Functional Programming Language Scheme

Mirjana Ivanovic

University of Novi Sad, Serbia

Scheme

- Scheme - dialect of programming language LISP.
  - Small but powerful
  - With a lot of built-in functions
  - Not pure functional language but with PF subset (absence of assignment statement, order of statement execution, non-strict semantics, static binding (variables-values),…

Agenda:

- Scheme (skîm) – data types
- Built-in functions,
- “Define” ili “Let/Letrec” blok,
- Examples.

Data types and operations/functions

- Boolean;
- Symbols;
- Numbers;
- Char, Strings, Comments;
- S-expressions – pairs and lists.

S-expressions – pairs and lists

- All expressions are S-expressions, including programs/function definitions.

  -atoms – Symbols and Numbers,
  -pars (a.b), components are **a** and **b**
  -lists (S1, S2,…Sn) – (a b), components are **a** and **(b)**.

  ((brat . (ujak.sestra)) (sto stolica)) ((brat ujak.sestra) (sto stolica))
  (1 . (2)) (1 2)
  (1 . ()) (1)
  (a . (b .(c . ()))) (1 2 3)

Agenda:

- Scheme (skîm) – data types
- Built-in functions,
- “Define” ili “Let/Letrec” blok,
- Examples.

Built-in functions

- Functions for S-expression transformations.
- Numerical functions.
- Logical and relational functions.
- Conditional expression.
- Definition and application of anonymous functions.
- Let i letrec blocks – identificators' visibility.
- ….

Functions for s-expression transformations.

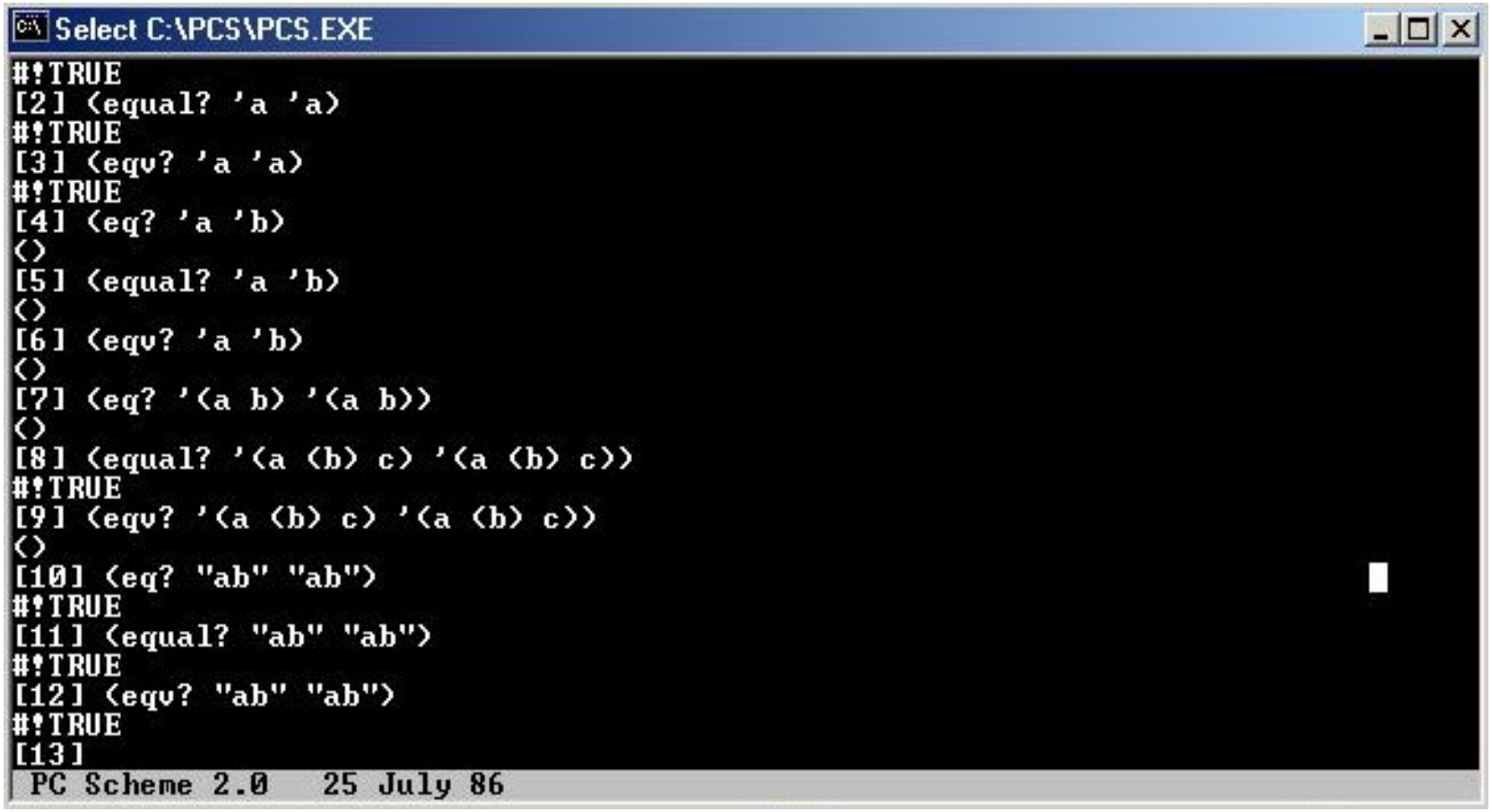
- → QUOTE - **(quote obj) => obj**
  → CONS - **(cons obj_1 obj_2) => (obj_1 obj_2)**
  → LIST – **(list obj_1 obj_2...obj_n) =>**
  (obj_1 obj_2...obj_n)
  → CAR - **(car '((1) 2 (3 (((3))))) => (1)**
  → CDR - **(cdr '((1) 2 (3 (((3))))) => (2 (3 (((3)))))**
  → C...R - **(c...r obj)**

