U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO
DEPARTMENT OF INFORMATICS ENGINEERING

U.PORTO FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO
DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2024/2025 – 1st Semester

Introduction to Prolog

13/11/2024

Daniel Castro Silva
dcs@fe.up.pt

Agenda

- Facts and Rules
- Queries
- How Prolog works

Prolog

- Prolog is the most widely used logic programming language
  - There are some language dialects, such as Edinburgh Prolog, and also a standardization – ISO Prolog
    - See Péter Szabó & Péter Szeredi, Improving the ISO Prolog Standard by Analyzing Compliance Test Results. Proc. of the 2006 Int'l Conf. on Logic Programming, pp 257–269
-

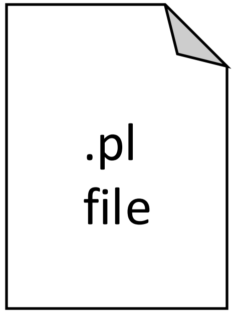
Table 1. ISO compliance test statistics of 1012 test cases

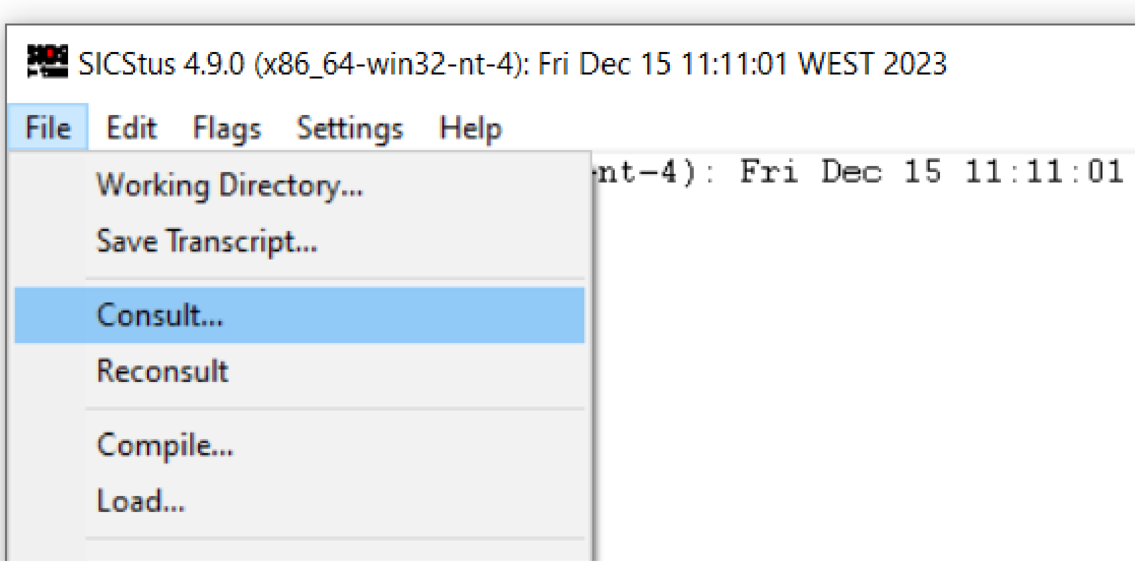
system	version	#OK	#failed	#dangerous	#irrelevant
SICStus	3.12.3	1010	1	0	1
aprolog	1.22	996	7	0	9
gprolog	1.2.16	929	67	7	9
SWI-Prolog	5.4.7	816	158	8	30
YAP	5.0.1	632	363	7	10
Ciao-Prolog	1.10p6	541	454	7	10

In this course, we'll be using SICStus Prolog v 4.9 (link to installer and keys available in Moodle)

Prolog

- Write your code in a text file with a .pl extension
  - Use the text editor of your choice
- In SICStus, load it using the *File* -> *Consult*... menu
  - Or call directly on the SICStus console: `| ?- consult('path/to/file.pl').`





Alternatively, you can use SPIDER (SICStus Prolog IDE, based on Eclipse)

Basic Idea

- Describe the situation of interest

- Ask a question

- Prolog:
  - logically deduces new facts about the situation we described
  - gives us its deductions back as answers

Facts

- Facts (aka knowledge base) express a relation that is true
  - You can (kind of) interpret them as lines in a database table

