Semantics: Recursion

TK2ICM: Logic Programming (2nd Term 2018-2019)

M. Arzaki

Computing Laboratory, School of Computing
Telkom University

SoC Tel-U

February 2019

Acknowledgements

This slide is compiled using the materials in the following sources:
Books:

1. P. Blackburn, J. Bos, K. Striegnitz, *Learn Prolog Now!* (Chapter 1-6, 10,11), London: College Publications (available online), 2006. **[LPN]**
2. M. Bramer, *Logic Programming with Prolog* (Chapter 1-9), 2nd Edition, Springer, 2013. **[LPwP]**
3. I. Bratko, *Prolog Programming for Artificial Intelligence* (Chapter 1-3, 5,6,8,9), Pearson Education, 2001. (advanced reference). **[PPAI]**
4. K. H. Rosen, *Discrete Mathematics and Its Applications* (Chapter1), 7th Edition, 2012.
5. M. Ben-Ari, *Mathematical Logic for Computer Science* (Logic Programming Sections), 2nd Edition, 2000.

Lecture slides and lecture notes:

1. *Prolog Programming* by Kristina Striegnitz.
2. *Learn Prolog Now!* by Patrick Blackburn, Johan Bos, and Kristina Striegnitz.
3. *Logic Programming* at Fasilkom UI by A. A. Krisnadhi and A. Saptawijaya.
4. *Computational Logic Part 2: Logic Programming* at Fasilkom UI by L. Y. Stefanus.
5. *Logic Programming* at ILLC, University of Amsterdam by U. Endriss.
6. *Functional Programming* at Fasilkom UI by A. Azurat.
7. *Bahasa Prolog* at FPMIPA UPI by Munir.
8. Other available sources online.

Some of the pictures are taken from the above resources. This slide is intended for academic purpose at FIF Telkom University. You are prohibited for using these slides for professional purpose outside Telkom University, unless with the author authorization. If you have any suggestions/comments/questions related with the material on this slide, send an email to `<pleasedontspam>@telkomuniversity.ac.id`.

Contents

Contents

Recursion

- Prolog predicates can be defined recursively.
- A predicate is recursively defined if one or more rules in its definition refers to itself.
- There are two forms of recursion:
  1. Direct recursion. Predicate `pred1` is defined in terms of itself.

Recursion

- Prolog predicates can be defined recursively.
- A predicate is recursively defined if one or more rules in its definition refers to itself.
- There are two forms of recursion:
  1. Direct recursion. Predicate `pred1` is defined in terms of itself.

```
pred1:- pred1, pred2.
```

  2. Indirect recursion. Predicate `pred1` is defined using `pred2`, which is defined using `pred3`, ..., which is defined using `pred1`.

Recursion

- Prolog predicates can be defined recursively.
- A predicate is recursively defined if one or more rules in its definition refers to itself.
- There are two forms of recursion:
  1. Direct recursion. Predicate `pred1` is defined in terms of itself.

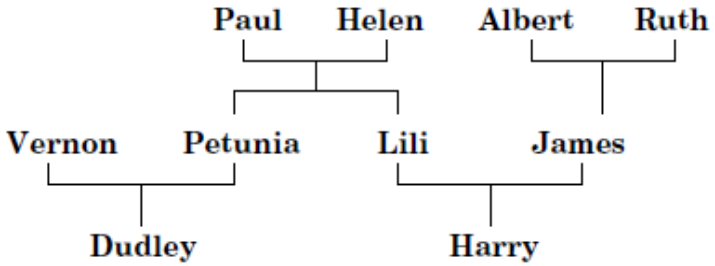
```
pred1:- pred1, pred2.
```

  2. Indirect recursion. Predicate `pred1` is defined using `pred2`, which is defined using `pred3`, ..., which is defined using `pred1`.

```
pred1:- pred2.
pred2:- pred3.
pred3:- pred1.
```

