Procedural control of reasoning

Automated proving methods answer a question by trying all logically permissible options in the knowledge base.

These reasoning methods are domain-independent. But in some situations it is not feasible to search all logically possible ways to find a solution.

We often have an idea about how to use knowledge and we can “guide” an automated procedure based on properties of the domain.

We will see how knowledge can be expressed to control the backward-chaining reasoning procedure.

These slides are mainly based on Ronald J. Brachman, Hector J. Levesque. Knowledge representation and reasoning, Morgan Kaufmann, 2004.

Facts and rules

The clauses in a KB can be divided in two categories:

- Facts – are ground terms (without variables)
- Rules – are conditionals that express new relations – they are universally quantified.

Mather(jane,john)

Father(john,bill)

…

Parent(x,y) $\Leftarrow$ Mother(x,y)

Parent(x,y) $\Leftarrow$ Father(x,y)

Rules involve chaining and the control issue regards the use of the rules to make it most effective.

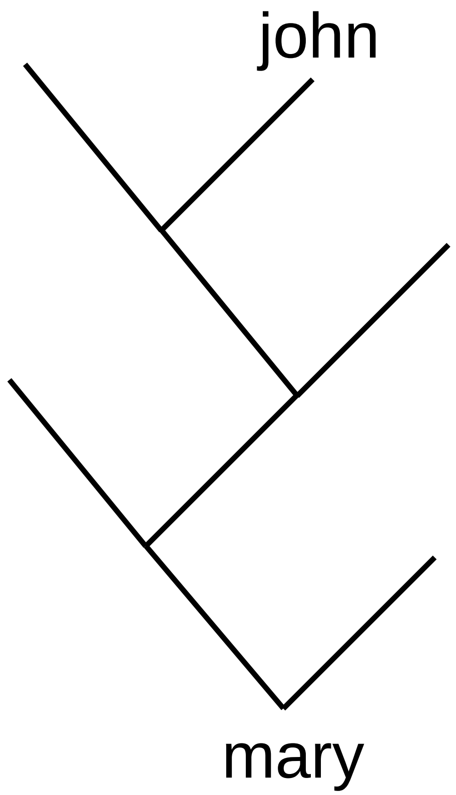
Rule formation and search strategies

We can express the Pred relation in two logically equivalent ways:

1. Pred(x,y) ⇐ Parent(x,y)
   Pred(x,y) ⇐ Parent(x,z) ∧ Pred(z,y)

2. Pred(x,y) ⇐ Parent(x,y)
   Pred(x,y) ⇐ Parent(z,y) ∧ Pred(x,z)

Rule formation and search strategies



1. We search top-down in the family tree
2. We search down-top

If people had on average one child, then 1) would be of order d and 2) of order 2^d, where d is the depth of search. If people had more than 2 children, 2) would be a better option.

Algorithm design

The Fibonacci series
$$\begin{cases} x_0=0 \\ x_1=1 \\ x_{n+2}=x_{n+1}+x_n,\ n\geq 0 \end{cases}$$

Fib(0,1)

Fib(1,1)

Fib(s(s(n)),v) ⇐ Fib(n,y) ∧ Fib(s(n),z) ∧ Plus(y,z,v)

Plus(0,z,z)

Plus(s(x),y,s(z)) ⇐ Plus(x,y,z)

Note: 0 is shortcut for zero; 1 for s(zero); 2 for s(s(zero)) and so on.

Algorithm design

The Fibonacci series $\begin{cases} x_0=0 \\ x_1=1 \\ x_{n+2}=x_{n+1}+x_n,\ n\geq 0 \end{cases}$

Fib(0,1)

Fib(1,1)

Fib(s(s(n)),v) $\Leftarrow$ Fib(n,y) ∧ Fib(s(n),z) ∧ Plus(y,z,v)

Plus(0,z,z)

Plus(s(x),y,s(z)) $\Leftarrow$ Plus(x,y,z)

Note: 0 is shortcut for zero; 1 for s(zero); 2 for s(s(zero)) and so on.