Statements end with a period

```
male(homer).            % homer is a male
female(marge).          % marge is a female
father(homer, bart).    % homer is the father of bart
mother(marge, bart).    % marge is the mother of bart
```

Arguments between parentheses and separated by commas

Predicate (relation) names start with lowercase letter

Semantics

- The semantics (interpretation) needs to be defined and shared

```
father(homer, bart).       % homer is the father of bart
father(homer, bart).       % the father of homer is bart
```

- This inherent ambiguity only highlights the importance of using appropriate and descriptive names as well as code comments

```
% single-line comment

/* multi-line
   comment */
```

Naming conventions and code comments represent a part of the evaluation of the practical assignment

Rules

- Rules allow for the deduction of new knowledge from existing knowledge (facts and other rules)
  - Rules are expressed in the form of Horn Clauses:
    - Head :- Body

```
grandfather(X, Y):- father(X, Z), parent(Z, Y).     % X is the grandfather of Y
                                                     % if X is the father of Z
                                                     % and Z is a parent of Y
```

neck

and

```
%multiple definitions of a rule with the same head: rule one or rule two or...

parent(X, Y):- father(X, Y).     % X is a parent of Y if X is the father of Y
parent(X, Y):- mother(X, Y).     % X is a parent of Y if X is the mother of Y
```

Disjunction

- Disjunction can also be expressed with the ; operator

```
parent(X, Y):- father(X, Y).      % X is a parent of Y if X is the father of Y
parent(X, Y):- mother(X, Y).      % X is a parent of Y if X is the mother of Y

% is equivalent to

parent(X, Y):- father(X, Y) ; mother(X, Y).
```

- The disjunction operator (;) should be used sparingly
  - Always use parentheses to clarify

Rules

- Rules have both a declarative and a procedural interpretation
  - Declarative interpretation

```
grandfather(X, Y):-          % X is the grandfather of Y
        father(X, Z),        % if X is the father of Z
        parent(Z, Y).        % and Z is a parent of Y
```

  - Procedural interpretation

```
grandfather(X, Y):-          % to solve grandfather(X,Y)
        father(X, Z),        % first solve father(X, Z)
        parent(Z, Y).        % and then parent(Z, Y)
                             %  (solve = execute)
```

Rules

- The head of a rule can have 0 or more arguments

```
parent(X, Y):- father(X, Y).      % X is a parent of Y if X is the father of Y

father(X):- father(X, Y).         % X is a father if he is the father of some Y

fathers:- father(X, Y).           % fathers is true if there is a(t least one)
                                  % father/child relation
```

A rule with no arguments is a good entry point to a program

Prolog and Logic

	Prolog	Logic
Implication	A :- B	B -> A
Conjunction	A , B	$A \land B$
Disjunction	A ; B	$A \lor B$

Prolog Programs

- A Prolog program is a finite set of **predicates**
  - `grandfather, father,` etc. are predicates
  - Predicates use facts and rules to express knowledge as relations
  - Relations are generalizations of functions
    - Usually more versatile, usable in multiple directions
- A computation is a proof of a goal from a program
  - Using [a form of] Selective Linear Definite (SLD) clause resolution with a unification algorithm
- A **correct** program does not allow the deduction of unwanted facts
- A **complete** program allows the deduction of everything intended

Terms

- Everything in Prolog is a *term*, which can be a *constant*, a *variable* or a *compound term*
- **Constants** represent elementary objects
  - **Numbers**
    - **Integers** (e.g., 4, -8) (bases other than decimal can also be used, e.g., 8’755)
    - **Floats** (e.g., 1.5, -1.6) (also supports exponent, e.g., 23.4E-2)
  - **Atoms**
    - Start with lower-case letter (e.g., john_doe, johnSmith42)
    - String within single quotes (e.g., ‘John Doe’, ‘John Smith 42’)

Terms

- **Variables** act as placeholders for arbitrary terms
  - Start with a capital letter (e.g., Variable1)
  - Start with an underscore (e.g., _Var2)
  - Single underscore (_) (anonymous variable)
- **Compound terms** are comprised of a *functor* and *arguments* (which are terms)
  - The **functor** is characterized by its *name* (an atom) and *arity* (the number of arguments), usually represented as *name/arity*
  - E.g., point/2 represents a functor named *point* with two arguments
    - point(4, 2) is a possible instance of point/2, and so is point(foo, point(3, bar))