Contents

Family Tree



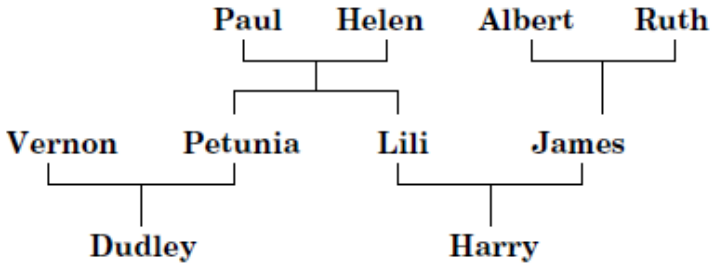

```
parent_of(paul,petunia).
parent_of(helen,petunia).
parent_of(paul,lili).
parent_of(helen,lili).
parent_of(albert,james).
parent_of(ruth,james).
parent_of(petunia,dudley).
parent_of(vernon,dudley).
parent_of(lili,harry).
parent_of(james,harry).
```

Problem

Define a predicate `ancestor_of(X,Y)` which is true if `X` is an ancestor of `Y`.

Example

Family Tree




```
parent_of(paul,petunia).
parent_of(helen,petunia).
parent_of(paul,lili).
parent_of(helen,lili).
parent_of(albert,james).
parent_of(ruth,james).
parent_of(petunia,dudley).
parent_of(vernon,dudley).
parent_of(lili,harry).
parent_of(james,harry).
```

Problem

Define a predicate `ancestor_of(X,Y)` which is true if `X` is an ancestor of `Y`.

Example

We shall have: `ancestor_of(paul,petunia)`, `ancestor_of(helen,petunia)`, `ancestor_of(paul,dudley)`, `ancestor_of(paul,harry)`, etc..

Defining Ancestors

From previous lecture, we see that:

Defining Ancestors

From previous lecture, we see that:

- `grandparent_of(X,Y):- parent_of(X,Z), parent_of(Z,Y).`

Defining Ancestors

From previous lecture, we see that:

- `grandparent_of(X,Y):- parent_of(X,Z), parent_of(Z,Y).`
- `greatgrandparent_of(X,Y):- parent_of(X,W), parent_of(W,Z), parent_of(Z,Y).`

Defining Ancestors

From previous lecture, we see that:

- `grandparent_of(X,Y):- parent_of(X,Z), parent_of(Z,Y).`
- `greatgrandparent_of(X,Y):- parent_of(X,W), parent_of(W,Z), parent_of(Z,Y).`
- `greatgreatgrandparent_of(X,Y):- parent_of(X,A), parent_of(A,B), parent_of(B,C), parent_of(C,Y).`
- The "normal way" to define predicate does not work, since we don't know exactly "how many parents we have to go back".

Fortunately, Prolog allows us to define a predicate recursively. Notice that:

1. `X` is an ancestor of `Y` if `X` is parent of `Y`, i.e.:

Defining Ancestors

From previous lecture, we see that:

- `grandparent_of(X,Y):- parent_of(X,Z), parent_of(Z,Y).`
- `greatgrandparent_of(X,Y):- parent_of(X,W), parent_of(W,Z), parent_of(Z,Y).`
- `greatgreatgrandparent_of(X,Y):- parent_of(X,A), parent_of(A,B), parent_of(B,C), parent_of(C,Y).`
- The "normal way" to define predicate does not work, since we don't know exactly "how many parents we have to go back".

Fortunately, Prolog allows us to define a predicate recursively. Notice that:

1. `X` is an ancestor of `Y` if `X` is parent of `Y`, i.e.:

```
ancestor_of(X,Y):- parent_of(X,Y).
```

2. `X` is an ancestor of `Y` if `X` is a parent of someone who is an ancestor of `Y`.