Most of the computation is redundant

Fib(10,_) calls Fib(9,_) and Fib(8,_)

Fib(11,_) calls Fib(10,_) and Fib(9,_)

Each application of Fib calls Fib twice and it generates an exponential number of Plus subgoals.

Algorithm design

An alternative is

$$Fib(n,v) \Leftarrow F(n,1,0,v)$$

$$F(0,y,z,z)$$

$$F(s(n),y,z,v) \Leftarrow Plus(y,z,s) \wedge F(n,s,y,v)$$

F(n,_,_,v)

the solution is obtained when n is 0

two consecutive Fib numbers

starts from n towards 0

Goal order

From logical point of view, all ordering of subgoals are equivalent, but the computational differences can be significant.

For example

$$AmericanCousin(x,y) \Leftarrow American(x) \wedge Cousin(x,y)$$

We have two options:

Find an American and see if he is a cousin.

Find a cousin and see if he is American.

In this case, solving first Cousin(x,y) and then American(x) is better than the other way around.

Backtracking control and negation as failure

- predicate ! (“cut”) in PROLOG[1]

! is always true; it prevents backtracking in the place it occurs in the program.

If ! doesn’t change the declarative meaning of the program, then it is called green; otherwise it is red.

The function

$$f(x) = \begin{cases} 0, & x \le 3 \\ 2, & x \in (3,6] \\ 4, & x > 6 \end{cases}$$

can be implemented as:

```
f(X,0):-X=<3.
f(X,2):-3<X,X=<6.
f(X,4):-6<X.
```

1. Ivan Bratko. Prolog Programming for Artificial Intelligence, Pearson Education Canada, 4th Edition, 2011.

Backtracking control and negation as failure

- predicate ! (“cut”) in PROLOG

! is always true; it prevents backtracking in the place it occurs in the program.

If ! doesn’t change the declarative meaning of the program, then it is called green; otherwise it is red.

The function

$$f(x) = \begin{cases} 0, & x \le 3 \\ 2, & x \in (3,6] \\ 4, & x > 6 \end{cases}$$

can be implemented as:

```
f(X,0):-X=<3.
f(X,2):-3<X,X=<6.
f(X,4):-6<X.
```

```
?-f(1,Y),2<Y.
    X=1,Y=0, 1=<3,2<0 false
    X=1,Y=2 false
    X=1, Y=4 false
```

The program should have stopped after the first check.

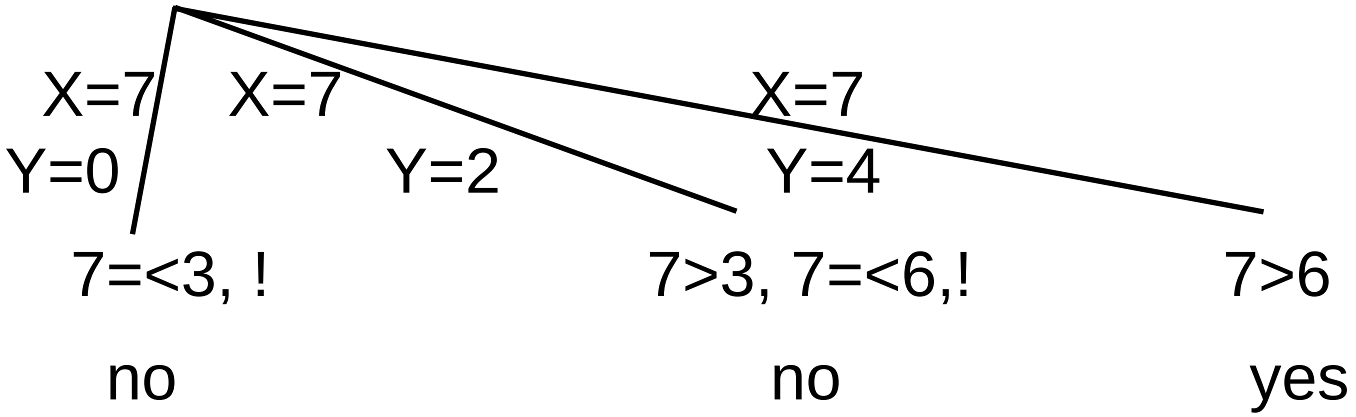
Backtracking control and negation as failure

f(X,0):-X=<3,!.

f(X,2):-3<X,X=<6,!. green !

f(X,4):-6<X.