Numerical functions, Logical and relational functions.

- * (+, -,/…) (* num_1 num_2 ...num_n)
- -1+ (sub1), 1+ (add1), quotient, reminder,…

- <?, <=?, =?, <>?, >? i >=?
- EQ?, EQV?, EQUAL? (complex S-expressions)
- AND, OR, … - (or e_1 e_2 ... e_n)

→ EQ?, EQV? ili EQUAL?



Conditional expressions

- **IF**
  Format: **(if cond expT [alternative])**

- **COND**
  Format: **(cond (i e_1 e_2 ... e_n)...))**

```
(cond       ((=? 1 1))
              ((pair? '(2 1)) 13)
              (else       (* 2 (+ 3 2)) )
            )
```

Agenda:

- Scheme (skîm) – data types
- Built-in functions,
- “Define” ili “Let/Letrec” blok,
- Examples.

Definition and application of anonymous functions.

- **LAMBDA - (lambda arg e_1 e_2 ... e_n)**

```
(lambda (x) (+ x x))
((lambda (x y) (* x y)) 2 3)
```

- **DEFINE** – naming of anonymous function

```
(define add3 (lambda (x) (+ x 3))) → naming.
(add3 4)                           →7

(define (fakt n)
        (if (<? n 2)
                1
                (* n (fakt (-1+ n))) ) )
(fakt 5)
```

Let i letrec blocks

- **LET**
  (let ((var$_1$ form$_1$) ... (var$_n$ form$_n$)) e$_1$ e$_2$ ... e$_m$)

  (let ((x 2) (y 3)) (* x y)) →6

- **LETREC – for recursive functions**
  (letrec ((var$_1$ form$_1$) ... (var$_n$ form$_n$)) e$_1$ e$_2$ ... e$_m$)

```
(letrec ((even? (lambda (n)
                  (if (zero? n) #!true (odd? (-1+ n)))))
         (odd? (lambda (n)
                  (if (zero? n) #!false (even? (-1+ n)))))  )
          (even? 88) )      →#!true
```

Agenda:

- Scheme (skîm) – data types
- Built-in functions,
- “Define” ili “Let/Letrec” blok,
- Examples.

