Teaching Beginners Prolog
How to Teach Prolog

2. Fassung

Ulrich Neumerkel

Institut für Computersprachen
Technische Universität Wien
A-1040 Wien, Austria
http://www.complang.tuwien.ac.at/ulrich
ulrich@mips.complang.tuwien.ac.at

I The “magic” of Prolog — Common obstacles

II How to read programs

III Course implementation — Programming environment

Part I
Common obstacles

- The “magic” of Prolog

  Prolog appears as magic **if** one tries to learn Prolog

  by looking at execution traces

  using side effects

  — Which introductory book does *not* cover them?

- Previous skills and habits
- Prolog’s syntax
- Naming of predicates and variables
- List differences

Syllabus

Two apparently conflicting goals:

- Training (project oriented)
  Larger projects do not work well
- Teaching (concept oriented)

Basics:

- Basic reading skills for understanding Prolog programs
- Avoiding common mistakes, develop coding style

Previous skills to build on

- Programming skills
- Mathematical skills
- Language skills

Previous (counterproductive) programming skills

The self-taught programmer

Bad programming habits

Severe handicap: Edit-Compile-Run-Dump-Debug

"Let the debugger explain what the program is doing"

- How do you make sure that your programs have no errors?
- Do you use assertions frequently?
- Do you write down assertions/consistency checks *before* you write the actual code?
- How do you test? How do you ensure that results are correct?
- How can the program falsify your claim of correctness?

Prolog shows no mercy upon the illiterate programmer.

Previous programming skills

Procedural languages

difference to Prolog not that large when knowing

- structured programming (proponents Dijkstra et al.) :
  - to avoid bad habits: Verify, don't run (& don't debug)
    unclear: how to ensure accurateness of spec?
  - *never* visualize execution
  - avoid anthropomorphisms — computer language $\neq$ language
    linguistic analogy not helpful
- invariants, pre- postconditions
- *testable* assertions — e.g. Eiffel

  seldom taught along with *practical* programming
- C's `assert.h` (Even in C you can do better!)

Programming and Mathematical skills

Beginners have lots of problems understanding Prolog because they never learned structured programming.

Mathematical skills

- mathematical logic as prerequisite
- calculational skills (e.g. manipulating formulæ)
- unification

Language skills

- Only helpful skill to build on.
- Many difficulties of Prolog can be clarified by reading programs in plain English.
- E.g. quantification problems in negation:

  female(Female) $\leftarrow$
      \+male(Female).

  Everything/everyone, **really** *everything/everyone that/who is not male is female.*
  Therefore: Since a chair/a hammer/the summer isn't male it is female etc.

Language skills cont.

```
female(Female) ←
    person(Female),
    \+male(Female).
```

Napoleon is a person (defined) but we haven't defined Napoleon as being male, so we assume he is female.

- Detect defaulty data structure definitions

```
is_tree(_Element).        % Everything is a tree.
is_tree(node(L, R)) ←
    is_tree(L),
    is_tree(R).
```

Prolog's Syntax, Difficulties

Minor typos make a student resort to bad habits

Comma vs. period

Prolog's syntax is not robust: "male(john)." is a goal or fact, depending on the context.

```
father_of(Father, Child) ←
    child_of(Child, Father),
    male(Father), % !

male(john).
...
```

Happens to 84% of students.

Prolog's syntax — increasing robustness

1. Redesign Prolog's syntax. (Prolog II)
2. Take a subset of existing syntax. (GUPU)

   make spacing and indentation significant

   (a) Each head, each goal goes into a single line.
   (b) Goals are indented. Heads are not indented.
   (c) Only comma can separate goals (i.e. *no* disjunction)
   (d) Different predicates are separated by blank lines.

```
a ← !,        ⟹      a ←
    c.                   !, % Don't play down the cut! !!
                         c.
```

⇒ more helpful error messages possible

Names of predicates

key to understanding

assignments for finding the right names

Misnomers

- action oriented prescriptive names

  append/3, reverse/2

  use past participle instead, sometimes noun

- leave the argument order open

  child/2, length/2

- pretend too general or too specific relation

  reverse/2, length/2

- tell the obvious: body_list//1

Finding a good predicate name

1. Start with intended types

   type1_type2_type_3_type4(Arg1, Arg2, Arg3, Arg4)