?-f(7,Y).



redundant tests

f(X,0):-X=<3,!.

f(X,2):-X=<6,!. red ! - if we remove ! and ask ?-f(1,Y).

f(X,4).

Y=0;

Y=2;

Y=4;

false

Backtracking control and negation as failure

The parent of a “cut” is that PROLOG goal that matches the head of the rule that contains that “cut”.

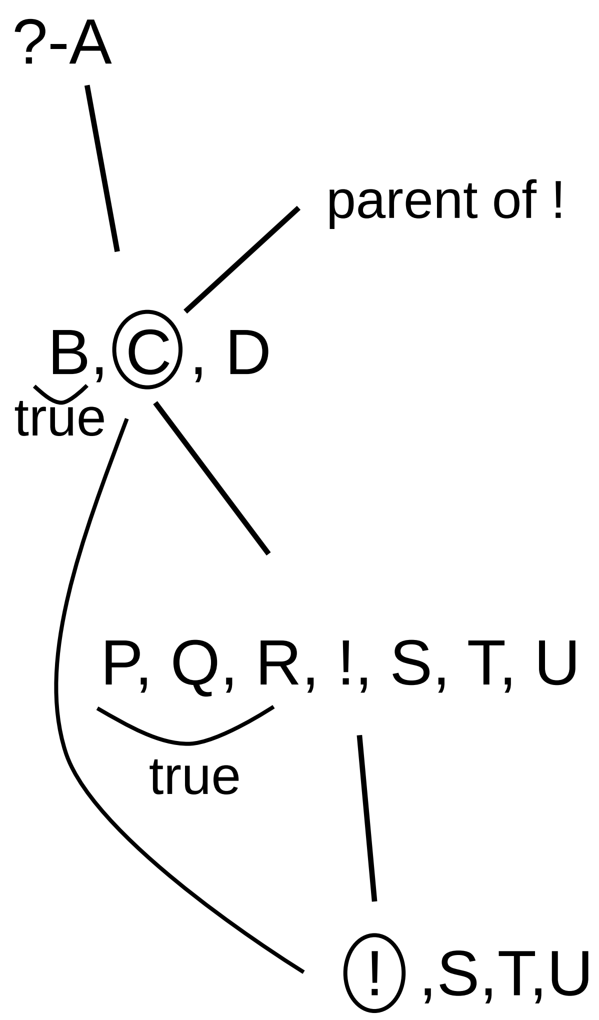
```
C:-P,Q,R,!,S,T,U.
C:-V.
A:-B,C,D.
```

?-A.

Backtracking is possible for P,Q,R, but as soon as ! is executed, all of the alternative solutions are suppressed.

Also, the alternative C:-V will be suppressed.

Backtracking control and negation as failure



in the goal tree, backtracking is prevented

between ! and its parent

! affects only the execution of C

Backtracking control and negation as failure

```
max(X,Y,X):-X>=Y,!.
max(_,Y,Y).

member(X,[X|L]):-!.
member(X,[_|L]):-member(X,L).
```

Given the following KB:

```
p(1).
p(2):-!.
p(3).
```

What are PROLOG answers to the following questions?

```
?-p(X).
?-p(X),p(Y).
?-p(X),!,p(Y).
```

Backtracking control and negation as failure

Negation as failure

Predicate “fail” is always false.

John likes all animals, with the exception of snakes

```
likes(john,X):-snake(X),!,fail.
likes(john,X):-animal(X).
```

We define the unary predicate “not” as following: not(G) fails if G succeeds; otherwise not(G) succeeds.

```
not(G):-G,!,fail.
not(G).
```

Now we can write

```
likes(john,X):-animal(X),not(snake(X)).
```

Backtracking control and negation as failure

Procedurally, we distinguish between two types of negative situations with respect to a goal G:

- being able to solve ¬G
- being unable to solve G – this happens when we run out of options when trying to prove that G is true.