Defining Ancestors

From previous lecture, we see that:

- `grandparent_of(X,Y):- parent_of(X,Z), parent_of(Z,Y).`
- `greatgrandparent_of(X,Y):- parent_of(X,W), parent_of(W,Z), parent_of(Z,Y).`
- `greatgreatgrandparent_of(X,Y):- parent_of(X,A), parent_of(A,B), parent_of(B,C), parent_of(C,Y).`
- The "normal way" to define predicate does not work, since we don't know exactly "how many parents we have to go back".

Fortunately, Prolog allows us to define a predicate recursively. Notice that:

1. `X` is an ancestor of `Y` if `X` is parent of `Y`, i.e.:

```
ancestor_of(X,Y):- parent_of(X,Y).
```

2. `X` is an ancestor of `Y` if `X` is a parent of someone who is an ancestor of `Y`.

```
ancestor_of(X,Y):- parent_of(X,Z), ancestor_of(Z,Y).
```

Notice that the above recursive predicate is declaratively equivalent to `ancestor_of(X,Y):- ancestor_of(Z,Y), parent_of(X,Z).`

Contents

Defining Descendant

Suppose we have a knowledge base recording facts about the child relation:

```
child(charlotte,caroline).
child(caroline,laura).
/* child(X,Y) means X has a child whose name is Y */
```

Problem

How do we define the predicate `descend(X,Y)` that is true whenever `Y` is a descendant of `X`?

Our first attempt:

Defining Descendant

Suppose we have a knowledge base recording facts about the child relation:

```
child(charlotte,caroline).
child(caroline,laura).
/* child(X,Y) means X has a child whose name is Y */
```

Problem

How do we define the predicate `descend(X,Y)` that is true whenever `Y` is a descendant of `X`?

Our first attempt:

```
descend(X,Y):- child(X,Y).
```

Defining Descendant

Suppose we have a knowledge base recording facts about the child relation:

```
child(charlotte,caroline).
child(caroline,laura).
/* child(X,Y) means X has a child whose name is Y */
```

Problem

How do we define the predicate `descend(X,Y)` that is true whenever `Y` is a descendant of `X`?

Our first attempt:

```
descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),child(Z,Y).
```

Now suppose we extend our knowledge base as follows:

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).
```

Problem

How do we define the predicate `descend(X,Y)` that is true whenever `Y` is a descendant of `X`?

Possible attempt:

Now suppose we extend our knowledge base as follows:

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).
```

Problem

How do we define the predicate `descend(X,Y)` that is true whenever `Y` is a descendant of `X`?

Possible attempt:

```
descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
descend(X,Y):- child(X,W), child(W,Z), child(Z,Y).
```

Now suppose we extend our knowledge base as follows:

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).
```

Problem

How do we define the predicate `descend(X,Y)` that is true whenever `Y` is a descendant of `X`?

Possible attempt:

```
descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
descend(X,Y):- child(X,W), child(W,Z), child(Z,Y).
```

Again, The "normal way" to define predicate does not work, since we don't know exactly "how many child to go onward".

We can define the predicate `descend(X,Y)` as follows:

```
descend(X,Y):- child(X,Y).
/* Y is a descendant of X if Y is a child of X */
descend(X,Y):- child(X,Z), descend(Z,Y).
/* Y is a descendant of X if Y is a child of someone who is a
descendant of X */
```

An Example of Search Tree from Recursion

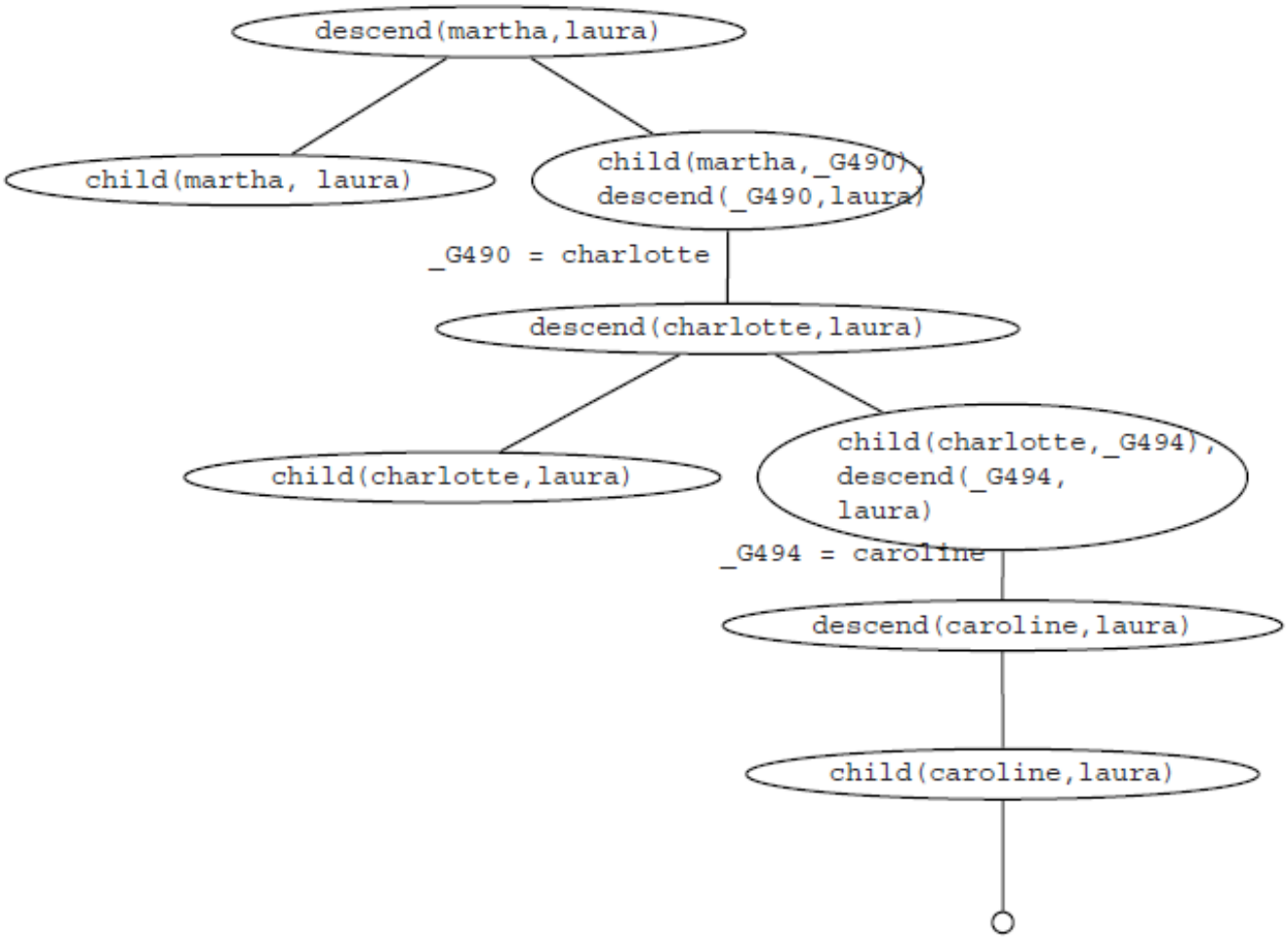
Suppose we have the following knowledge base.

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).
```

- Using the aforementioned predicate `descend(X,Y)`, we know that `descend(martha,laura)` is true.
- We are interested to study the search tree of this recursive invocation.

Recall that we use the predicate:

```
descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), descend(Z,Y)
```


descend(martha,laura)
child(martha, laura)
child(martha,_G490),
descend(_G490,laura)
_G490 = charlotte
descend(charlotte,laura)
child(charlotte,laura)
child(charlotte,_G494),
descend(_G494,
laura)
_G494 = caroline
descend(caroline,laura)
child(caroline,laura)

Contents

Flight Path

Suppose we have the following knowledge base that express the possible direct flights between cities:

```
direct(jakarta,yogyakarta).
direct(jakarta,surabaya).
direct(jakarta,bali).
direct(bandung,yogyakarta).
direct(bandung,surabaya).
direct(bandung,bali).
direct(surabaya,makassar).
direct(makassar,manado).
```

Problem

Suppose we want to define that there is a flight from city `A` to city `B` if there is a series of direct flights connecting city `A` to city `B`. Express this possibility using the predicate `flight(X,Y)`.