   "child of a person" : person_person/2

2. If name too general, refine

   person_person ⇒ child_person/2

   list_list/2 ⇒ list_reversedlist/2

3. Emphasize relation *between* arguments

   - shortcuts like prepositions
     child_of/2
   - past participles alone.
     list_reversed/2

Example of name finding

"length of a list"

- number_list/2 ⇒length_list/2
- list_number/2 ⇒ list_length/2
- Argument order not important
- Traditional names often too general (length/2)

Shorter names

Omit less important arguments *at the end*
shortened name ends with an underscore

country_(Country, Region, Population, ...)

Type definitions

Convention: is_type(Type) or type(Type)

- documentation purpose
- serve as template for predicates defined over data structures

O'Keefe-rules

- unsuitable (for beginners)
- deal with procedural aspects
- inputs and outputs
  atom_chars vs. atom_to_chars

Variable names

Lack of type system makes consistent naming essential

- for lists: [Singularform|Pluralform] , e.g. [X|Xs]
- naming void variables in the head

  e.g. _Xs instead of _

  member(X,[X|_]).
- state numbering (e.g. list differences)

Understanding differences

– misleading name: “difference list”

– misunderstanding: “difference lists” are not lists
  Student statement: “My Prolog doesn't have difference lists”

+ instead : list difference, difference of lists, differential list (?)

– differences too early

+ use grammars first

    more compact, less error-prone, less typing
    amazingly powerful
    compact string notation

– differences presented as incomplete data structures — “holes”

+ motivate differences with ground lists

+ differences are not specific to lists

+ differences and state

Part II
Reading of programs

Algorithm = Logic + Control

Common misinterpretation

Prolog program = Pure Prolog + Control predicates

Inpure parts required?

Separation of declarative and procedural aspects is not helpful.

Family of related reading techniques

Focus on distinct (abstract) parts/properties of the program

- informal reading in English
- declarative reading
- (almost) procedural reading
- termination reading
- resource consumption

Informal reading

use English to

- focus the student's attention on the meaning of program
- avoid operational details
- clarify notions
- clarify language ambiguities
- clarify confusion of "and" and "or"

ancestor_of(Ancestor, Person) ←
    child_of(Person, Ancestor).

Someone is an ancestor of a person if he is the parent of that person.
Alternatively: *Parents are ancestors.*

ancestor_of(Ancestor, Descendant) ←
  child_of(Person, Ancestor),
  ancestor_of(Person, Descendant).

Someone is an ancestor of a descendant if he is the parent of another ancestor of the descendant.

Alternatively: *Parents of ancestors are ancestors*

Reading complete predicates is often too clumsy:

Someone is an ancestor of a descendant, (either) if he is the parent of that descendant, **or** *if he is the parent of another ancestor of the descendant.* (unspeakable)

Alternatively: *Parents* **and** *their ancestors are ancestors.* (too terse)

Informal reading is intuitive but limited to small programs.
⇒ Extend informal reading to read larger programs

Declarative reading of programs

- consider only parts of program at a time
- cover the uninteresting/difficult parts (~~like this~~)
- shortens sentences to be read aloud

Analysis of clauses

Read single clause at a time.

Add remark: *But there may be something else.*

ancestor_of(Ancestor, Person) ←
    child_of(Person, Ancestor).

~~ancestor_of(Ancestor, Descendant)~~ ←
    ~~child_of(Person, Ancestor)~~,
    ~~ancestor_of(Person, Descendant)~~.

Someone is an ancestor of a person if he is the parent of that person. (But there may be other ancestors as well).

Alternatively: *At least parents are ancestors.*

~~ancestor_of(Ancestor, Person)~~ ←
    ~~child_of(Person, Ancestor)~~.
ancestor_of(Ancestor, Descendant) ←
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).

Someone is an ancestor of a descendant if he's the parent of another person being an ancestor of the descendant. But ...

At least parents of ancestors are ancestors.