Variables in Programs

- Variables are universally instantiated in logic programs

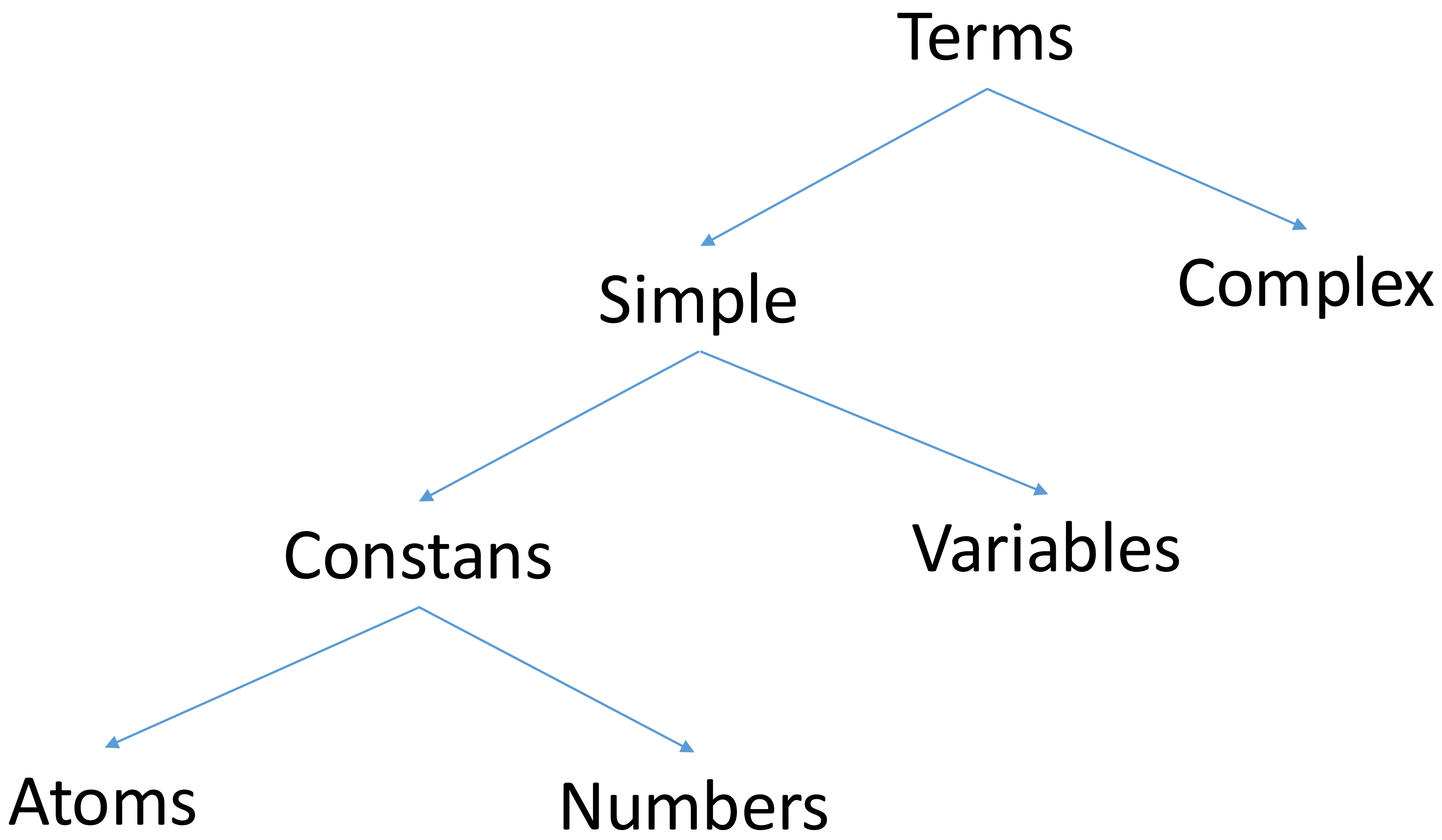
```
plus(0, S, S).                % 0 is the neutral element of addition
mult(1, V, V).                % 1 is the neutral element of multiplication


human(Homer).                 % everything is human
father(homer, Bart)           % homer is the father of everything


grandfather(X, Y):- father(X, Z), parent(Z, Y).
```