Defining Flight Path

```
flight(X,Y):- direct(X,Y).
/* there is a flight from X to Y if there is a direct flight from
X to Y */
flight(X,Y):- direct(X,Z), flight(Z,Y).
/* there is a flight from X to Y if there is a direct flight from
X to some city that has a flight path to Y */
```

Contents

A (Very) Brief History of Number

- Nowadays, when we write numerals, we usually use decimal notation (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and so on) but as you probably know, there are many other notations.
- Computers usually use binary notation to represent numerals (0, 1, 10, 11, 100, 101, 110, 111, 1000, and so on).
- Other cultures use different systems.
- For example, the ancient Babylonians used a base 60 system, while the ancient Romans used a rather ad-hoc system (I, II, III, IV, V, VI, VII, VIII, IX, X).

Problem

A (Very) Brief History of Number

- Nowadays, when we write numerals, we usually use decimal notation (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and so on) but as you probably know, there are many other notations.
- Computers usually use binary notation to represent numerals (0, 1, 10, 11, 100, 101, 110, 111, 1000, and so on).
- Other cultures use different systems.
- For example, the ancient Babylonians used a base 60 system, while the ancient Romans used a rather ad-hoc system (I, II, III, IV, V, VI, VII, VIII, IX, X).

Problem

How did human define number in the first place? Assuming that there is an intelligent life outside our planet, how do we communicate numbers with "them"?

Peano Axioms

In mathematical logic, the Peano axioms, also known as the Dedekind–Peano axioms or the Peano postulates, are a set of axioms for the natural numbers presented by the 19th century Italian mathematician Giuseppe Peano.

Peano Axiom

1. `0` is a numeral.
2. If `X` is a numeral, then so is `succ(X)`
   Note: `succ(X)` represents the successor of X.

We can define the above axiom in Prolog as follows:

```
numeral(0).
numeral(succ(X)):- numeral(X).
```

Exercise

Try to execute the above script and the following query: `?- numeral(X)..`

Defining Natural Numbers

We see that, natural numbers can be defined in the following way:

- `0 = 0;`

Defining Natural Numbers

We see that, natural numbers can be defined in the following way:

- `0 = 0;`
- `1 = succ(0);`

Defining Natural Numbers

We see that, natural numbers can be defined in the following way:

- `0 = 0;`
- `1 = succ(0);`
- `2 = succ(succ((0));`

Defining Natural Numbers

We see that, natural numbers can be defined in the following way:

- `0 = 0;`
- `1 = succ(0);`
- `2 = succ(succ((0));`
- `3 = succ(succ(succ((0)));`

Defining Natural Numbers

We see that, natural numbers can be defined in the following way:

- `0 = 0;`
- `1 = succ(0);`
- `2 = succ(succ((0));`
- `3 = succ(succ(succ((0)));`
- `4 = succ(succ(succ(succ((0))));`
- and so on.

Contents

Addition

- How do we define addition of two natural numbers?

Addition

- How do we define addition of two natural numbers?
- Since we were kid, we were told that: $0 + 1 = 1$, $1 + 1 = 2$, and so on.

Addition

- How do we define addition of two natural numbers?
- Since we were kid, we were told that: $0 + 1 = 1$, $1 + 1 = 2$, and so on.
- Okay, now suppose we are using Peano arithmetic, how do we define addition of two numbers?

Addition

- How do we define addition of two natural numbers?
- Since we were kid, we were told that: $0 + 1 = 1$, $1 + 1 = 2$, and so on.
- Okay, now suppose we are using Peano arithmetic, how do we define addition of two numbers?
- Let `add/3` be a ternary predicate such that `add(X,Y,Z)` is true whenever `Z` is equal to `X` plus `Y` in Peano arithmetic notation.