Erroneous clauses

For error location it is not necessary to see the whole program

ancestor_of_too_general(Ancestor, Person) ←
    child_of_too_general(Ancestor, Person).
~~ancestor_of_too_general(Ancestor, Descendant)~~ ←
    ~~child_of_too_general(Person, Ancestor)~~,
    ~~ancestor_of_too_general(Person, Descendant)~~.

Analysis of the rule body

- goals restrict set of solution
- cover goals to see generalized definitions

father(Father) $\leftarrow$
    male(Father),
    ~~child_of(_Child, Father)~~.

Fathers are at least male.

(But not all males are necessarily fathers)

father_toorestricted(franz) $\leftarrow$
    ~~male(franz)~~,
    ~~child_of(_Child, franz)~~.

Body is irrelevant to see that definition is too restricted.

Searching for errors

If erroneous definition is

1. too general. Use: Analysis of clauses to search too general clause
2. too restricted. Use: Analysis of the rule body

Reading method leads to analgous writing style.

Procedural reading of programs

- special case of the declarative reading
- uncover goals in strict order
- look at variable dependence
  - first occurrence of variable
    variable will always be free
  - further occurrence
    connected to goal/head

1. ancestor_of(Ancestor, Descendant) $\leftarrow$ % $\Longleftarrow$
    ~~child_of(Person, Ancestor)~~,
    ~~ancestor_of(Person, Descendant)~~.
  $\Rightarrow$ Head does not exclude anything.
2. ancestor_of(Ancestor, Descendant) $\leftarrow$
    child_of(Person, Ancestor), % $\Longleftarrow$
    ~~ancestor_of(Person, Descendant)~~.
  $\Rightarrow$ Ancestor can influence child_of/2.
  $\Rightarrow$ Descendant doesn't influence child_of/2.
  $\Rightarrow$ Person will be always free.
3. ancestor_of(Ancestor, Descendant) $\leftarrow$
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant). % $\Longleftarrow$
  $\Rightarrow$ Descendant only influences ancestor_of/2.

Termination

- often considered weak point of Prolog
- nontermination is a property of
  a general purpose programming language
- only simpler computational models guarantee termination
  (datalog, categorical programming languages)
- floundering is also difficult to reason about
- pretext to stop declarative thinking, usage of debuggers etc.
- $\leftarrow$ Goal. terminates if $\leftarrow$ Goal, fail. terminates (and fails)

Idea:

- termination reading special case of procedural reading
- consider simpler predicate
- if simpler predicate terminates (& fails), the original predicate terminates as well

Termination reading

- cover all irrelevant clauses
  - cover all facts
  - non recursive parts

~~append([], Xs, Xs)~~.
append([X|Xs], Ys, [X|Zs]) ←
    append(Xs, Ys, Zs).

- cover variables that are handed through (Ys)

~~append([], Xs, Xs)~~.
append([X|Xs], ~~Ys,~~ [X|Zs]) ←
    append(Xs, ~~Ys,~~ Zs).

- cover head variables (approximation)

~~append([], Xs, Xs)~~.
append([~~X~~ |Xs], ~~Ys,~~ [~~X~~ |Zs]) ←
    append(Xs, ~~Ys,~~ Zs).

Resulting predicate:

```
appendtorso([_X|Xs], [_Z|Zs]) :-
    appendtorso(Xs, Zs).
```

- if appendtorso/2 terminates, append/3 will terminate
- appendtorso/2 never succeeds
- only a safe approximation
  ← append([1|_], _, [2|_]).
  ← appendtorso([1|_], [2|_]).
  appendtoro/2 does not terminate while append/3 does
- **The** misunderstanding of append/3
  rôle of fact append([], Xs, Xs)
  often called "end/termination condition"
  But: append([], Xs, Xs) has no influence on termination!

Reasoning about termination: append3/4

$$\begin{array}{ll}
\text{append3A(As, Bs, Cs, Ds)} \leftarrow & \text{append3B(As, Bs, Cs, Ds)} \leftarrow \\
\quad \text{append(As, Bs, ABs),} & \quad \text{append(As, BCs, Ds),} \\
\quad \text{append(ABs, Cs, Ds).} & \quad \text{append(Bs, Cs, BCs).}
\end{array}$$

Which one terminates for merging and splitting?