“Not” in PROLOG doesn’t correspond exactly to the mathematical negation. When PROLOG processes a “not” goal, it doesn’t try to solve it directly, but to solve the opposite.

If the opposite cannot be demonstrated, then PROLOG assumes that the “not” goal is solved.

Such a reasoning is based on the Closed-World Assumption. That is to say that if something is not in the KB or it cannot be derived from the KB, then it is not true and consequently, its negation is true.

Backtracking control and negation as failure

For example, if we ask:

?-not(human(mary)).

The answer is “yes” if human(mary) is not in KB. But it should not be understood as “Mary is not a human being”, but rather “there is not information in the program to prove that Mary is a human being”

Usually, we do not assume the “Close-World” – if we do not explicitly say “human(mary)”, we do not implicitly understand that Mary is not a human being.

Backtracking control and negation as failure

Other examples:

1. composite(N):-N>1,not(primeNumber(N)).

   The failure to prove that a number greater than 1 is prime is sufficient to conclude that the number is composite.

2. good(renault).

   good(audi).

   expensive(audi).

   reasonable(Car):-not(expensive(Car)).

```
?-good(X),reasonable(X).
?-reasonable(X),good(X).
```

! is useful and, in many situations, necessary, but it must be used with special attention.

Grammars in Prolog

A grammar is a formal specification of the rules that define the accepted structures of a language.

Def. A grammar is a tuple G=(N,T,S,P), where N is the alphabet of the non-terminal symbols (denoted by capital letters), T is the alphabet of the terminal symbols (denoted by lower case letters), $N \cap T = \emptyset$,

$S \in N$ is the start symbol,

P is the set of production rules $P \subseteq V_G^* N V_G^* \times V_G^*$, $V_G = N \cup T$.

Notation: $\alpha \rightarrow \beta$ instead of $(\alpha,\beta) \in P$.

Def. A context free grammar is a grammar where the production rules have the form:

$$A \rightarrow x \text{ , } x \in V_G^*, A \in N.$$

Grammars in Prolog

Def. The language generated by a grammar G is

$$L(G)=\{ w \mid w \in T^*, S \overset{*}{\Rightarrow} w\}$$

Example:

$S \rightarrow SS$

$S \rightarrow aSb$

$S \rightarrow bSA$

$S \rightarrow ab$

$S \rightarrow ba$

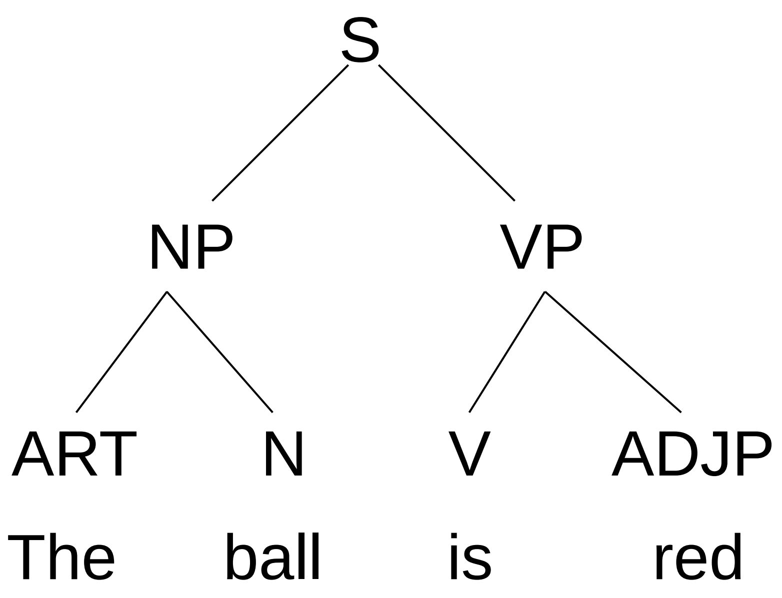
A derivation:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow aSbSS \Rightarrow a^2b^2SS \Rightarrow a^2b^2abS \Rightarrow a^2b^2abab$$

Grammars in Prolog

A sentence S consists of many syntactic groups like:

- -NP (noun phrase);
- -VP (verb phrase);
- -ADJP (adjectival phrase);
- -ADVP (adverb phrase) etc.



