Logic Programming and Deductive Databases

Chapter 4: Prolog Syntax

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Summer 2021

http://www.informatik.uni-halle.de/~brass/lp21/

Objectives

After completing this chapter, you should be able to:

- write syntactically correct Prolog.
- explain the basic data structure for rules and terms (abstract syntax).
- use the operator syntax.

  And translate operator syntax into standard syntax.

- use list syntax.

Contents

Introduction (1)

- A pure Prolog program is a set P of definite Horn clauses (clauses with exactly one positive literal).

  Prolog uses an un-sorted (or one-sorted) logic.

- A query or (proof) goal Q in Prolog is a conjunction of positive literals.

  I.e. its negation for refutation provers gives a Horn clause with only negative literals.

- The purpose of a Prolog system is to compute substitutions θ such that $P \models Q\,\theta$.

  I.e. one wants values for the variables such that the query is true for these values in each model of the program.

Introduction (2)

- In Prolog it is possible that the computed substitutions θ with $P \models Q\theta$ are not ground.
- E.g. consider the query $q(X)$ for the program

$$q(X) \leftarrow p(X).$$
$$p(X).$$

  Then it is not necessary to replace X in the query by any concrete value. The program implies $\forall X\ q(X)$.

  Then one is not interested in all substitutions with $P \models Q\theta$, but only in a set of substitutions that "subsumes" all other substitutions.

- In deductive DBs, rules and queries are restricted such that only ground answers are computed.

Introduction (3)

- While in mathematical logic, the concrete syntax is not very important (e.g. one assumes any alphabet), this chapter explains the exact Prolog syntax.
- In Chapter 8, some features will be explained that are necessary for many practical Prolog programs, but do not have a nice logical semantics.
- The classical "impure" feature is the cut, but also arithmetic predicates and I/O make Prolog semantics more complicated.

Introduction (4)

- In contrast to the examples in Chapter 1, now function symbols are permitted.
- Function symbols are supported in Prolog and some modern deductive database systems.

  Originally, function symbols are not permitted in deductive databases, because then termination of query evaluation cannot be guaranteed.

- Function symbols are interpreted as term constructors, e.g. for lists. In logic programming, one basically considers only Herbrand interpretations.

  I.e. function symbols are not interpreted ("free interpretation").

Contents

Lexical Syntax (1)

Prolog Atoms:

- Lowercase (Letter | Digit | _)*

  E.g.: thisIsAnAtom, x27, also_this_is_permitted.

- ' (arbitrary characters)* '

  If the sequence of characters contains ', one must double it, e.g. `'John''s'`, or escape it with "\", e.g. `'John\'s'`. Modern Prologs support many more escape sequences starting with "\". If one needs "\" itself, one must write "\\" instead. The Prolog Standard seems to say that the quote is doubled. SWI Prolog understands both variants. Note: `'a'` and `a` are the same atom.

- (#|$|&|*|+|-|.|/|:|<|=|>|?|@|\|^|~)+

  But: "." followed by whitespace marks the end of the clause.
  Another exception is "/*", which starts a comment.

- Special atoms: `!`, `;`, `[]`, `{}`.

Lexical Syntax (2)

Constants in Prolog:

- Atoms (see above): e.g. `red`, `green`, ..., `monday`, ...

  Atoms are internally represented as pointers to a symbol table.

- Integers: e.g. `23`, `-765`, `16'1F` (=31), `0'a` (=97)

  ⟨`Radix`⟩`'`⟨`Number`⟩ is the Edinburgh Prolog syntax. The ISO-Standard requires instead that hexadecimal numbers start with `0x`, octal numbers with `0o`, and binary numbers with `0b`. It supports `0'` for the ASCII-code. The Edinburgh Prolog syntax is probably more portable.

- Floating point numbers: e.g. `-1.23E5`.

- Strings: e.g. `"abc"`.

Lexical Syntax (3)

Strings in Prolog:

- In classical Prolog systems, strings are represented as lists of ASCII codes, e.g. `"abc"` is `[97,98,99]`.
- This makes string processing easy and flexible, but each character might need e.g. 16 bytes of storage.

