Programming languages and paradigms

Mirjana Ivanovic
University of Novi Sad
Serbia
mira@dmi.uns.ac.rs

Programming languages and paradigms

- Program - a formal description of a computation process or specification of computation in a programming language
- Programming languages are **formal languages**
  - precise and unambiguous description of lexics, syntax and semantics
    - Lexics - a set of rules for forming proper words in a language
    - Syntax - a set of rules for forming meaningful sentences combining proper words
    - Semantics - a description of the meaning of syntactic constructions
- Based on above definition there are 2 essential programming paradigms **(approach to programming, set of basic ideas and principles for groups of similar languages)** :
  - **imperativne** – description of the process of computation
  - **deklarative** – description of the specification of computation

Imperative programming languages

- **Program** - a set of commands that change the values of a set of variables during execution (program state)
- **Assignment Command** - basic command to assign a value to a variable
- **Control flow commands** (if-then-else, switch, while, do-while, for)
- Categories of imperative programming languages
  - **Procedural** - decomposition of programs into functions and procedures, local and global variables
  - **Modular** - decomposition of the program into modules, the module contains logically related definitions of variables, functions and data types, private and public part of the module
  - **Object-oriented** - decomposition of programs into classes, class contains logically related definitions of variables and functions, classes as data types (object types), private and public part of the class, inheritance of classes
- **Structured programming style** - imperative programming without the use of jump and interruption commands

Declarative programming languages

- Program is the specification of computation (problem description), we do not describe the flow of computation (problem solving)
- **Functional programming languages**
  - **Program** – **set of "pure" functions composed of expressions** instead of commands
  - **Composition of functions and recursive functions**
  - There is **no variables**, only **immutable parameters of functions and identifiers** for which expressions are binding (immutable variables)
  - **functions** – functions can be parameters and / or results of other functions, a function can be part of a data structure (e.g. list of functions)
- **Logical programming languages**
  - **Program – a set of logical formulas** that describe the **properties of objects and relations between objects** (facts) and **relations between relations** (rules)
  - **Built-in inference mechanisms** which allows giving **answers** to the user's **based on the facts and rules** given in the program

Imperative and declarative PL

- Both imperative and declarative programming languages are **high-level programming languages** (machine-independent languages)
- **Abstraction allows to produce more readable and concise programs** in which irrelevant (technical) details are neglected.

- **Declarative programming languages are more abstract** than imperative ones
  - Imperative programming languages make abstraction based on concrete set of machine commands
  - Declarative programming languages **make abstraction based on**
    - **concrete set of machine commands**
    - **Model of computer** as machine which executes commands that change state of memory **(abstraction of flow of program execution)**

QuickSort

```java
public static <E extends Comparable<? super E>> List<E> quickSort(List<E> arr) {
    if (arr.isEmpty())
        return arr;
    else {
        E pivot = arr.get(0);

        List<E> less = new LinkedList<E>();
        List<E> pivotList = new LinkedList<E>();
        List<E> more = new LinkedList<E>();

        // Partition
        for (E i: arr) {
            if (i.compareTo(pivot) < 0)
                less.add(i);
            else if (i.compareTo(pivot) > 0)
                more.add(i);
            else
                pivotList.add(i);
        }

        // Recursively sort sublists
        less = quickSort(less);
        more = quickSort(more);

        // Concatenate results
        less.addAll(pivotList);
        less.addAll(more);
        return less;
    }
}
```