“Define” OR “Let/Letrec” blok

***Example 1*:**

```
(define (prvi x akum)
        (if (= x 0)
                akum
                (drugi (- x 1) (+ akum x))
        )
)
 (define (drugi x akum)
        (if (= x 0)
                akum
                (prvi (- x 1) (- akum x))
        )
)

(prvi 5 0)
```

“Define” ili “Let/Letrec” blok

- ***Example 2:***

```
(define (prvidrugi x akum)
        (letrec
              ((prvi
                    (lambda (x akum)
                          (if (= x 0)
                              akum
                              (drugi (- x 1) (+ akum x))   )
                    )
               )
               (drugi
                    (lambda (x akum)
                          (if (= x 0)
                              akum
                              (prvi (- x 1) (- akum x))    )
                    )
               )
              )
              (prvi x akum)
        )
    )
(prvidrugi 5 0)
```

Miranda

$F(x)= a\ x^2 + bx + c$

```
QESolution a b c
            = error “complex solutions”, delta<0
            = [-b/(2*a)], delta=0
            = [b/(2*a)+radix/(2*a),-b/(2*a)-radix/(2*a)],delta>0
             where
                    delta = b*b-4*a*c
                     radix = sqrt delta
```

Miranda

no_dup x = x, #x<2

no_dup (a:a:x) = no_dup (a:x)

no_dup (a:b:x) = a: no_dup (b:x), a ≅ b

Miranda

- **Higher-order functions**

```
foldr op k [] = k
foldr op k (a:x) = op a (foldr op k x)
```

- **ZF (Zermelo-Frankel) expressions**

```
perm [] = [[]]
perm x = [a : y | a←x ; y←perms(x−−[a])]
```

Programing language Scala

Mirjana Ivanovic, University of Novi Sad, Serbia

1. Programing language Scala

- Scalable programming language, focus on abstraction, composition decomposition
- Martin Odesky, EPFL, Lausanne
- Inspired by ML, Haskel, Erlang and Java
- Supports OO and functional programmingf
- Allows combination with Java and .NET

1. Programing language Scala

- Purely object oriented
  - Each value is object
  - For type definitions use *classes* and properties *traits*
  - Inheritance:
    - *Subclasses*
    - Mixin-based multiple inheritance

1. Programing language Scala

- Supports functional style
  - Each function is/has value
  - Higher-order functions
  - Nested functions
  - One argument functions (*curry*)
  - Has *Case* classes for *patern matching*
  - Functions grouping - *singleton* objects

1. Programing language Scala

- Static type binding
  - Generic classes
  - Variation of anotations
  - Upper and lower type boundaries
  - Inner classes and abstract types as parts of objects
  - Structured types
  - Explic type referencing
  - Implicit parameters and conversions
  - Methods polimorfism
  - XML mode - after '<' Scala compiler switches in XML mode and parse XML text

Classes

```
class Krug {
  val Pi = 3.14
  var r: Double
}
```

Methods – are also operators

```
def uvecaj(val c: Double): Unit {
  r *= c
}
```