Procedural reading of append3A/4

append3A(As, Bs, Cs, Ds) $\leftarrow$
  append(As, Bs, ABs), % $\Longleftarrow$ terminates only if As is known
  ~~append(ABs, Cs, Ds)~~.

Result:
  terminates *only* if As is known (no open list)

$\Rightarrow$ reject append3A/4

- only a part of the predicate was read
  (the second goal was *not* read)
- it was not necessary to imagine Prolog's precise execution
- no "magic" of backtracking, unifying etc.
- no stepping thru with a debugger — a debugger shows irrelevant details (inferences of the second goal)

Procedural reading of append3B/4

append3B(As, Bs, Cs, Ds) $\leftarrow$
    append(As, BCs, Ds), % $\Longleftarrow$ terminates if As or Ds known
    ~~append(Bs, Cs, BCs)~~.
append3B(As, Bs, Cs, Ds) $\leftarrow$
    append(As, BCs, Ds),
    append(Bs, Cs, BCs). % $\Longleftarrow$ if Bs or BCs (=Ds) known

Result:

1. terminates if As and Bs are known (more than merging)
2. terminates if Ds is known (= splitting)

Fair enumeration of infinite sequences

- termination reading is about termination/non-termination only
- in case of non-termination, fair enumeration still possible
- much more complex in general
- order of clauses significant
- e.g. unfair if two independent infinite sequences

  ```
  list_list(Xs, Ys) ←
      length(Xs, _),
      length(Ys, _).
  ```
- explicit reasoning about alternatives (backtracking)
- use *one* simple fair predicate (e.g. *one* length/2) instead
- learn the limits, but don't go to them

Resource consumption

- analytical vs. empirical
- *Do not try to understand precise execution!*
- prefer measuring over tracing
- abstract measures often sufficient

  E.g. inference counting, size of data-structures

  - inference counting

    list_double(Xs, XsXs) $\leftarrow$
        append(Xs, Xs, XsXs).
    $\leftarrow$ length(XsXs, N), list_double(Xs, XsXs).
    When counting, ignore facts (similar to termination reading)

Rename 2nd argument, delay unification

list_double(Xs, XsXs) $\leftarrow$
    append(Xs, Ys, XsXs),
    Xs = Ys.

$\leftarrow$ list_double(Xs, XsXs).

Requires N and not $N/2$ inferences (+ unification costs)

– size of data structures

(If everything else is the same)

size of data structures approx. proportional to execution speed

Reading of definite clause grammars

Comma is read differently:

nounphrase $\longrightarrow$	% A noun phrase consists of
determiner,	% a determiner **followed by**
noun,	% a noun **followed by**
optrel.	% an optional relative clause.

Declarative reading of grammars

Context free grammars are the declarative formalism *per se* but still it is helpful to consider generalizations:

nounphrase $\longrightarrow$	% A noun phrase (at least)
determiner,	% starts with a determiner
~~noun~~,	% —
optrel.	% ends with an optional relative clause

Procedural reading of grammars

Take implicit argument (list) into account

```
seq([]) --->
    [].
seq([X|Xs]) --->
    [X],
    seq(Xs).

seq3(Xs, Ys, Zs) --->
    seq(Xs),
    seq(Ys),
    seq(Zs).
```

```
append3(As, Bs, Cs, Ds) <-
    phrase(seq3(As, Bs, Cs), Ds).
```

splitting and joining works

Part III
Course implementation

- 2nd year one semester course
  2hrs/week (students claim: 9×5hrs work)
- nine weeks (example groups) about 70 small assignments

Course contents

- Basic elements (facts, queries, rules)
- Declarative reading (first only with datalog)
- Procedural reading (—””—)
- Termination (—””—)
- Terms
- Term arithmetic
- Lists
- Grammars

- List differences (*after grammars*)
- State & general differences (make/next/done)
- Limits of pure Prolog (unfairness etc.)
- Meta-logical & control

  most important part: error/1 (terminate execution with an error message)