← Java

Haskell

```haskell
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

Introduction to Prolog Language

Chapter 1 – Introduction to Prolog

1.1 Defining relation by facts

1.2 Defining relations by rules

1.3 Recursive rules

1.4 How Prolog answers questions

1.5 Declarative and Procedural Meaning of Programs

1.1 Defining Relations by facts

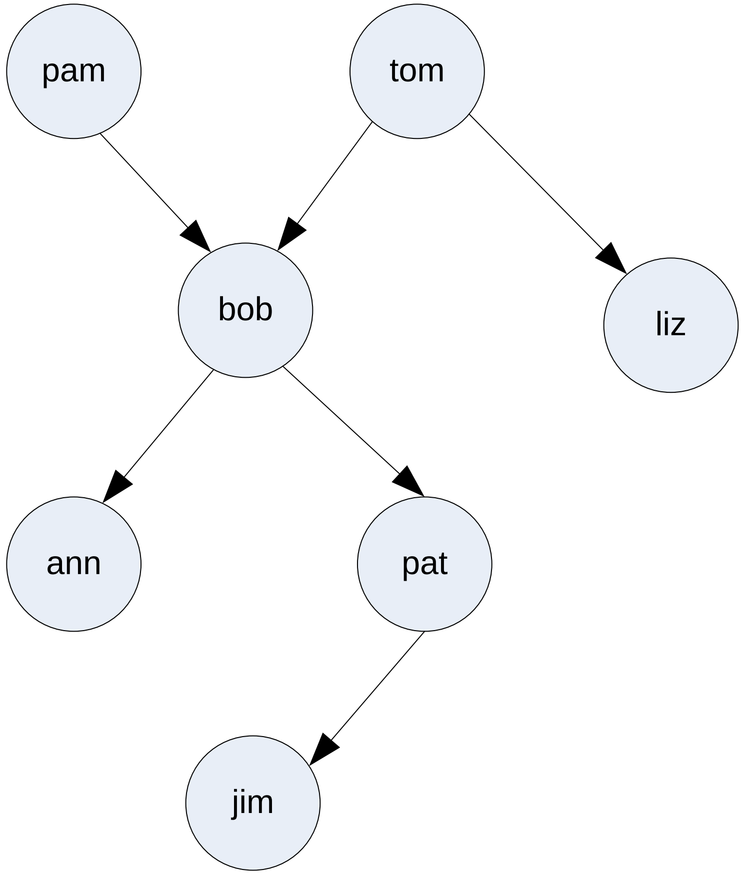
- Prolog (programming in logic) is a programming language for symbolic, non-numeric computation
- Specially suite for solving problems that involve objects and relations between objects.

When we tried to say tom is a parent of bob

tom and bob are objects and parent is a relation between object tom and bob

In prolog, we can write like **parent(tom,bob).**

Example: Family Tree



parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob, ann).
parent(bob,pat).
parent(pat,jim).

Instances or relationships

A relation is defined as a set of all its instances

How to ask Prolog?

- ?- parent(bob,pat). ☞ yes
- ?-parent(liz, pat). ☞ no
- Using Variables – defined as Capital Letter

  ?-parent(X,liz).

  - ☞ X=tom

  ?-parent(bob,X).

  - ☞ X=ann *if more than one answer, press ; to get others or press enter to stop*
  - ☞ X = pat

  ?-parent(X,Y).

- ☞ Using , to make conjunction (and)
  - ☞ Who grandparent of jim?
  - ☞ ?- parent(Y,jim), parent(X,Y).
- ☞ Using ; to make disjunction (or)
  - ☞ ?-parent(Y,jim);parent(Y,pat).

Anonymous variable

- Suppose we want to find out if Hazel is a mother but we do not care whose mother she is:

  ?- mother(hazel,_).

  Matches anything, but never has a value.

- The values of anonymous variables are not printed out.
- Successive anonymous variables in the same clause do not take on the same value.

Anonymous variable

- Use it when a variable occurs only once and its value is never used.