Grammars in Prolog

A rule like S → NP VP states that the sentence contains a noun phrase and a verbal phrase in this order.

Such rules are called PS (phrase structure) rules and a grammar that is defined by PS rules is called a PS grammar (is a context free grammar).

For natural languages, the terminal symbols are the words in that language and the non-terminal symbols are S, NP, VP etc.

The Definite Clause Grammar (DCG) notation

A PS rule like S → NP VP can be written in Prolog as:

s(L1,L) :- np(L1,L2), vp(L2,L).

where

L1 is the initial input sequence as a list (e.g., [The, student, loves, a, book]);

L2=[loves, a, book] is the initial input sequence without the noun phrase [the, student];

L=[ ] is L2 without the verbal phrase [loves, a, book] – is the remaining sequence after parsing the initial sequence with the rule S → NP VP.

The rules that 'treat' the terminal symbols (i.e., words of the natural language) have the form:

n([student | L],L).

The DCG notation

In the DCG notation, Prolog the rule

```
s(L1,L) :- np(L1,L2), vp(L2,L).
```

is written as:

```
s --> np,vp.
```

and the Prolog fact

```
n([student | L],L).
```

is written as

```
n --> [student].
```

The DCG notation

More general, the DCG rules are translated in Prolog as following:

```
n(Z)-->n1,n2,…,nm.
```

with n1,n2,…,nm non-terminal symbols

is

```
n(Z,X,Y):-n1(X,Y1),n2(Y1,Y2),…,nm(Ym_1,Y).
```

```
n(Z)-->n1(W),[t2],n3,[t4].
```

with n1,n3 non-terminal symbols
and t2, t4 terminal symbols

is

```
n(Z,X,Y):-n1(W,X,[t2 | Y1]),n3(Y1,[t4 | Y]).
```

The DCG notation

Example:

```
s --> [a],[b].
s --> [a],s,[b].



?-s([a,a,b,b],[]).


?-s([a,a,b,b,c],[c]).


?-s([a,a,b,b,c],[a]).
```

The DCG notation

Example:

```
s --> [a],[b].          equivalent to s([a,b | X] , X).
s --> [a],s,[b].                      s([a | X ] , Y) :- s(X , [b | Y]).
```

```
?-s([a,a,b,b],[]).

?-s([a,a,b,b,c],[c]).

?-s([a,a,b,b,c],[a]).
```

For more on Grammars in Prolog please see:

1. Ivan Bratko. Prolog Programming for Artificial Intelligence, Pearson Education Canada, 4th Edition, 2011 – chapter 'Language Processing with Grammar Rules'
2. Florentina Hristea, Maria Florina Balcan. Căutarea şi reprezentarea cunoştinţelor în Inteligenţa artificială. Teorie şi aplicaţii, Editura Universităţii din Bucureşti, 2005 – chapter 7.2.4

Java-SWI Prolog interface - created by Irina Ciocan

ConexiuneProlog.java | panou_intrebare.java | ExempluInterfataProlog.java | CititorMesaje.java | Fereastra.java | ExpeditorMesaje.java

```java
import java.io.IOException;

/**
 *
 * @author Irina
 */
public class ConexiuneProlog {
    final String caleExecutabilSicstus="C:\\Program Files\\swipl\\bin\\swipl-win.exe";

    final String nume_fisier="exemplu_prolog.pl";

    final String scop="inceput.";


    Process procesSicstus;
    ExpeditorMesaje expeditor;
    CititorMesaje cititor;
    Fereastra fereastra;
    int port;


    public Fereastra getFereastra(){
        return fereastra;
    }
```