Variables occurring only in the body of a rule can be seen as existentially quantified

We need to be careful when using variables with facts


Terms
Simple
Complex
Constans
Variables
Atoms
Numbers

Complex Terms

- Atoms, numbers and variables are building blocks for **complex terms**

- Complex terms are built out of a **functor** directly followed by a sequence of **arguments**
  - Arguments are put in round brackets, separated by commas
  - The functor must be an atom

- Ex.

```
human(Homer).
father(homer, Bart).
```

Coding Efficiency Considerations

- Use implicit unification instead of additional variables

```
change_player(X, Y):- X = 1, Y = 2.
change_player(X, Y):- X = 2, Y = 1.
```

Should instead be written as

```
change_player(1, 2).
change_player(2, 1).
```

- Always place input arguments before output arguments
  - SICStus indexes predicates by their first argument

= is the unification operator (kind of ‘[possibly] equal’);
\= (not unifiable) can be interpreted as ‘can’t be equal’

Coding Style Considerations

- Although white space and code indentation are meaningless, there are some coding style guidelines you should consider following, to increase code readability:
  - Indent the code consistently
  - Put each sub-goal on a separate, indented line
  - Use human-readable names for predicates and variables
  - Try to limit the length of code lines and number of lines per clause
  - …

See Covington et al. (2012). Coding Guidelines for Prolog. Theory and Practice of Logic Programming, 12(6): 889-927

Agenda

- Facts and Rules
- Queries
- How Prolog works

Queries

- Computations in Prolog start with a question, which has two possible answers:
  - Yes (possibly with answer substitution – variable binding)
  - No
- The attempt to prove the question right/wrong (is it a consequence of the program?) produces the computations

```
| ?- male(homer).
yes

| ?- father(homer, bart).
yes
```

```
| ?- female(marge).
yes

| ?- father(marge, bart).
no
```

Variables in Queries

- Queries can include variables
  - Variables are existentially quantified in queries
- A variable starting with an underscore is a '*don't care*'

```
| ?- father(X, bart).
X = homer ?
yes

| ?- father(_X, bart).
yes

| ?- male(_).
yes
```

```
| ?- male(X).
X = homer ?
yes

| ?- male(X).
X = homer ? ;
X = bart ? n
no
```

If satisfied with the answer, just hit enter
If you want another answer, type 'n', 'no' or ';'

Variables and Compound Queries

- Queries can be more complex, combining goals
- Variables are used to glue together the different goals
  - Underscore alone (_) is the exception

```
| ?- male(X), parent(X, bart).
X = homer ? ;
no

| ?- male(_X), parent(_X, bart).
yes
```

```
| ?- male(_X), parent(Y, bart).
Y = homer ? ;
Y = marge ? ;
Y = homer ? ;
Y = marge ? ;
no
```

Why the duplicates?
Just wait a few slides!

Closed World Assumption

- Assumption that everything that is true is known to be true (i.e., is represented as a clause in the program)

- Therefore, everything that cannot be deduced from the clauses in the program is assumed to be false

```
| ?- male(donald).
no
```

  - Requires attention to make sure everything we want to deduce can be deduced from the program clauses

Horn Clauses

- Everything in Prolog is expressed as a Horn Clause
  - Rules are complete horn clauses (head :- body)

```
parent(X, Y):- father(X, Y).    ⇔    father(X, Y) => parent(X, Y)
```

  - Facts are horn clauses where the body is always true (just the head)

```
male(homer):- true.        ⇔        male(homer).
```

  - Queries are horn clauses without a head (just the body)

```
| ?- father(X, bart).
```

Predicates

- A **predicate** is a set of clauses for the same functor
  - **Clauses** are either facts or rules
  - `parent` is a predicate with two clauses

```
parent(marge, bart).
parent(homer, bart).
```

- Functors with the same name but different arity refer to different predicates

```
father(X):- father(X, Y).     % X is a father
                              % if X is the father of some Y
```

Documentation

- Documentation should include a **mode declaration** for each argument:
  - + (input): the argument is instantiated when the predicate is called
  - - (output): the argument is not instantiated in the predicate call
  - ? (in/out): the argument can be instantiated or not

```
% square(+number, -square)          % calculates the square
                                    % of a given number

% parent(?parent, ?child)           % parent/child relation
```