```
is_a_grandmother(X):-mother(X,Y),
parent(Y,_).
```

Cannot be anonymous because it has to occur in 2 places with the same value.

Summary

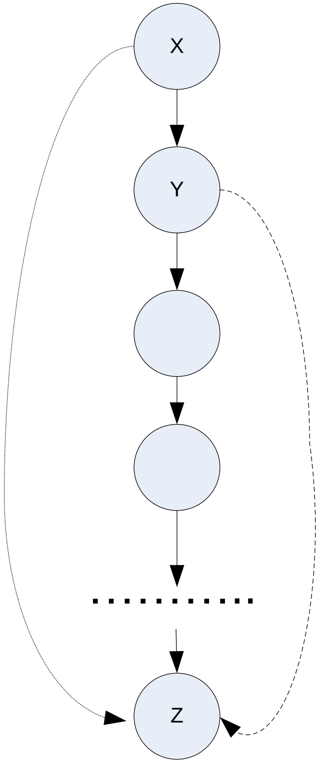
- Use Prolog to define a relation
- User can ask back the relation defined in Prolog program
- Prolog consists of *Clauses*.
- Each clause terminates with a full stop.
- There are concrete object or constants (such as tom, ann) and are called ***atom***
- General objects (such as X, Y –starting with capitals) called ***variable***.
- Questions to the system consists of one or more goals.

1.2 Defining Relations by rules

- Prolog clauses are three types: ***facts, rules and questions***
- ***Facts*** declares things that are always unconditionally true e.g male(bob).
- **Rules** declare things that are true depending on a give condition
  - e.g grandparent(X,Z):- parent(X,Y),parent(Y,Z).
  - Right-hand side is called a condition part or body
  - Left-hand side is called a conclusion or head
- **Questions** – The user can ask the question what things are true.

1.3 – Recursive rules

- Sometimes, we need to write recursive rules in prolog, like
- Predecessor case
  - predecessor(X,Z):-parent(X,Z).
  - predecessor(X,Z):-
    parent(X,Y), predecessor(Y,Z).



Putting Comment:
/* */ => between those /* and */ are comment
% => starting from % to end of line is comment

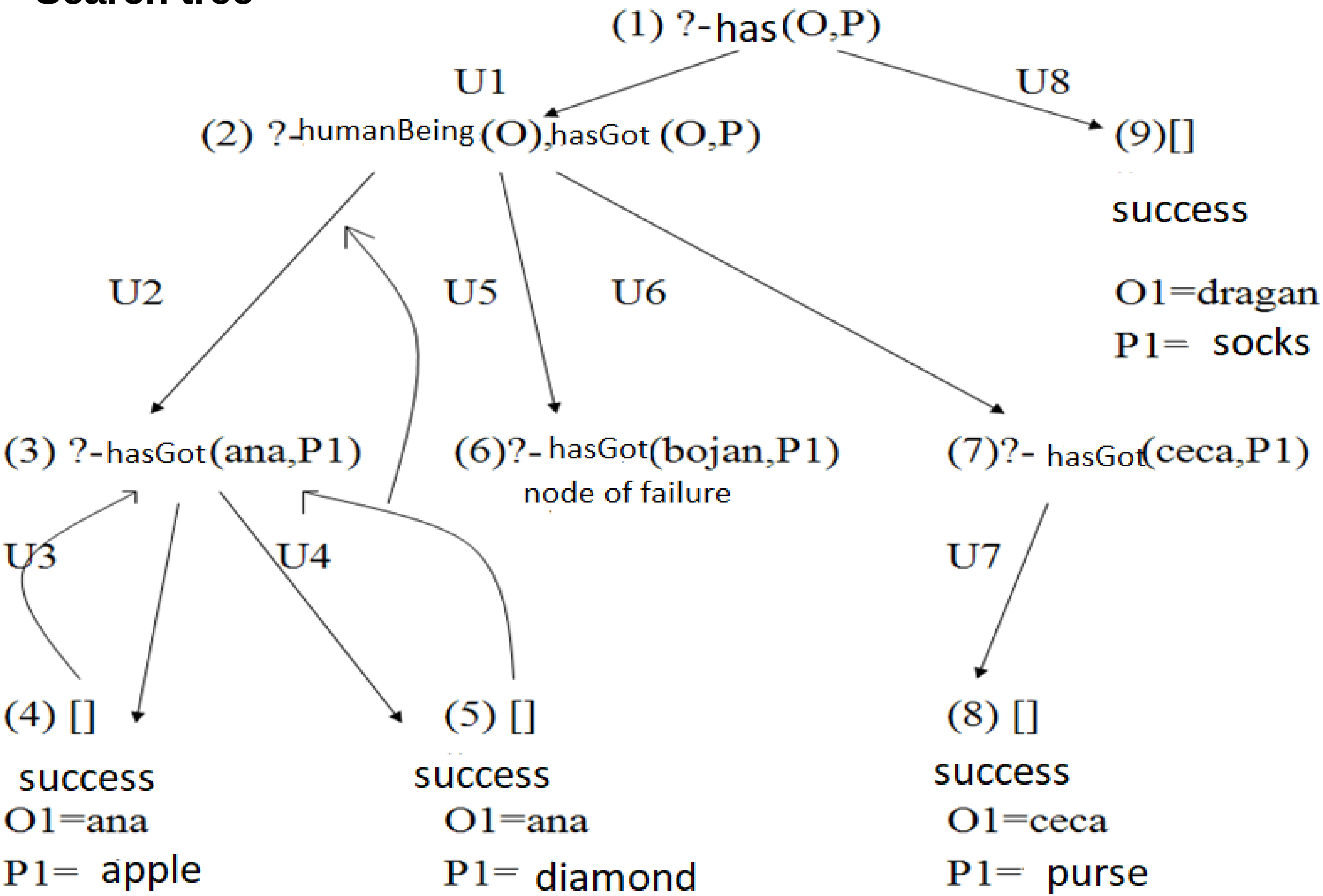
How prolog answer questions Informal explanations

- Prolog seeks for the goals provided by the user as questions
- Prolog searches the successful path and if it reaches unsuccessful branch, it backtracks to previous one and tries to apply alternative clauses
- That why, there is some important clues to write program to run faster.

Example.