Classes - singleton objects

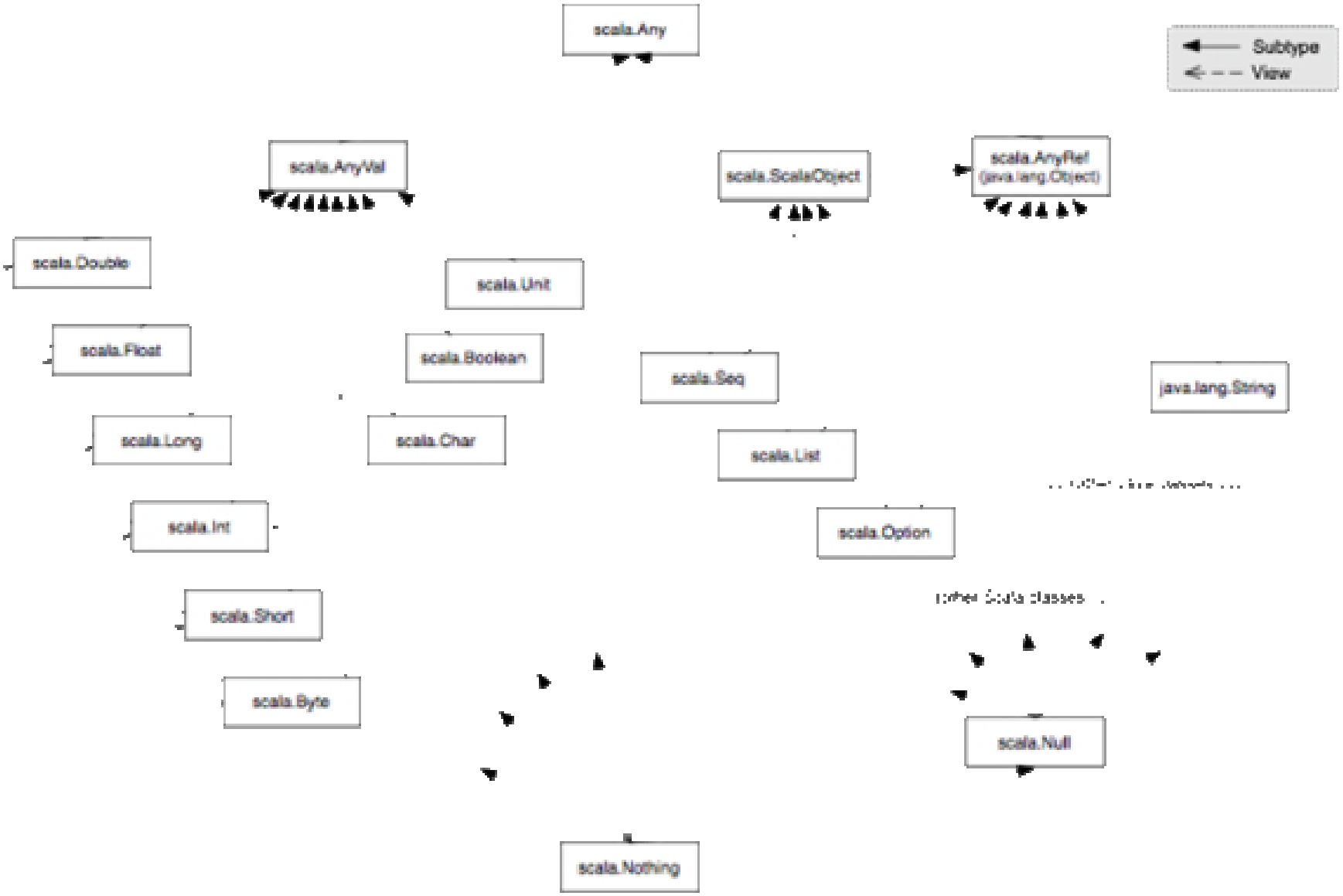
Program – creates circles

```
object Krugovi {
    def main(args: Array[String]) {
          var k = new Krug(4)
          val koef = 6
          println(k)
          pt.uvecaj(koef)
          println(k)
    }
}
```

Data Types

- All values are objects – instances of classes
- Numeric values and functions are also objects
- Arrays, Collections (tuples, lists, sets, maps-hesh tables)
- Superclass is *scala.Any* with 2 subclasses :
  - *scala.AnyVal* – superclass for value classes
  - *scala.AnyRef* – superclass for reference classes

