exercises (Att23)

Conclusion

This is the end ...