```
has(O, P):-humanBeing(O),hasGot(O, P).
has(dragan, socks).
humanBeing(ana).
humanBeing(bojan).
humanBeing(ceca).
hasGot(ana, apple).
hasGot(ana, diamond).
hasGot(ceca, purse).
```

Search tree



(1) ?-has(O,P)
U1
U8
(2) ?-humanBeing(O),hasGot (O,P)
(9)[]
success
O1=dragan
P1= socks
U2
U5
U6
(3) ?-hasGot(ana,P1)
(6)?- hasGot(bojan,P1)
node of failure
(7)?- hasGot(ceca,P1)
U3
U4
U7
(4) []
(5) []
(8) []
success
O1=ana
P1= apple
success
O1=ana
P1= diamond
success
O1=ceca
P1= purse

Declarative and Procedural Meaning of Programs

- **Declarative Meaning** – is concerned only with how the relations is defined by the program or what will be the output of the program

- **Procedural Meaning** – is concerned with how the relations are evaluated by the prolog system or how this output is obtained

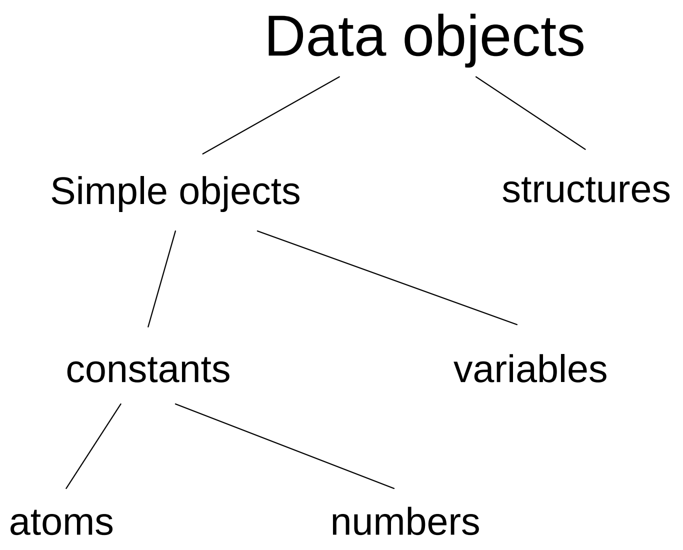
Suggestion: Write program in declaration way and don't worry about how does it compute to obtain the goals. It would be Prolog program development

Summary

- Prolog programming consists of defining relations and querying about relations
- A program consists of clauses, and there are three types: ***facts, rules*** and ***questions.***
- A relation can be specified by ***facts***
- A procedure is a set of clauses about the same relations.
- Two types of prolog meanings: declarative and procedural meaning

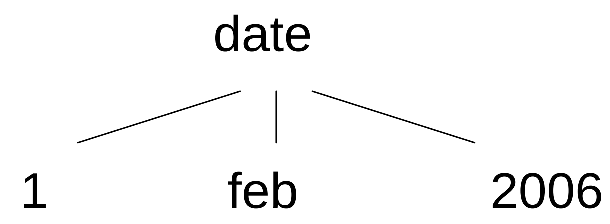
Chapter 2- Syntax and Meaning of Prolog Program

- Data Objects – is composed of simple objects, structures, constants, variables, atoms and numbers.
  - **Atoms and number**
    - Atoms can create in three ways:
      - (1) String of letters, digits and the underscore character, '_', starting with a lower case letter
      - (2) String of special characters, e.g <--->
      - (3) String of characters enclosed in a single quotes, like 'Tom'
  - **Variables** – can create with string of letter, digits and the underscore character, but starting with upper case character or underscore characters.
    - E.g X, _x
    - Anonymous variables, used as underscore, eg. _
      - ?-parent(X,_).
  - ***Lexical Scope*** – all variables are scoped in one clauses and all atoms are scoped to the whole program



Structures

- Are objects that have several components
- The components themselves can be structure.
  - e.g date(1,feb, 2006). or date(Day,feb,2006).



- Also called **structure** as terms in syntactically and it can **represent as tree**
- The **root** of tree is called **funtor** and the **subtrees** are called **arguments**
- Each functor is defined with two things
  (1)The name, whose syntax is that of atoms;
  (2)The arity- the number of arguments

Matching - Unification

- **Match** – given **two terms**, they **are identical** or the **variables in both terms can have same objects after being instantiated**