- Superclass for user defined class is *scala.ScalaObject*


scala.Any
Subtype
View
scala.AnyVal
scala.ScalaObject
scala.AnyRef
(java.lang.Object)
scala.Double
scala.Unit
scala.Float
scala.Boolean
scala.Seq
java.lang.String
scala.Long
scala.Char
scala.List
scala.Int
scala.Option
scala.Short
scala.Byte
scala.Null
scala.Nothing

Data Types

- Example:

```
val x: Int = 42
val d: Double = 3.14
val s: String = “Hello world”
println(x.getClass.getSuperclass ==
d.getClass.getSuperclass) //true as both classes
                                are subclasses of AnyVal

println(x.getClass.getSuperclass ==
d.getClass.getSuperclass) // false as String is
                               subclass of AnyRef
```

Controls structures

- If expression
- While statement
- Do-while statement
- For expression

- Generic classes – parametrized classes, as parameters have another data types

```
class Stack[T] {
      var elems: List[T] = Nil
      def isEmpty: Boolean = elems.isEmpty
      def push(x: T) { elems = x :: elems }
      def top: T = elems.head
      def pop() { elems = elems.tail }
 }
```

Functions

- Function call – executes its apply() method
  - Function which returns Square of a number

```
val kvadrat = new Function1[Int, Int] {
  def apply(x: Int): Int = x * x
}
println(kvadrat(4))
```

- Functions could be anonymous

- High-order functions

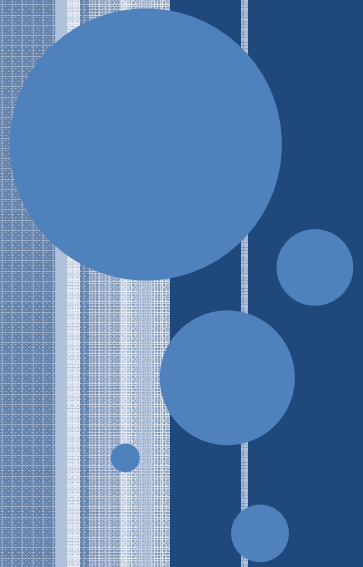
```
def saberi(f: Int => Int, a: Int, b: Int) =
  f(a)+f(b)
def sledbenik(x:Int):Int = x+1
println(sledbenik(3)) => 4
println(saberi((x:Int) => x*x, 3, 4) => ???
```

Pattern matching

- Scala built in *pattern matching* mehanizam

```
Izraz match {
    case izraz1 => vrednost1
    case izraz2 => vrednost2
    ...
    case izrazN => vrednostN
  }
```

LOGIC PROGRAMING - PROLOG



PROLOG

- 1879. G. Frege – First order predicate calculus
- 1915-1930 Computational theory and automatic theorem proving
- Principles of unification and resolution – 60thies of last century
- D. Prowitz 1960 introduced unification as method for comparing patterns in automatic theorem proving

PROLOG

- Principle of rezolution - system of formal inference, J.A. Robinson 1965

- Generalized modus ponens and formalized application of unification for pattern comparing

- R. Kowalksi explained:

$$A \leftarrow B1, B2, \ldots, Bn$$

A is valid if B1,…,Bn, are valid, i.e. it is procedure A is a procedure of a programming language which body is B1,..,Bn

- It means 'To execute A, it is necessary to execute B1,…Bn', applied mechanism is resolution

PROLOG

- 1971 A. Colmerouer and his group from Marseju implement SYSTEM8- program for mechanical theorem prooving i.e interpretation for logic programs

- This language and its interpreter - PROLOG (Programing in Logic-Programation et Logique)

- PROLOG 10 (Edinburgh prolog) and its syntax are recognized today as standard

PROLOG

- Very specific areas of application:
  - Relational databases
  - Mathematical logics
  - Abstract task solving based on inference
  - Natural language processing
  - Different areas of AI