  (nonvar/1, var/1, error/1, cut)
- Negation
- Term analysis
- Arithmetic

Topics not covered

(*): covered in an advanced course (3hrs)

1. setof(Template, Goal, Solutions) (*)
   "answer substitutions" vs. "list of solutions" confusing — quantification tricky
2. meta interpreters (*) — *program = data* too confusing
   instead use pure meta interpreters "in disguise" (e.g. regular expressions)
3. meta call (*)
4. explicit disjunction (*) — meaning of alternative clauses must be understood first
5. if then else (*) — leads to defaulty programming style
   if used, restrict condition to var/nonvar and arithmetical comparison
6. data base manipulation (*) — difficult to test — if used, focus on setof/3-like usage
7. advanced control (*) — reasoning about floundering difficult
8. constraints (*)
9. extra logical predicates
10. debuggers, tracers — reason for heavy usage of cuts

GUPU Programming Environment

Gesprächs**u**nterstützende **P**rogrammierübungs**u**mgebung
conversation supporting programming course environment
Guided tour: http://www.complang.tuwien.ac.at/ulrich/gupu

- specialized for Prolog courses
- uses a subset of Prolog
- focuses on clean part of Prolog
  i.e. no side effects allowed
- side effect free interaction
- comfortable querying and testing
- Only two (nonoverlapping) windows:
  - example texts to be edited
  - help texts with simple mark up links

  (no window to execute or test)

```
## 1. Beispiel ##############################################
# Stellen Sie eine Frage (mit < ).

# Beachten Sie bitte den Unterschied zwischen einer
# Anfrage wie z.B.
:- ocean(Ozean).
# und einer < Frage. Siehe Anhang A. Verwenden Sie die
# < Fragen nur, wenn Sie Hilfe brauchen. Siehe auch
# \Hinweis{Tastatur}.

## 2. Beispiel ##############################################
# Schreiben Sie eine kleine Datenbasis (mit zumindest
# 10 Personen), die familiäre Beziehungen beschreibt:
# (In den folgenden Beispielen werden einige komplexere
# Verwandtschaftsbeziehungen definiert, formulieren Sie
# daher bitte eine Datenbasis, die komplex genug ist.

# -- Hier können Sie die Funktionstasten zum raschen
# Kopieren von Funktoren verwenden. Siehe Anhang B. --

kind_von(joseph_I, leopold_I).
kind_von(karl_VI, leopold_I).
kind_von(maria_theresia, karl_VI).
kind_von(joseph_II, maria_theresia).
kind_von(joseph_II, franz_I).
kind_von(leopold_II, maria_theresia).
kind_von(leopold_II, franz_I).
kind_von(marie_antoinette, maria_theresia).
kind_von(franz_II, leopold_II).

:- kind_von(Kind, Elternteil).
:- männlich(Mann).
! ! Prädikat :männlich/1: nicht oder in nicht geladenem Beisp\
iel definiert. \Hinweis{laden}
```

```
Bitte lesen Sie zuerst die Beschreibung dieser
Programmierumgebung in Anhang A und B!
Auf dieser Seite können Sie allgemeine Hinweise
lesen. Um einen Hinweis zu lesen, mit dem
Cursor vor einen Hinweis und DO drücken.
          \Hinweis{init9495last} (Vom WS)

          \Hinweis{Tastatur}        (Allgemein)
          \Hinweis{Reservierung}    (Allgemein)
          \Hinweis{Übungsmodus}     (Allgemein)
          \Hinweis{Maschinenwahl}
          \Hinweis{ÜberlasteteMaschinen}
          \Hinweis{Konsistenzprüfung}
          \Hinweis{Bewertungsmodus}
          \Hinweis{KompakteListen}
          \Hinweis{Suffix}
ad Bsp.26 \Hinweis{Zahlenpaare}
ad Bsp.29 \Hinweis{Datenstrukturdefinition}
          \Hinweis{AufbauendeLVAs} (SommerS.95)
          \Hinweis{Wozu_Prolog}
ad Bsp.28 \Hinweis{appendnachsuffix}
ad Bsp.53 \Hinweis{Instanzierungsmuster} Erkl.
ad Bsp.57 \Hinweis{Frosch} Die ganze Geschichte
ad Bsp.58 \Hinweis{Variablen_in_DCGs}
ad Bsp.62 \Hinweis{Mögliche_Instanzierungen}
ad Bsp.67 \Hinweis{Diagonalen}
          \Hinweis{PrologAllgemein}

Abgabetermine sind nun mittwochs 24h00.
1. Abgabetermin ist Mittwoch 22. März.
```

```
-----          n599 server 100% 20:18 Freie Zeit xterm  (GUPU)   --%%-Emacs: init.hlp         (Hinweise)--All---
```