Example

We have

1. `add(succ(0),0,succ(0))` is true.
2. `add(succ(0),succ(0),Result)` implies `Result =`

Addition

- How do we define addition of two natural numbers?
- Since we were kid, we were told that: $0 + 1 = 1$, $1 + 1 = 2$, and so on.
- Okay, now suppose we are using Peano arithmetic, how do we define addition of two numbers?
- Let `add/3` be a ternary predicate such that `add(X,Y,Z)` is true whenever `Z` is equal to `X` plus `Y` in Peano arithmetic notation.

Example

We have

1. `add(succ(0),0,succ(0))` is true.
2. `add(succ(0),succ(0),Result)` implies `Result = succ(succ(0))`.
3. `add(succ(0),succ(succ(0)),Result)` implies `Result =`

Addition

- How do we define addition of two natural numbers?
- Since we were kid, we were told that: $0 + 1 = 1$, $1 + 1 = 2$, and so on.
- Okay, now suppose we are using Peano arithmetic, how do we define addition of two numbers?
- Let `add/3` be a ternary predicate such that `add(X,Y,Z)` is true whenever `Z` is equal to `X` plus `Y` in Peano arithmetic notation.

Example

We have

1. `add(succ(0),0,succ(0))` is true.
2. `add(succ(0),succ(0),Result)` implies `Result = succ(succ(0))`.
3. `add(succ(0),succ(succ(0)),Result)` implies `Result = succ(succ(succ(0)))`.
4. `add(succ(succ(succ(0))),succ(succ(0)),Result)` implies `Result =`

Addition

- How do we define addition of two natural numbers?
- Since we were kid, we were told that: $0 + 1 = 1$, $1 + 1 = 2$, and so on.
- Okay, now suppose we are using Peano arithmetic, how do we define addition of two numbers?
- Let `add/3` be a ternary predicate such that `add(X,Y,Z)` is true whenever `Z` is equal to `X` plus `Y` in Peano arithmetic notation.

Example

We have

1. `add(succ(0),0,succ(0))` is true.
2. `add(succ(0),succ(0),Result)` implies `Result = succ(succ(0))`.
3. `add(succ(0),succ(succ(0)),Result)` implies `Result = succ(succ(succ(0)))`.
4. `add(succ(succ(succ(0))),succ(succ(0)),Result)` implies `Result = succ(succ(succ(succ(succ(0)))))`.

Some Important Axioms

- To define additions of two numbers, we need several axioms from algebra.

Some Important Axioms

- To define additions of two numbers, we need several axioms from algebra.
- Axiom 1: for every number x, we have $x + 0 = x$. In Prolog, this axiom can be encoded as `add(0,X,X).` This clause becomes the base case.

Some Important Axioms

- To define additions of two numbers, we need several axioms from algebra.
- Axiom 1: for every number x, we have $x + 0 = x$. In Prolog, this axiom can be encoded as `add(0,X,X).` This clause becomes the base case.
- Since $x + 0 = 0 + x = x$, we can also include the clause `add(X,0,X).`
- What if both argument are not zero? How do we define the recursive case?

Some Important Axioms

- To define additions of two numbers, we need several axioms from algebra.
- Axiom 1: for every number x, we have $x + 0 = x$. In Prolog, this axiom can be encoded as `add(0,X,X).` This clause becomes the base case.
- Since $x + 0 = 0 + x = x$, we can also include the clause `add(X,0,X)`.
- What if both argument are not zero? How do we define the recursive case?
- Suppose we are interested to define `add(X,Y,Z)` where `X`, `Y`, `Z` are not zero.