- Descriptive language, describes objects/entities from real world and relations between them
- Program in PROLOG is set of statements/sentences that describes some facts about objects or rules that explain how solution is connected to given facts

PROLOG

- Programing in PROLOG-u has 3 steps:
  - Presenting facts
  - Defining rules about objects and relations between them
  - Queries about objects and rules among them

- Program in Prolog is set of facts about objects and relations between them, rules that have to be applied in order to obatain solutions

- Facts and rules are sentences of PROLOG

PROLOG

Facts (aksioms)

Example1

Marko reads book ←→ reads(marko, book) ←→ x(y, z)

PROLOG

Queries

Example2

Facts:

```
drive(pavle, car).
drive(marko, plain).
drive(ana, bicicle).
drive(milan, car).
drive(mara, tram).
```

- Simple query: ?- drive(milan, car). Answer: yes
- Little bit complex query: ?- drive(Who, car).
  Answers: pavle; milan
- General query: ?- drive(Who, What).
  Answers: pavle; marko,…

PROLOG

Rules(procedurea)

- Set of data bases and rule base is **knowledge base**

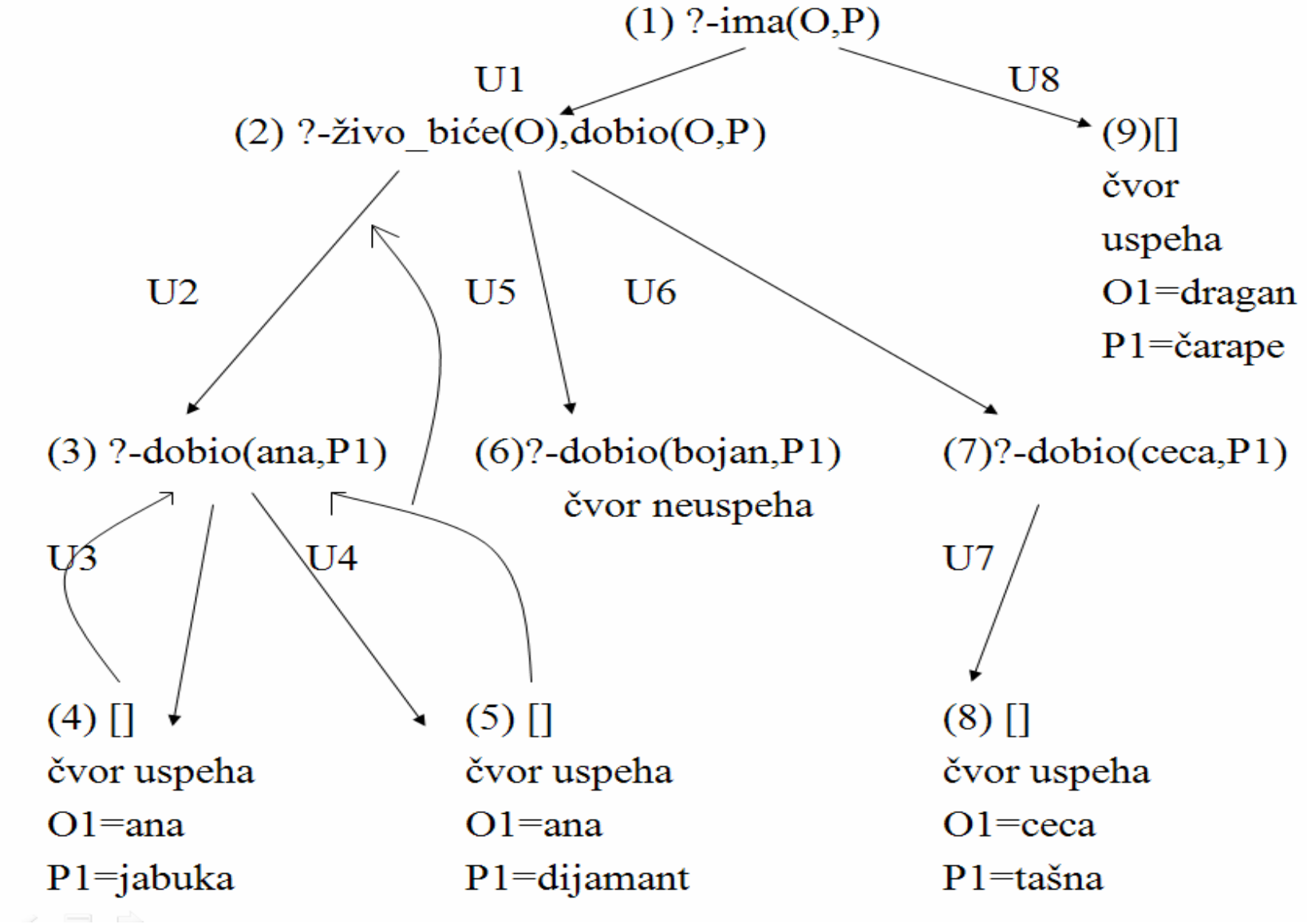
Example 3 Knowledge base of a family.

```
male(marko).
male(milan).
female(marija).
female(jelena).
parent(marija, marko, milan).
parent(marija, marko, jelena).
 is_sister(X,Y) :- female(X),          // Rule
        parent(M, F, X),
         parent(M, F, Y),
        not(X=Y).

?-  is_sister (X, milan).
  X=jelena;
  no
```