  - E.g date(D,M,2006) = date(D1,feb,Y1) means
    - D=D1, M=feb, Y1=2006

- General Rule to decide whether two terms, S and T match are as follows:
  - If S and T are constants, S=T if both are same object
  - If S is a variable and T is anything, S=T
  - If T is variable and S is anything, T=S
  - If S and T are structures, S=T if
    - S and T have same funtor
    - All their corresponding arguments components have to match

Declarative and Procedural Way

- Prolog programs can be understood two ways: declaratively and procedurally.
- P:- Q,R

- Declarative Way
  - P is true if Q and R are true

- Procedural Way
  - To solve problem P, first solve Q and then R (or) To satisfy P, first satisfy Q and then R

What is difference?

- Procedural way does not only define logical relation between the head of the clause and the goals in the body, but also the order in which the goal are processed.

Declarative meaning

- **Determine whether a given goal is true**, and if so, for what values of variables it is true.

- **An instance of a clause C is the clause C with each of its variables substituted by some term.**

- A **variant of a clause C** is such an instance of the clause C where **each variable is substituted by another variable**.
  - E.g hasachild(X):-parent(X,Y).
  - Two variants are:
    - hasachild(A):- parent(A,B).
    - hasachild(X1):-parent(X1,X2).
  - Instance of this clause are:
    - hasachild(peter):-parent(peter,Z).
    - hasachild(barry):-parent(barry,small(caroline)).

Formal Declarative Meaning

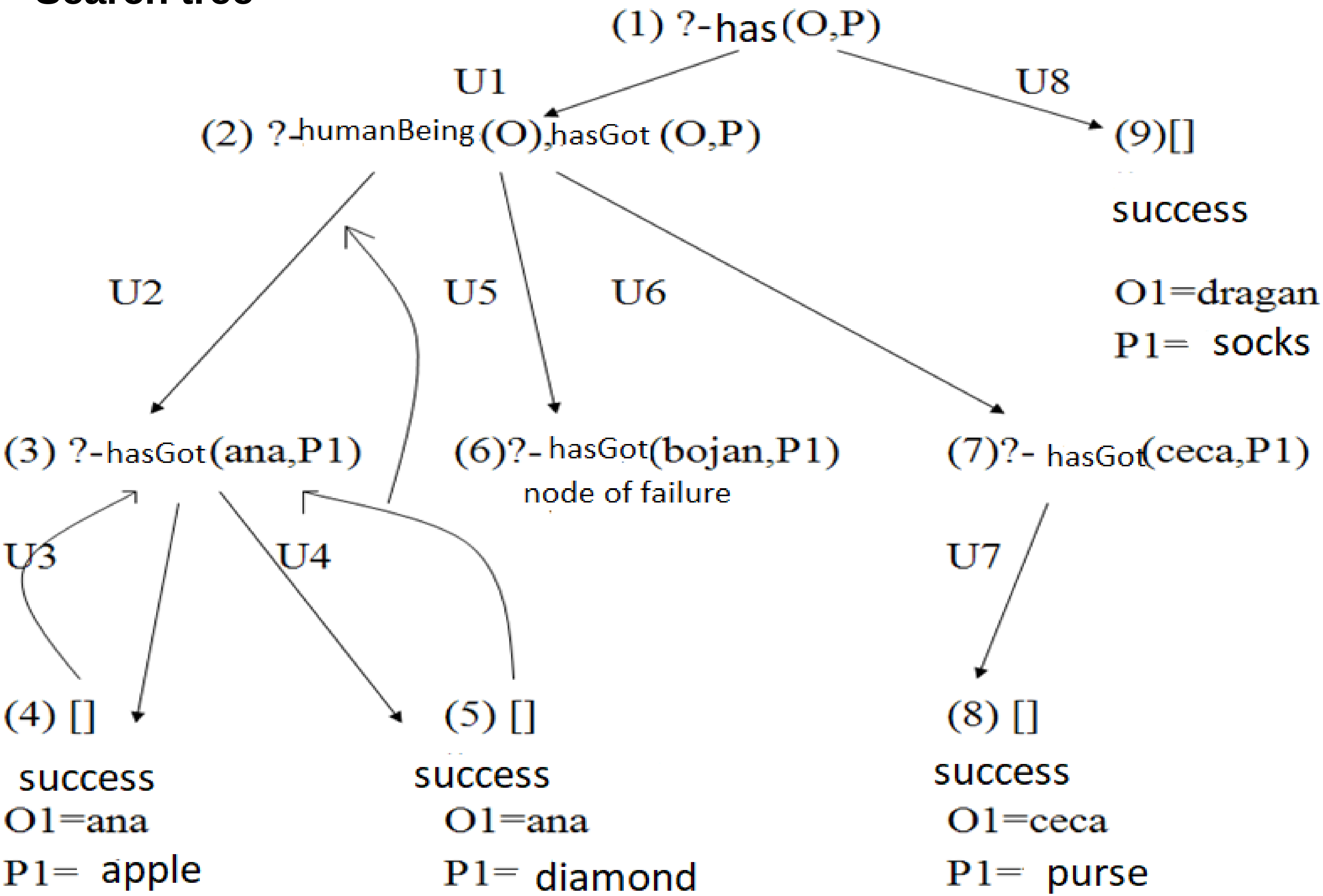
- Given a program and a goal G,
- A goal G is true (that is satisfiable, or logically follows from the program) if and only if:
  - There is a clause C in the program such that
  - There is a clause instance I of C such that
    - The head of I is identical to G, and
    - All the goals in the body of I are true.

Conjunction= , and disjunction = ;

Procedural Meaning

- Specifies how prolog answer questions
- To answer a question means to try to satisfy a list of goals
- A procedure for executing (or) satisfying a list of goals with respect to a given program.

Search tree



(1) ?-has(O,P)
U1
U8
(2) ?-humanBeing(O),hasGot (O,P)
(9)[]
success
O1=dragan
P1= socks
U2
U5
U6
(3) ?-hasGot(ana,P1)
(6)?-hasGot(bojan,P1)
node of failure
(7)?- hasGot(ceca,P1)
U3
U4
U7
(4) []
success
O1=ana
P1= apple
(5) []
success
O1=ana
P1= diamond
(8) []
success
O1=ceca
P1= purse

Monkey and Banana

- Problem – In the middle of the room, there is a banana hanging on the ceiling and the monkey tries to reach by using a box.

- Approach
  - Initial states
    - Monkey is at the floor
    - Money is on the floor
    - Box is at window
    - Monkey does not have banana
  - Four types of move
    - Grap banana
    - Climb box
    - Push box
    - Walk around

Monkey and Banana(Cont'd)