  Also, `write("abc")` prints `[97,98,99]`.
- Modern Prologs often represent strings as arrays of characters (as usual in other languages).

  This creates, however, portability problems: Old programs might not run.

  In ECLiPSe, the conversion is done with `string_list(String, List)`. In SWI Prolog, it is `string_to_list(String, List)`.

Lexical Syntax (4)

Atoms vs. Strings:

- Atoms are internally represented as pointers into a symbol table ("dictionary").
- Therefore comparing and copying them is very fast.
- However, creating a new atom takes some time.
- Also, once an atom is created, it is never deleted from memory (depending on the Prolog system).
- It is possible to create atoms dynamically (at runtime), but one should do this only if one expects to reference them again and again.

Lexical Syntax (5)

Predicates and Function Symbols in Prolog:

- Predicate and function symbol names are atoms.
- Prolog permits to use the same name with different arities. These are different predicates, e.g.:

```
father(Y) :- father(X,Y).
```

- In the Prolog literature, one normally writes p/n for a predicate with name p and arity n.

  Remember that the arity is the number of arguments. E.g. the above rule contains the predicate `father/1` in the head, and the predicate `father/2` in the body. There is no link between these two distinct predicates except what is explicitly specified with the rule.

Lexical Syntax (6)

Variables in Prolog:

- (Uppercase | _) (Letter | Digit | _)*
- Exception: "_" (anonymous variable):
  Each occurrence denotes a new system-generated variable.
- Many Prolog systems print a warning ("singleton variable") if a variable that appears only once in a rule does not start with an underscore "_".

  This helps to protect against typing errors in variable names: If the variable really appears only once, one could as well use the anonymous variable.

Lexical Syntax (7)

Comments in Prolog:

- From "`%`" to the line end (as in TEX).
- From "`/*`" to "`*/`" (as in C).

Logical Symbols in Prolog:

- "`:-`" for $\leftarrow$.
- "`,`" for $\wedge$.

Contents

Abstract Prolog Syntax (1)

- The abstract syntax describes the data structures that the parser creates (e.g. operator tree).
- The concrete syntax defines e.g. operator priorities, abbreviations, and special "syntactical sugar".

  E.g. the concrete input might contain parentheses and special delimiter characters that are not contained in the internal representation of the program.

- Prolog was originally an interpreted language.

  Today, it is typically compiled into byte code for the "WAM".

- Then the abstract syntax describes the data structures on which the interpreter works.

Abstract Prolog Syntax (2)

Program:

- A program is a sequence of clauses.

Clause:

- A clause is one of the following:
  - Fact: A literal.

    Literal means here always "positive literal".
  - Rule: Consists of a literal and a goal.

    The literal is called the head of the rule, and the goal is called the body of the rule.
  - Query/Command: A goal.

Abstract Prolog Syntax (3)

Goal (simple version, this chapter):

- A goal is a sequence of literals.

Goal (complex version, later):

- A goal is one of the following:
  - A literal.
  - A cut.
  - A conjunction of two goals.
  - A disjunction of two goals.
  - If goal, then goal, possibly else goal.

Abstract Prolog Syntax (4)

Literal (Positive Literal):

- A literal consists of
  - An atom p, and
  - n terms $t_1, \ldots, t_n$, $n \geq 0$.
- p/n is the predicate of the literal.
- t_i is the i-th argument of the literal.
- If A is a literal, let $pred(A)$ denote the predicate of A.

Abstract Prolog Syntax (5)

Term:

- A term is one of the following:
  - Variable (the anonymous variable is treated specially)
  - Atom
  - A composed term consisting of an atom f and $n \geq 1$ terms $t_1, \ldots, t_n$.

    f/n is the functor of this term.
  - Number: integer, real, possibly rationals etc.
  - String (if this is not a list of ASCII codes).
  - Stream (open file).
  - . . . (possibly other types of objects).

Contents

Operator Syntax (1)

Example:

- `+(1, 1)` is a term in standard syntax.