PROLOG – finding solutions

Search tree



PROLOG

- Possible to define standard operations

- Lists can be defined recursively
  - Empty list
  - Non-empty list has Head ant Tail

  - Empty list [ ]

- Non-empty list [X | Y], X is Head and Y is Tail

- Finite lists [a, b, c, d] => Head is a, Tail is [b, c, d]

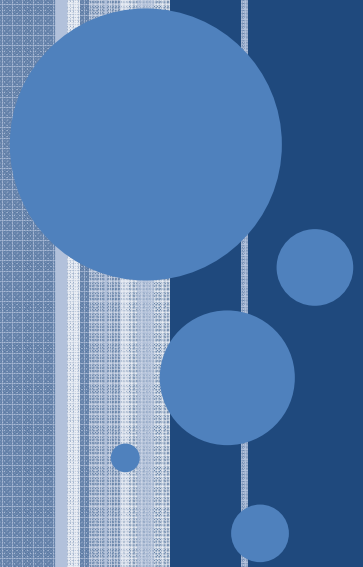
- Example – deleting element from a list:
  takeout(X, [X | R], R).
  takeout(X, [F | R], [F | S]) :- takeout(X, R, S).

PROLOG –LISTS

- Intersection of 2 lists :

```
intersection([ ], M, [ ]).
intersection([X | Y], M, [X | Z]) :- member(X, M), intersection(Y, M, Z).
intersection([X | Y], M, Z) :- \= member(X, M), intersection(Y, M, Z).
```

EXAMPLES – PROLOG, SCHEME, SCALA



FIBONACCI - PROLOG

```
fibon(1, 1).
fibon(2, 1).

fibon(N, X) :- N > 2,
        N1 is N - 1,
        N2 is N - 2,
        fibon(N1, X1),
        fibon(N2, X2),
            X is X1 + X2.
```

FIBONACCI - SCHEME

```
(define (fibonaci1 n)
    (if (<= n 1)
        1
        (+ (fibonaci1 (- n 1)) (fibonaci1 (- n 2)))
    )
)
```

FIBONACCI - SCALA

```
object Fibb{

    def fib(n:Int):Int = {
        def fibHelper(n:Int, a:Int, b:Int):Int = {
            if(n == 0) a
            else fibHelper(n - 1, b, a + b)
        }
    return fibHelper(n, 0, 1)
    }

    def main(args: Array[String]) {
        print(fib(10));
    }
}
```