Some Important Axioms

- To define additions of two numbers, we need several axioms from algebra.
- Axiom 1: for every number x, we have $x + 0 = x$. In Prolog, this axiom can be encoded as `add(0,X,X).` This clause becomes the base case.
- Since $x + 0 = 0 + x = x$, we can also include the clause `add(X,0,X)`.
- What if both argument are not zero? How do we define the recursive case?
- Suppose we are interested to define `add(X,Y,Z)` where `X`, `Y`, `Z` are not zero.
- Notice that `add(succ(X),Y,succ(Z))` is true if `add(X,Y,Z)` is true.

Some Important Axioms

- To define additions of two numbers, we need several axioms from algebra.
- Axiom 1: for every number x, we have $x + 0 = x$. In Prolog, this axiom can be encoded as `add(0,X,X).` This clause becomes the base case.
- Since $x + 0 = 0 + x = x$, we can also include the clause `add(X,0,X).`
- What if both argument are not zero? How do we define the recursive case?
- Suppose we are interested to define `add(X,Y,Z)` where `X`, `Y`, `Z` are not zero.
- Notice that `add(succ(X),Y,succ(Z))` is true if `add(X,Y,Z)` is true.
- We also have `add(X,succ(Y),succ(Z))` is true if `add(X,Y,Z)` is true.

Addition of Natural Numbers

```
numeral(0).
numeral(succ(X)):- numeral(X).
add(0,X,X). add(X,0,X).
add(succ(X),Y,succ(Z)):- add(X,Y,Z).
add(X,succ(Y),succ(Z)):- add(X,Y,Z).
```

However, the rules can be simplified as:

Addition of Natural Numbers

```
numeral(0).
numeral(succ(X)):- numeral(X).
add(0,X,X).
add(succ(X),Y,succ(Z)):- add(X,Y,Z).
```

Try to execute the above Prolog script!

Contents

Declarative and Procedural Meaning

Suppose we have a rule of the form `P:- Q,R`, where `P`, `Q`, and `R` are terms. This rule has two meanings.

1. Declarative meaning:
   - `P` is true if `Q` and `R` are true.
   - From `Q` and `R` follows `P`.
2. Procedural meaning:
   - To solve `P`, first solve `Q`, and then solve `R`.
   - To satisfy `P`, first satisfy `Q` and then `R`.

Declarative versus Procedural Semantics

- Declarative meaning does not depend on the order of the clauses and the order of the goals in clauses.
- Procedural meaning does depend on the order of goals and clauses:
  - The order can affect the efficiency of the program.
  - An unsuitable order may even lead to **infinite recursive calls**.

Simple Experiment 1

What are the differences of the following scripts?

Natural Numbers 1

```
numeral(0).
numeral(succ(X)):- numeral(X).
```

and

Natural Numbers 2

```
numeral(succ(X)):- numeral(X).
numeral(0).
```

Interrupting Process and Avoiding Infinite Recursion

Remark

In Windows to *interrupt* SWI-Prolog process, select *run*, then *interrupt*, and type a for bringing then *abort* command. If this step is unsuccessful, you may restart your Prolog interpreter.

Remark

For any recursive predicate, **it is necessary to define the base case first**.

Simple Experiment 2

- Now we might ask ourselves, is there any difference between `P:- Q,R` and `P:- R,Q`.
- Recall the following knowledge base that express possible flights among cities.

```
direct(jakarta,yogyakarta).
direct(jakarta,surabaya).
direct(jakarta,bali).
direct(bandung,yogyakarta).
direct(bandung,surabaya).
direct(bandung,bali).
direct(surabaya,makassar).
direct(makassar,manado).
```

- We have several ways of expressing whether there is a possible flight from city `A` to city `B`.