  Standard syntax is $f(t_1, \ldots, t_n)$ for a composed term with functor f/n and arguments $t_1, \ldots, t_n$.

- `1+1` is the same term in operator syntax.

- This is only a more convenient input syntax.
  Internally, `+(1,1)` and `1+1` are the same term.

  There is absolutely no difference in their meaning.

- Operator syntax can be used also for literals, e.g. `X \= Y`, `5 < 7`.

Operator Syntax (2)

Operators:

- Many operators are predeclared, but the Prolog user can declare new operators.

  Declaring an operator only modifies the input syntax of Prolog (in Prolog programs and user input read with the built-in predicate `read`). By itself, it does not associate any specific meaning with the operator.

- An operator has
  - Name: Any Prolog atom.
  - Priority: From 1 (high priority) to 1200 (low).

    E.g. $*$ (priority 400) binds more strongly than $+$ (priority 500).
  - Associativity: One of `fx`, `fy`, `xf`, `yf`, `xfx`, `yfx`, `xfy`.

Operator Syntax (3)

Operator Types:

- `fx`, `fy`: Prefix operator, e.g. "`-X`".
- `xf`, `yf`: Postfix operator, e.g. "`7!`".
- `xfx`, `yfx`, `xfy`: Infix operator, e.g. "`1 + 1`".

Associativity:

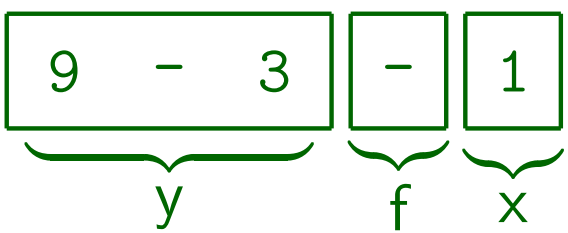
- `x`: term, the topmost operator of which has a numerically lower priority (which means really higher priority).
- `y`: term with numerically lower or equal priority.

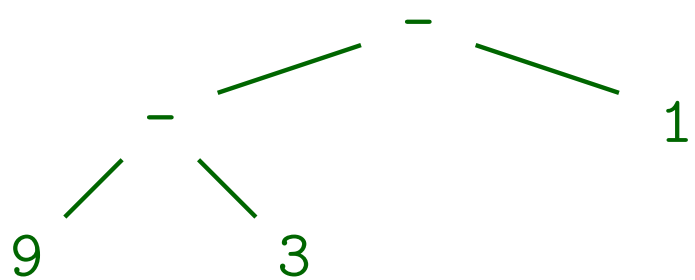
Operator Syntax (4)

Example for Associativity:

- `-` has type `yfx`, i.e. another `-` can be in the left operand, but not in the right (except inside "`(...)`").
- Thus, the term `9 - 3 - 1` means `-(-(9,3),1)`.

  "`-`" is a left-associative operator.





- E.g. `X is 9 - 3 - 1` binds `X = 5`.

  And not to 7, which would be the result of `9 - (3 - 1)`.

Operator Syntax (5)

Showing the Structure of Terms:

- `display(`T`)` prints the term T in standard syntax.
- E.g. `display(9-3-1)` prints `-(-(9,3),1)`.
- `write(`T`)` prints the term T using operator syntax.

  For function symbols declared as operators. Otherwise, standard syntax is the only option. This does not depend on the use of operator syntax or standard syntax for input of the term: The original syntax is not stored in the internal data structures. E.g. `write(+(1,1))` prints `1+1`.

Querying Declared Operators:

- "`current_op(Prio, Type, Operator)`" can be queried to get a list of all declared operators.

  "`current_op`" is one of many built-in predicates, i.e. predicates that are not defined by clauses, but by a procedure inside the Prolog system.