- One of the most powerful properties of Prolog is its versatility

Prolog Versatility

- The versatility of Prolog can be seen in most predicates:
  - For instance, parent/2 allows:
    - Confirming that two given people are parent/child
    - Obtaining the children of a given person
    - Obtaining the parents of a given person
    - Obtaining all parent/child pairs
  - In most other languages, we would need to implement four different functions to achieve this, or include extra logic to test instantiation

Prolog and Relational Algebra

- A Prolog program can be seen as a database
  - Facts represent tables
  - Rules represent views

- Prolog can be used to implement all relational algebra operations, like union, cartesian product, projection, selection, ...

Relational Operations

- Union

$$r\_union\_s(X_1, ..., X_n) \text{:- } r(X_1, ..., X_n).$$
$$r\_union\_s(X_1, ..., X_n) \text{:- } s(X_1, ..., X_n).$$

- Cartesian product

$$r\_times\_s(X_1, ..., X_m, X_{m+1}, ..., X_{m+n}) \text{:- } r(X_1, ..., X_m), s(X_{m+1}, ..., X_{m+n}).$$

- Projection

$$r\_1\_3(X_1, X_3) \text{:- } r(X_1, X_2, X_3).$$

- Selection

$$r\_1(X_1, X_2, X_3) \text{:- } r(X_1, X_2, X_3), X_2 > X_3.$$

- Intersection

$$r\_inters\_s(X_1, ..., X_n) \text{:- } r(X_1, ..., X_n), s(X_1, ..., X_n).$$

- Join

$$r\_join\_s(X_1, X_2, X_3) \text{:- } r(X_1, X_2), s(X_2, X_3).$$

- Difference

$$r\_minus\_s(X_1, ..., X_n) \text{:- } r(X_1, ..., X_n), \backslash + \; s(X_1, ..., X_n).$$

Prolog and Relational Algebra

- Complex terms vs ‘normalized’ facts

```
has(john, book('River God', author(smith, wilbur, 1933), 1993)).
```

vs.

```
author(a37, smith, wilbur, 1933).
book(b521, 'River God', 1993).
author(a37, b521).
person(p432, john).
has(p432, b521).
```

Complex Terms vs. Normalize Facts

Aspect	Complex Terms	Normalized Facts
Structure	Hierarchical / nested (e.g., `book('River God', author(smith, wilbur, 1933), 1993)`)	Flat and relational (e.g., `book(b521, 'River God', 1993). author(a37, smith, wilbur, 1933).`)
Compactness	Very compact — all info in one term	More verbose — split into multiple clauses
Readability	Intuitive for small examples	Closer to database style
Data Reuse	Hard (same subterm repeated)	Easy (link by identifiers)
Querying	Simple for direct relationships	More flexible and scalable for complex queries
Maintenance	Becomes cumbersome as data grows	Easier to extend and modify
Best suited for	Small, self-contained knowledge	Large, structured knowledge bases