Expressing Flights

```
flight1(X,Y):- direct(X,Y).
flight1(X,Y):- direct(X,Z), flight1(Z,Y).
% original version

flight2(X,Y):- direct(X,Y).
flight2(X,Y):- flight2(Z,Y), direct(X,Z).
% swap clauses of the second definition for flight1

flight3(X,Y):- direct(X,Y).
flight3(X,Y):- direct(Z,Y), flight3(X,Z).
% another modification of flight1

flight4(X,Y):- direct(X,Y).
flight4(X,Y):- flight4(X,Z), direct(Z,Y).
% swap clauses of the second definition for flight 3
```

Exercise

Using the the knowledge base about the flights between two cities, try posing the following queries:

1. `flight1(jakarta,manado)`. Do the same thing for `flight2`, `flight3`, and `flight4`.
2. `flight1(manado,jakarta)`. Do the same thing for `flight2`, `flight3`, and `flight4`.
3. `flight1(jakarta,jakarta)`. Do the same thing for `flight2`, `flight3`, and `flight4`.
4. `flight1(jakarta,X)`. Do the same thing for `flight2`, `flight3`, and `flight4`.
5. `flight1(X,jakarta)`. Do the same thing for `flight2`, `flight3`, and `flight4`.

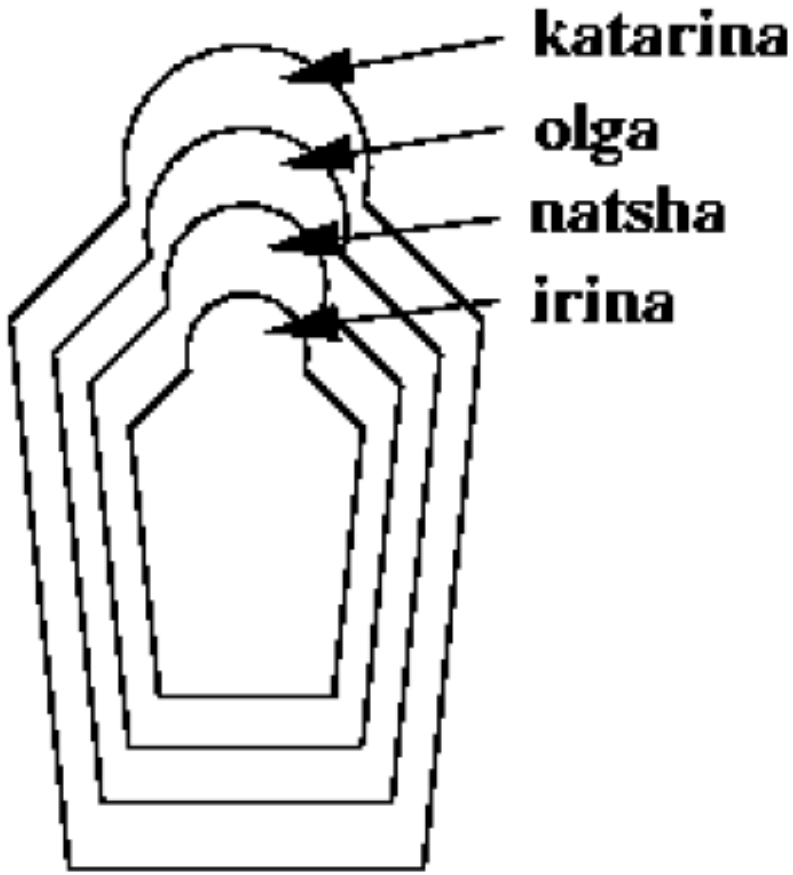
Remark

General heuristic in problem solving: **it is usually best to try the simplest idea first!**

Contents

Matryoshka Doll

Suppose we have following illustration.



Exercise

Define a predicate `in/2`, that tells us which doll is (directly or indirectly) contained in which other doll. E.g., the query `in(katarina,natasha)` should evaluate to true, while `in(olga,katarina)` should fail.

Greater Than...

Exercise

Define a predicate `greater_than/2` that takes two numerals in the notation that we introduced in this lecture (i.e., `0`, `succ(0)`, `succ(succ(0))`, ...) as arguments

and decides whether the first one is greater than the second one. E.g:

- `?- greater_than(succ(succ(succ(0))),succ(0)).` evaluates to true.
- `?- greater_than(succ(succ(0)),succ(succ(succ(0)))).` evaluates to false.