Operator Syntax (6)

Operator Declaration:

- A new operator is declared by calling/executing the built-in predicate "`op(Prio, Type, Operator)`".
- E.g. after executing the goal

  ```
  op(700, xfx, is_child_of)
  ```

  the following is a legal syntax for a fact:

  ```
  emil is_child_of birgit.
  ```

- It is completely equivalent to

  ```
  is_child_of(emil, birgit).
  ```

Operator Syntax (7)

Note:

- Besides facts and rules, a Prolog program can contain goals. One must write "`:- op(...).`"

  Unless one enters "`op(...)`" interactively, in which case the Prolog system is already in query mode. But if one should write "`op(...)`" as a fact in a file, one will probably get an error message that one tries to redefine a built-in predicate.

- The Prolog compiler executes this while compiling the program. It modifies the internal parser tables.

- One can then use the operator in the rest of the same input file and in later user input.

Predefined Operators (1)

Logic, Control:

Op.	Priority	Type	Meaning
`:-`	1200	xfx	"if" in rules
`:-`	1200	fx	marks a goal
`-->`	1200	xfx	syntax rule
`;`	1100	xfy	disjunction (or)
`->`	1050	xfy	then (for if-then-else)
`,`	1000	xfy	conjunction (and)
`\+`	900	fy	negation as failure

Predefined Operators (2)

Arithmetic Comparisons:

Op.	Priority	Type	Meaning
`<`	700	xfx	is less than
`>`	700	xfx	is greater than
`>=`	700	xfx	greater than or equal
`=<`	700	xfx	less than or equal
`=:=`	700	xfx	is equal to
`=\=`	700	xfx	is not equal to
`is`	700	xfx	evaluate and assign

- These functions evaluate arithmetic expressions in their arguments (`is` only on the right side).

Predefined Operators (3)

Arithmetics:

Op.	Priority	Type	Meaning
`+`	500	yfx	sum
`+`	200	fx	identity (monadic +)
`-`	500	yfx	difference
`-`	200	fx	sign inversion (monadic -)
`*`	400	yfx	product
`/`	400	yfx	division (quotient)
`div`	400	yfx	integer division (floor)
`//`	400	yfx	integer division (toward zero)
`mod`	400	xfx	modulo (division rest for `div`)
`rem`	400	xfx	modulo (division rest for `//`)

Predefined Operators (4)

Bit Operations:

Op.	Priority	Type	Meaning
/\	500	yfx	bitwise and
\/	500	yfx	bitwise or
>>	400	yfx	right shift
<<	400	yfx	left shift
\	200	fx	bitwise negation

Predefined Operators (5)

Term Comparisons:

Op.	Priority	Type	Meaning
=	700	xfx	does unify with
==	700	xfx	is strictly equal to
\==	700	xfx	is not strictly equal to
@<	700	xfx	comes before
@>	700	xfx	comes after
@=<	700	xfx	comes before or is equal
@=>	700	xfx	comes after or is equal

Term Conversion:

Op.	Priority	Type	Meaning
=..	700	xfx	convert term to list

Some Syntactical Restrictions

Restrictions in Mixed Syntax:

- In standard syntax "$f(t_1, \ldots, t_n)$", one cannot put a space between "f" and "$($".

  A space is necessary, when the argument of a prefix operator starts with a parenthesis, e.g. "`\+(1) > 2`" vs. "`\+ (1) > 2`".

- In standard syntax "$f(t_1, \ldots, t_n)$", the prioity of operators in the argument terms t_i must be < 1000.

  "`,`" is an operator with priority 1000. Use parentheses if necessary.

- If a prefix-operator is used as atom without arguments, it must be put into parentheses: `(op)`.

Contents

List Syntax (1)

- The functor "`./2`" is used as list constructor.

  SWI Prolog uses `'[|]'` instead of "`.`". That is quite logical (see below), but SWI Prolog is probably the only such system: "`.`" is the classical symbol. SWI Prolog introduced "dictionaries" and needed the "`.`" for that.

- The left argument is the first element of the list.
- The right argument is the rest of list.
- The atom "`[]`" is used to represent the empty list.
- E.g. the list $1, 2, 3$ can be written as

```
.(1, .(2, .(3, [])))
```

- However, Prolog accepts the abbreviation `[1, 2, 3]` for the above term.