check the path to the SWI executable

the Prolog KB

the Prolog predicate that connects the Prolog KB to Java

Build the Java project ExempluInterfataPrologSwi and run

...\ExempluInterfataPrologSwi\src\ExempluInterfataProlog.java

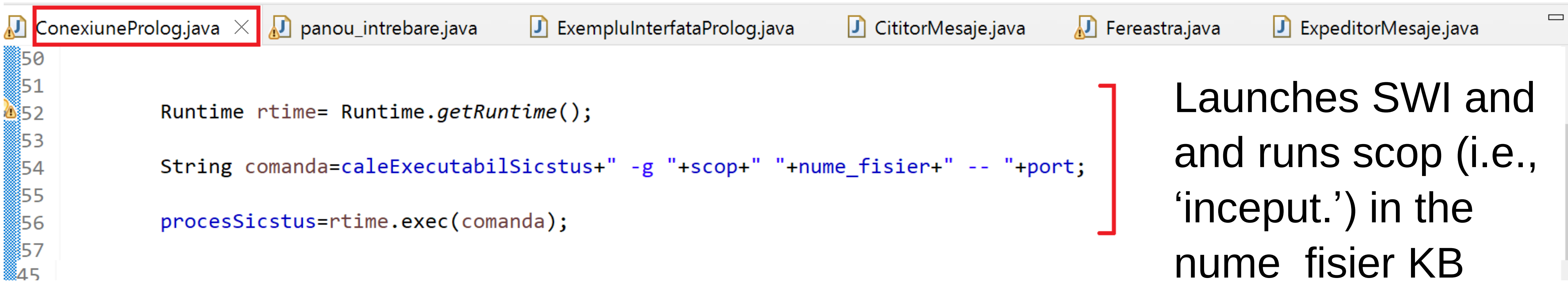
Java-SWI Prolog interface

ConexiuneProlog.java | panou_intrebare.java | ExempluInterfataProlog.java | CititorMesaje.java | Fereastra.java | ExpeditorMesaje.java

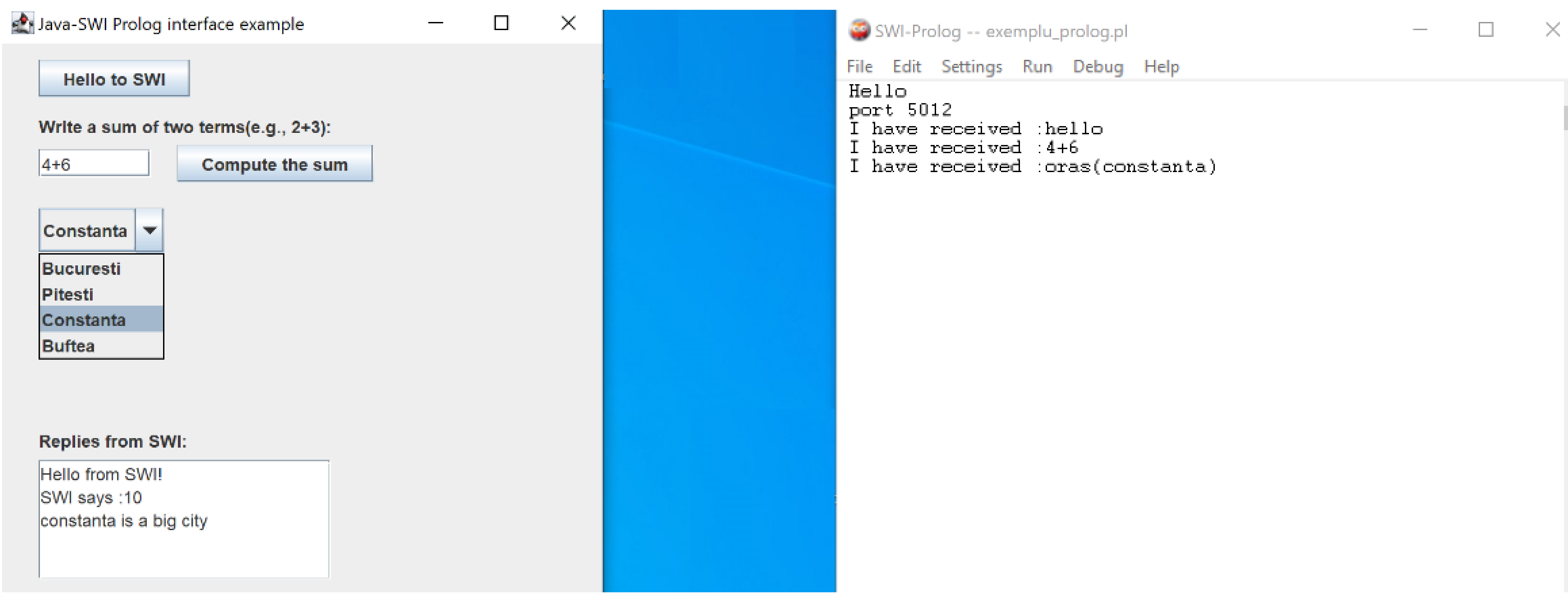
```java
Runtime rtime= Runtime.getRuntime();

String comanda=caleExecutabilSicstus+" -g "+scop+" "+nume_fisier+" -- "+port;

procesSicstus=rtime.exec(comanda);
```