```
move(state(middle,onbox,middle,hasnot),          % before move
        grasp,                                   % grap banana
        state(middle,onbox,middle,has)).         % After move
move(state(P,onfloor,P,H),
        climb,                                   % climb box
        state(P,onbox,P,H)).
move(state(P1,onfloor,P1,H),
        push(P1,P2),                             % push box from P1 to P2
        state(P2,onfloor,P2,H)).
move(state(P1,onfloor,B,H),
        walk(P1,P2),
        state(P2,onfloor, B,H)).
canget(state(_,_,_,has)).                        % can 1: Monkey already has it
canget(State1):-                                 % do somework to get it
    move(State1,Move,State2),                    % do something
   canget(State2).                               % Get it now
```

?- canget(state(atdoor,onfloor,atwindow,hasnot)). => Yes

Way of Satisfying the goal in procedural way

- If the goal list is empty -> Success
- If not, scan all clauses from top to bottom to find, the head to match with the goal. If no match found and end of program, failure
- If found, generate variant of the goal and instantiate all variables from that goal to all reminding goal lists
- Execute recursively the new goal list until it reaches success or failure.

Example

- big(bear).
- big(elephant).
- small(cat).
- brown(bear).
- black(cat).
- gray(elephant).
- dark(Z):-black(Z).
- dark(Z):- brown(Z).

- ?-dark(X),big(X)

1. Initiate goal list: dark(X),big(X).
2. Scan to find dark(X)
    1. Found dark(Z):-black(Z).
    2. New goal black(X),big(X)
3. Scan 2nd goal black(X)
    1. Found black(cat).
    2. New goal black(cat),big(cat).
4. Go to second goal big(cat)
    1. No found, so go back to black(X), big(X) and scan -> no found
5. Go back to dark(X), big(X) with dark(X) again
    1. Found dark(Z):- brown(Z).
    2. New goal brown(X), big(X).
6. Scan and found borwn(bear). So the goal shrink to big(bear).
7. Found big(bear)
8. Provide X=bear.

Orders of Clauses and Goals

- Danger of indefinite looping eg p:- p.

- When happened?.
  - **Declarative way is correct, but procedural way is wrong.** So, there is actually answer, but cannot reach from program.

- So how to avoid it -> many special techniques

- **Carefully to rearrange**
  - The order of clauses in the program
  - The order of goals in the bodies of the clauses

So, how to program Prolog

- Do declarative way to program because it is easier to formulate and understand
- Prolog will help you to get procedural work
- If fails, rearrange the order of clauses and goals into suitable order from procedural aspect

Representation of Lists

- **List is a data structure** and is either **empty or consists of two parts**, called a head and a tail and can be represented as
  - [X,Y,Z].
  - [Head | Tail].
  - .(Head,Tail). Where Head is atoms and Tail is in list
  - We can write [a,b,c] or .(a,.(b,.(c,[]))).
- List is handled as binary tree in Prolog

List Operations

- Checking some objects is an element of a list -> **member**
  - e.g member(b,[a,b,c]). => true
  - member(b,[a,[b,c]]). => false
- Concatenation -> conc(L1,L2,L3).
  - conc([a,b,c],[1,2,3],L).=> L = [a,b,c,1,2,3]
- Adding item into list => add(X,L,L3).
  - add(a,[b,c],L) => L=[a,b,c]
- Deleting Item => del(X,L,L1).
  - del(a,[a,b,c],L). => L=[b,c]
- sublist => sublist(S,L).
  - Sublist([a],[[a],b,c]) => true
- Permuntation => permutation(L,P).
  - Permutation([a,b],P). => P = [a,b]; P=[b,a]

Operator Notation

- Can define new operator by inserting special clauses called directives, e.g :-op(600,xfx,has).

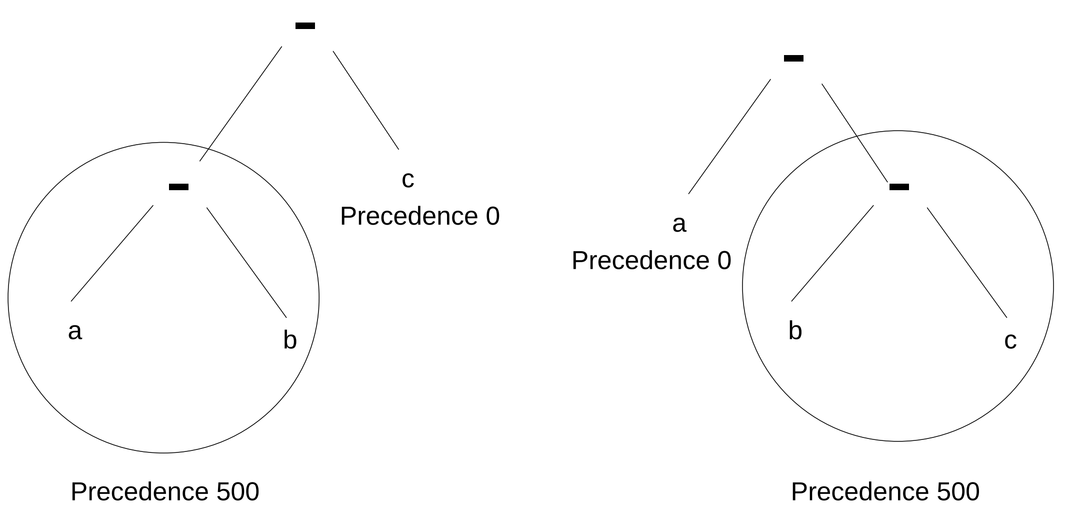
- **:-op(precedence,type of operator, functor).**
  - **Precedence is between 1 to 1200**
  - **Type of operator denoted with f**
  - **Functor -> operator name**

- Three group of type of operator
  - Infix operator -> xfx , xfy, yfx
  - Prefix operator -> fx, fy
  - Postfix operator -> xf, yf
  - x represents an argument whose precedence must be strictly lower than that of the operator
  - y represents an argument whose precedence is lower or equal to that of the operator

- If an argument is enclosed with parentheses or it is an unstructured objects, then precedence is 0.

- If argument is structure then, its precedence is equal to the precedence of its principal functor.

Operator Notation (Cont'd)



For a – b – c case, assume that – has precedence of 500
Then, if – is yfx type, the right interpretation is not correct because the precedence of b – c is not less than the precedence of – . Thus, use (a-b) –c

Summary

- Readability of the program can be improved by infix, prefix or postfix
- Operator definition introduces new notation. Operator called functor holds together components of structures
- A **programmer can define his or her own operators. Each operator is defined by its name, precedence and type**
- Precedence is an integer within some range usually from between 1 to 1200.
- The operator with the highest precedence in the expression is the principle functor of the expression
- Operator with lowest precedence binds strongest
- The type of an operator depends on two things:
  - The position of the operator with respect to the argument
  - The precedence of the arguments compared to the precedence of the operator itself.
  - xfy -> x indicates an argument whose precedence is strictly lower than that of operator and y indicates an argument whose precedence is less than or equal to that of the operator

Arithmetic

- Basic arithmetic opeartors are
  + = addition
  - = substraction
  * = mutiplication
  / = division
  ** = power
  // = integer division
  mod = modulo

e.g ?- X=1+2. => X = 1 + 2
      ?- X is 1 + 2. => X = 3

So, is is operator for arithmetic expression
?- X is 5/2, Y is 5//2, Z is 5 mod 2.
X=2.5
Y=2
Z = 1

Comparison Operator

- X > Y => X is greater than Y
- X < Y => X is less than Y
- X >= Y => X is greater than or equal to Y
- X =< Y => X is less than or equal to Y
- X =:= Y => the X and Y values are equal
- X =\= Y => the X and Y values are not equal

- == tests whether its arguments already have the same value.
- = attempts to unify its arguments with each other, and succeeds if it can do so.
- With the two arguments instantiated, the two equality tests behave exactly the same.

Summary

- List is either empty of consists of a head, presented as atom and a tail which is also a list.
- membership, conc, add, del
- The operator notation allows the user to tailor the syntax of programs toward particular needs and also improve readability
- New operators are defined by the directive **op**, stating the name of an operator, its type and precedence.
- Arithmetic is done by built in procedure. Use **is** procedure to evaluate and comparison with <, =< etc