  It is uncommon that one ever uses "`.`" explicitly.

List Syntax (2)

- I.e. [t_1, ..., t_n] is an abbreviation for

  .(t_1, ..., .(t_n, []) ...)

- One can also write "[X|Y]" for ".(X, Y)".

- More generally, also the abbreviation

  [t_1, ..., t_n | t_{n+1}]

  for the following term is accepted:

  .(t_1, ..., .(t_n, t_{n+1}) ...)

  I.e. after the vertical bar "|", one writes the rest of the list. Before it, the first list elements. [1 | 2, 3] is a syntax error. [1|2] is not a syntax error, but it would be a type error if Prolog were typed.

List Syntax (3)

- E.g. the following are different notations for the list $1, 2, 3$:
  - `[1, 2, 3]`.
  - `.(1, .(2, .(3, [])))`.
  - `[1, 2, 3 | []]`.
  - `[1 | [2, 3]]`.
  - `.(1, [2, 3])`.
- If one tries `write(`t`)` for each of these terms, the system will always print `[1, 2, 3]`.

  In SWI Prolog, try: `write('[|]'(1, '[|]'(2, '[|]'(3, [])))).`

List Syntax (4)

- Now list processing predicates are easy to define.
- E.g. `append(X, Y, Z)` is true iff the list `Z` is the concatenation of lists `X` and `Y`, e.g.

```
append([1, 2], [3, 4], [1, 2, 3, 4])
```

- It is defined as follows:

```
append([], L, L).
append([F|R], L, [F|RL]) :-
        append(R, L, RL).
```

- Some Prolog systems have append as a built-in predicate.

  In SWI-Prolog, it is defined in a library that is automatically loaded when append is called an not yet defined.

- Exercise: Define `member(X, L)`: `X` is an element of `L`.

Contents

Formal Prolog Syntax (1)

- The input is a term of priority 1200, followed by a "full stop":

  Term(1200) "."

  White space (a space, line break, etc.) must follow so that "." is recognized as "full stop".

- The term is interpreted as clause:
  ":-" and "," are declared as operators.

- There are certain type restrictions, e.g. the head of the clause cannot be a variable or number.

  Prolog requires that the predicate is an atom, and e.g. not a variable.

Formal Prolog Syntax (2)

Term(N):

- Operator(N,fx) Term(N-1)

  Exception: "`-1`" is a numeric constant, not a composed term. Furthermore, if "Term(N-1)" starts with "`(`", a space is required.

- Operator(N,fy) Term(N)
- Term(N-1) Operator(N,xfx) Term(N-1)
- Term(N-1) Operator(N,xfy) Term(N)
- Term(N) Operator(N,yfx) Term(N-1)

Formal Prolog Syntax (3)

Term(N), continued:

- Term(N-1) Operator(N,xf)
- Term(N) Operator(N,yf)
- Operator(N,fx/fy)

  Prefix-operators that are used as atom count as term of their priority, and not as term of priority 0 as other operators.

- Term(N-1)

  I.e. it is not required that Term(N) really contains an operator of priority N. It may also contain an operator of numerically lower priority (which means higher binding strength), or contain no further operators outside parentheses (elementary terms are generated by Term(0) below).

Formal Prolog Syntax (4)

Term(0):

- Atom

  The atom cannot be declared as prefix operator, see above.

- Variable
- Number
- String
- Atom "(" Arguments ")"
- "[" List "]"
- "(" Term(1200) ")"

Formal Prolog Syntax (5)

Arguments:

- Term(999)
- Term(999) “,” Arguments

List:

- Term(999)
- Term(999) “,” List
- Term(999) “|” Term(999)

More Syntax Examples

- The rule "`p :- q, r.`" can also be entered in standard syntax: `:-(p, ','(q, r)).`
- The following are all the same literal:
  - `X is Y+1`
  - `is(X, Y+1)`
  - `is(X, +(Y,1))`
  - `X is +(Y,1)`

  I.e. one can use arbitrary mixtures of operator syntax and standard syntax and even when an atom is defined as operator, one can use the standard syntax.