Launches SWI and and runs scop (i.e., ‘inceput.’) in the nume_fisier KB

Java-SWI Prolog interface



Launches SWI and and runs scop (i.e., ‘inceput.’) in the nume_fisier KB



Java-SWI Prolog interface

ConexiuneProlog.java | panou_intrebare.java | ExempluInterfataProlog.java | CititorMesaje.java | Fereastra.java | ExpeditorMesaje.java

```java
    }//GEN-LAST:event_tfParametruActionPerformed

    private void okButtonActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_okButtonActionPerformed
        //okButton.setEnabled(false);
        String valoareParametru=tfParametru.getText();
        tfParametru.setText("");
        try {
            conexiune.expeditor.trimiteMesajSicstus(valoareParametru);
        } catch (Exception ex) {
            Logger.getLogger(Fereastra.class.getName()).log(Level.SEVERE, null, ex);
        }
    }//GEN-LAST:event_okButtonActionPerformed

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jButton1ActionPerformed
        try {
            conexiune.expeditor.trimiteMesajSicstus("hello");
        } catch (Exception ex) {
            Logger.getLogger(Fereastra.class.getName()).log(Level.SEVERE, null, ex);
        }
    }//GEN-LAST:event_jButton1ActionPerformed

    private void jComboBox2ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jComboBox2ActionPerformed
        String oras=(String)jComboBox2.getSelectedItem();
        System.out.println(oras);
        try {
            conexiune.expeditor.trimiteMesajSicstus("oras("+oras.toLowerCase()+")");
        } catch (Exception ex) {
            Logger.getLogger(Fereastra.class.getName()).log(Level.SEVERE, null, ex);
        }
    }//GEN-LAST:event_jComboBox2ActionPerformed

    /**
```

Java-SWI Prolog interface

exemplu_prolog - Notepad

File Edit Format View Help

```
proceseaza_termen_citit(IStream, OStream, X + Y,C):-
                                Rez is X+Y,
                                write(OStream,'SWI says ' : Rez),nl(OStream),
                                flush_output(OStream),
                                C1 is C+1,
                                proceseaza_text_primit(IStream, OStream,C1).

proceseaza_termen_citit(IStream, OStream,hello,C):-
                                write(OStream,'Hello from SWI!\n'),
                                flush_output(OStream),
                                C1 is C+1,
                                proceseaza_text_primit(IStream, OStream,C1).

proceseaza_termen_citit(IStream, OStream, oras(X),C):-
                                oras(X,Tip),
                                format(OStream,'~p is a ~p city\n',[X,Tip]),
                                flush_output(OStream),
                                C1 is C+1,
                                proceseaza_text_primit(IStream, OStream,C1).
```

Come from Java, (see the previous slide)

Sent back to Java