Interaction

1. edit text
2. press $\boxed{\text{DO}}$ to save, compile, test
3. comments (from system or lecturer) are written back *into* text

child_of(karl_VI, leopold_I).
child_of(maria theresia, karl_VI).
! child_of(maria$\ll * \gg$theresia, karl_VI).
! Argumentliste eines Funktors unterbrochen, ...
child_of(joseph_II, maria_theresia).

$\leftarrow$ append(Xs, Xs, Xs).
< @@@ % Xs = [].
< @@@ ! Ausführung dauert zu lang, Antwort unvollständig
< Why the loop here?
$*>$ **Compare it to $\leftarrow$ append(Xs, Xs, Zs), Xs = Zs.**

Program text, assertions

child_of(karl_VI, leopold_I).
child_of(maria_theresia, karl_VI).
child_of(joseph_II, maria_theresia).
child_of(joseph_II, franz_I).
child_of(leopold_II, maria_theresia).
child_of(marie_antoinette, maria_theresia).
$\leftarrow$ child_of(Child, Parent).
$\not\leftarrow$ child_of(joseph_II, friedrich_II).

Assertions

- $\leftarrow$ Goal. should succeed
- $\not\leftarrow$ NGoal. should not succeed (`:/-`), avoids talking about negation
- tested upon saving
- timeouts for "infinite loops"
- immediate feedback
- supports a more specification oriented programming method:
  1. formulate test cases (= specification)
  2. write predicate
  3. testing is now "for free"

Querying predicates

Two rôles of $\leftarrow$ Goal.

- assertion (tested upon saving)
- query

Answer substitutions

child_of(karl_VI, leopold_I).
child_of(maria_theresia, karl_VI).
child_of(joseph_II, maria_theresia).
child_of(joseph_II, franz_I).
child_of(leopold_II, maria_theresia).
child_of(marie_antoinette, maria_theresia).
$\leftarrow$ child_of(Child, Parent).
@@@ % Parent = leopold_I, Child = karl_VI.
@@@ % Parent = karl_VI, Child = maria_theresia.
@@@ % Parent = maria_theresia, Child = joseph_II.
@@@ % Parent = franz_I, Child = joseph_II.
@@@ % Parent = maria_theresia, Child = leopold_II.
@@@ ? Weitere Lösungen mit SPACE
$\not\leftarrow$ child_of(joseph_II, friedrich_II).

Answer substitutions cont.

- displayed in chunks
- locates most backtracking problems
- infinite sequences can be inspected
- redundant answer substitutions labeled
- answer substitutions inserted **into** program text
- easy to (re-)use answer substitutions for new goals
- timeouts

Example domains

1. The family database

   – recursion maybe better with recursive terms

   – infinite loops in the first week (timeouts)

   – doesn't compute something "real"

   + motivation, identification with own db (= often own family)

   + mapping Prolog to English much simpler if domain well known (e.g. uncle John ...)

   + clarify notions taken for granted (e.g., siblings)

   + data incompleteness

   + various degrees of inconsistency, integrity constraints

   + recursion not that difficult with procedural reading technique

2. Maps
3. Stories Mapping small fairy tales into Prolog.
4. (simplified) grammars of programming languages
5. RNA-analysis (along D.B.Searls NACLP89)
   + very pure
   + backtracking mechanism, efficiency issues
   + execution imposssible to understand step-by-step
     no procedural cheating possible
   + constraining variables
   + reordering parsing
6. Analyzing larger text
   E.g. extracting the words used etc.