Agenda

- Facts and Rules
- Queries
- How Prolog works

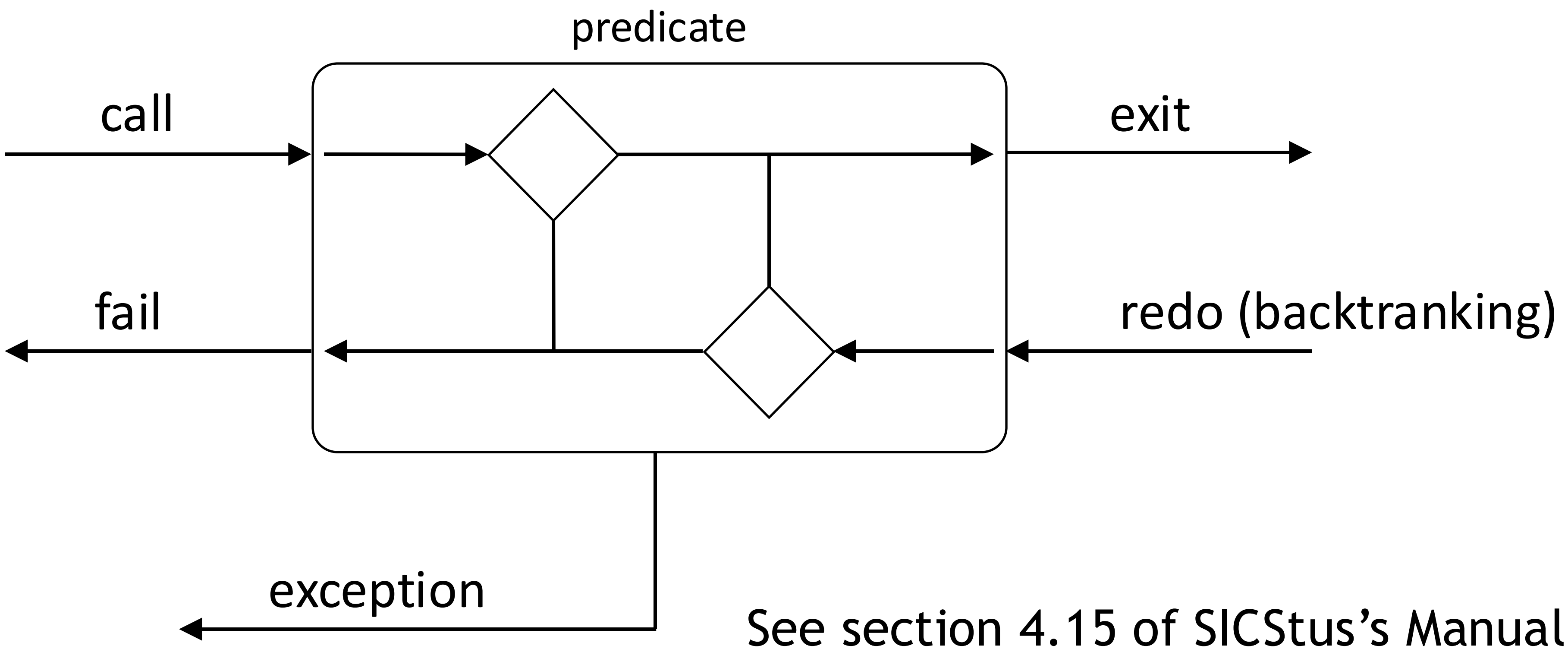
How Prolog Works

- Prolog’s mechanics work
  - Top to bottom
    - The order of clauses is important
  - Left to right
    - In rules, prove sub-goals in left-to-right order
  - With backtracking
    - If a sub-goal fails, go back to previous decision point

The Prolog Box Model

- Each call to a goal can be modelled as a four-gate box model

call: goal search
exit: goal succeeds
redo: backtracking
exit: goal cannot be satisfied



See section 4.15 of SICStus's Manual for more information on exceptions

Tracing

- Trace mode allows us to follow the computations step by step
  - Can be activated from the menu Flags -> Debugging -> trace
  - Or in the code, by calling *trace*
    - Disable it by calling *notrace*

```
foo(bar):- fubar(bar, baz),
           trace,            % activate trace mode
           qux(baz),         % the call to qux will be traced
           notrace,          % deactivate trace mode
           quux(bar).
```

See section 5 of SICStus's Manual for more information on Trace and Debugging

Tracing

- Trace message format:

```
N   S   InvID  Depth  Port: Goal ?
```

  - N (only visible at Exit ports) indicates that the goal call may backtrack to find alternative solutions
  - S indicates the existence of a spypoint
  - InvID (Invocation ID) is a unique identifier for each goal (can be used to match messages from the various ports)
  - Depth is an indication of the general call depth
  - Port is one of Call, Exit, Redo, Fail or Exception
  - Goal is the current goal of the computation

Additional Readings

- Prolog
  - Leon Sterling and Ehud Shapiro (1994). The Art of Prolog. The MIT Press (2nd ed). ISBN: 978-0262691635
  - Krzysztof R. Apt (1996). From Logic Programming to Prolog. Prentice Hall. ISBN: 978-0132303682
  - Patrick Blackburn, Johan Bos and Kristina Striegnitz (2006). Learn Prolog Now! College Publications. ISBN: 978-1904987178
  - Ivan Bratko (2011). Prolog Programming for Artificial Intelligence. Addison Wesley (4th ed). ISBN: 978-0321417466
  - Max Bramer (2013). Logic Programming with Prolog. Springer (2nd ed). ISBN: 978-1447154860