```
% adding the item X to the binary dictionary T1
% (element, binary-dictionary, binary-dictionary)

add(X,nil,t(X,nil,nil)).
add(X,t(Root,L,R),t(Root,L1,R)) :- X @< Root, add(X,L,L1).
add(X,t(Root,L,R),t(Root,L,R1)) :- X @> Root, add(X,R,R1).
```

Functional programming languages, Haskell

Functional programming languages

- **FP program – set of "pure" functions composed from expressions**
  - Principle of **referential transparency**
    - **Expression/function has always the same value for the same value of its arguments, independent on context in which expression/function is evaluated**
  - Function – expression is assigned to the name of function for some input parameters
  - Function gets a value when it is invoked by some concrete values of parameters, **no side-effects**
  - Expression is application of a function or operator on some arguments
    - Arguments can be expressions → make function compositions, recursive functions

Functional programing languages

- **Abstraction of flow of execution**
  - No commands and variables
    - Immutable function parameters
    - Immutable local variables
  - **Built-in mechanisms of expression evaluation, no need to know how it functions**
  - Conditional expression – expression value depends on value of some other sub-expression
  - Recursion instead of loops
- **Evaluation of FP program starts with a function application on concrete values of arguments**

Characteristics of Functional PL

- **FP abstracts the flow of program execution**
  - Shorter and more concise programs comparing to imperative programming
  - Higher degree of abstraction → smaller number of details → smaller possibility to make errors

- **Referential transparency of functions**
  - Smaller possibility to make errors
    - No side effects
  - Better formal analysis and validation of programs
  - Greater possibility for program parallelization
    - Subexpressions which are arguments of some other expression can be evaluated in parallel.

Higher-Order Functions

- **Higher-order functions can have functions as arguments, or their results are functions or both**
  - Example: derivation, integral
- Example.

```
function inc(x) = x + 1
function twice(f, x) = f(f(x))
twice(inc, 5)  → 7
```

- Three typical higher-order functions
  - **map f l** – apply function f on each element of its argument which is list l
  - **filter f l** – filter list l based on logical function f
  - **fold f l n** – reduces list l according to operator (binary function) f, n is neutral element of operator f
- Functions as elements of a data structure

Strict and ne-strict semantics

- **Strict semantics**
  - Expression (function) can be evaluated in some value only if all its subexpressions (arguments) can be evaluated in some values
  - ***Strict/eager evaluation, call by value***: expression value (function) can be evaluated after all its subexpressions (arguments) are evaluated
  - **Imperative programming languages are based on strict semantics, excluding logical expressions**
- **Non-strict semantics**
  - Expression (function) can be evaluated even if some its subexpressions can not be evaluated
  - ***Non-strict (lazy) evaluation, call by need***: Expression (function) is evaluated only if its value is needed
  - **Lazy FP languages: FP languages that support non-strict semantics (Miranda, Haskell)**

Strict and non-strict semantics

- **Examples.**
  - (x = 0) or (1 / x = 5)
    - for x = 0 expression has no value in strict semantics
    - In non-strict semantics it has valie true
  - length [2, 2 + 4, 6 / 0, 2 + 3 * 4]
    - in strict semantics function can not be evaluated as third expression can not be evaluated
    - In non-strict semantics elements of list are not evaluated, as function returns length of the list
  - function sqr(x) = x * x, evaluate sqr(2 + 3)
    - **Eager evaluation.** sqr(2 + 3) → sqr(5) → 5 * 5 → 25
    - **Lazy evaluation.** sqr(2 + 3) → (2 + 3) * (2 + 3) → 5 * 5 → 25

Infinite Data Structures

- **Non-strict semantics offer possibility to work with infinite data structures**
- **Example**. An infinite list of 1s can be defined as an infinitely recursive function without arguments

  function Ones = 1 : Ones

  - Operator : (cons) – *x : y* form the list with head *x*, and tail *y*
  - Ones → 1 : Ones → 1 : 1 : Ones → …
- **function Head(h : t) = h**
- Eager evaluation

  Head(Ones) → Head(1 : Ones) → Head(1 : 1 : Ones)
  → Head(1 : 1 : 1 : Ones) →Head (1 : 1 : 1 : 1 : Ones) → …
- Lazy evaluation

  Head(Ones) → Head(1 : Ones) → 1

Lambda calculus

- Theory of functions proposed by *Alonzo Church* 30es of 20 century
- Lambda calculation is transformation of **lambda expression** usin rules of lambda calculus
  - lambda expression is an identifier
  - If *x is* identifier, *e* and *n* lambda expressions then following are also lambda expressions
    - λx.e lambda abstraction
    - e n application (apply e on argument n)
- **Lambda abstraction is concept of anonymized function in FL**
  - `λx.x + 1`
  - `(λx.x + 1) 4` → `5`
  - `λx y.2x + y`
  - `(λx y.2x + y) 3 4` → `10`

Anonymized functions

- Often used as parameters of higher-order functions
- **Higher-order functions that return function as their value always return anonymized function**
- Without anonymized function

```
function inc(x) = x + 1
function twice(f, x) = f(f(x))
twice(inc, 5) → 7
```

With anonymized function

```
function twice(f, x) = f(f(x))
twice(λx.x + 1, 5) → 7
```

- **Example of function which returns function as its value:**

```
function incrementBy(x) = λy.y + x
```

Curry Functions

- **Currying: definition of function with n arguments as *n* nested functions with one argument (Haskell Curry)**

orginal function $\lambda x_1\ x_2\ \ldots\ x_n.e$

Curry function $\lambda x_1.(\lambda x_2.(\lambda x_3\ \ldots\ (\lambda x_n.e)))\ \ldots)$

- **Examples of Currying.**

```
function add(x, y) = x + y
function addCurry(x) = λy.x + y

addCurry(5)       → λy.5 + y
addCurry(5)(10) → (λy.5 + y) 10 → 15
```

Partial function application

- Let *f* is function with *k* argumenats

- **Partial application of function *f* is application of function *f* with less than *k* argumens**

- **Example.**

```
function add(x, y, z) = x + y + z
add(1, 2, 3)  → 6
add(1, 2)     → λz.3 + z
add(1)        → λy z.1 + y + z
```

- **Partial application ≡ currying, evaluation, de-Currying**

LISP (List Processing)

- First FP language has been developed in 60es, John McCarthy
- Only one type for everything – all data are s-expressions (symbolic expressions)
  - S symboli constants and numbers are i brojevi su s-expressions
  - If A and B are s-expressions then (A . B) is s-expressions - pair
  - If x_1 x_2 ... x_n s-expressions then (x_1 x_2 ... x_n) is s-expressions - lisa. () je empty list
    - List is sequence of nested pairs
      (1 2 3 4) ≡ (1 . (2 . (3 . (4 . ()))))
- **The same notation for data and functions/programs** – function definition and application are also s-expressions
  - (define (functionName arg1 arg2 ... argn) expression) ≡ definition f
  - (functionName arg1 arg2 ... argn) ≡ application f

LISP (List Processing)

- **Everything is s-expression**
  - Built-in functions for checking types of s-expressions: if s-expression is constant or number or pair or list or empty list,…
  - Quote (‘) function
    - ‘(+ 1 2) – it is s-expressions i.e. list with 3 elements
    - (+ 1 2)   – s-expressions evaluated in 3 (application of function +)

- **Conditional expression**
  - (if c e1 e2) -- if c is true then value of whole expression is the same as value of e1, if c is false then value of whole expression is the same as value of e2
  - **If expression represents value, contraty to if command**
    (+ 5 (if (> 4 5) 1 2)) → (+ 5 (if false 1 2)) → (+ 5 2) → 7

```
(define (fibonacci n)
  (if (< n 2) n
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))))
```

Successors of LISP

- ISWIM (if you see what I mean), Landin, ~1960
  - Infix notation instead of prefix notation for arithmetic-logic expressions
  - Constructions *let and where* local variables binding
  - SECD machine
- FP, Backus, ~1970
  - Functional programming as a composition of higher-order functions
- ML, Milner, ~1970
  - Parametric polymorphism, *type inference*
- SASL, KRC & Miranda, Turner
  - Lazy evaluation, ZF expressions for lists forming, Function definition as separate cases (sequence of equations) and *pattern matching*, *guard* expressions
- Haskell, 1987, international committee
  - "*Grand unification of functional languages*", *type classes*, *monads*