Teaching beginners Prolog
How to teach Prolog

Ulrich Neumerkel
Technische Universität Wien
Institut für Computersprachen
A-1040 Wien, Austria
ulrich@mips.complang.tuwien.ac.at

Abstract

We discuss in this tutorial the problems that arise in introductory Prolog courses and present several improvements. The major improvement concerns the way how Prolog programs can be read and understood. The traditional approach in teaching Prolog focuses first on two separate issues: the meaning of logic formulae and Prolog's execution mechanism. While both are intimately related to each other, this relation is rarely exploited. On the one hand reading logic formulae is not explained in depth but on the other hand an overly detailed account of Prolog's execution mechanism is given which is a burden to comprehending Prolog.

A practical way of reading Prolog programs is presented that focuses on the meaning of the programs and avoids execution traces and proof trees altogether. This reading scheme is then extended to cope with the more procedural aspects of Prolog like termination and resource consumption. The reading scheme allows a programmer to reason in an efficient manner while still avoiding reference to superfluous details of the computation.

Overview

This tutorial discusses various topics specific to introductory Prolog courses. The order of topics in this tutorial does not reflect the order that should be used in the classroom.

Syllabus - Syntax - Reading of Programs - Testing and Executing - Understanding differences - example domains.

Contents

1 Syllabus

1.1 Goals

There are two apparently conflicting goals of a Prolog course when designing the syllabus:

- Learn Prolog to eventually use it. (Training)
- Learn Prolog for the sake of education, edification, and its inherent values. (Teaching)

The first goal suggests a more project oriented syllabus with a few very large projects. The second goal suggests a more concept oriented syllabus, viewing Prolog rather as the Latin of programming languages.

From my experience an approach with few large projects has many disadvantages. Getting acquainted to the principal ideas takes a lot of time and should be done prior to starting the project anyway. Otherwise students simply stumble from error to error. Before beginning any real projects at least the following tasks should have been accomplished.

- Basic reading skills for understanding Prolog programs.
- Avoiding common mistakes, develop a programming (rather coding) style

My current course features nine weeks (example groups) totaling about 70 mostly smaller assignments.

1.2 Prerequisites

1.2.1 Programming skills

Student's programming skills are seldom what the structured programming movement meant to achieve. Most students associate with notions like invariants, pre- and post-conditions just useless verbiage set in comments of a procedural program. In fact the formal techniques of structured programming are taught too often in a class different from introductory programming or any practical programming class.

While students remember from such classes that verifying programs is still an open area — the fact that these techniques can be applied *now* to even mundane C-programs in the form of the assert(_) construct, is unknown to most students. A small questionnaire for devoted C-programmers about his coding style may help to clarify what is understood by the activity of programming, coding beyond general buzzwords:

- 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?

Beginners have lots of problems understanding Prolog because they were never learned the basic ideas of structured programming. It is not to blame others for students' difficulties with Prolog, but to justify that there is very few prior knowledge that an introductory course can be based on.

1.2.2 Language skills

So the only prior knowledge one can expect to build on are a students language skills. The reading techniques presented later on were developed to exploit this facts. So maybe it is not surprising that students from other universities who take my course occasionally (and optionally) and who are almost "illiterate" to computer usage and programming perform better than the average computer science student.

1.3 Course contents

Over the years the introductory course has been more and more focused to the essential parts leaving out almost all a typical textbook covers. Everything a manual can just cover adequately has been removed over the years.

In particular the following topics —in the order of possible candidates for inclusion first— are not covered:

1. meta interpreters: only some interpreters for simple tasks (regular expressions) appear in assignments. The notion interpreter is avoided for them for obvious reasons. General meta interpreters are avoided currently, since focus is on good programming techniques (i.e., non defaulty data structures) and not on tricky coding techniques (e.g., how to implement cut without an ancestor cut).
2. set predicates, e.g. setof(Template, Goal, Solutions) with Goal being a *known* goal
3. meta call
4. if then else, since it leads to a defaulty programming style
5. all extra logical predicates
6. debuggers, tracers

The complete lists of topics currently covered is: Basic elements (facts, queries, rules), Declarative reading, Procedural reading, Termination, Terms, Term arithmetic, Lists, Grammars, List differences, State & general differences (make/next/done), Limits of pure Prolog, Meta-logical & control (nonvar/1, var/1, error messages, cut), Negation, Term analysis, Arithmetic.

2 Syntax matters a lot

Computer programs are written primarily to be read by human beings, in particular the student who writes the program will profit a lot from a clean coding style.

2.1 Flaws in Prolog's syntax

The simplicity of Prolog's syntax appears first as very appealing. But many beginners struggle with Prolog's syntax because it is so simple. Small typos can have puzzling effects and lead to frustrating experiences. A single comma in place of a period may cause a program to change its meaning completely. In the following fragment, a goal for father_of/2 will always fail unless the fact male(john) was defined twice. Note that even a type system or the detection of accidentally void variables cannot help to find all such errors. In the case below, both fail to detect the error.

```
father_of(Father, Child) :-
   child_of(Child, Father),
   male(Father), % !

male(john).
...
```

To fluent Prolog programmers such errors may appear as rather minor, seldom encountered, and easy-to-fix. However, for beginners the situation is different. Virtually everybody encounters this problem and remains trapped in it for a while. The statistics collected over the last years show that from 488 participants 411 have made such an error at least once ; so 84 percent are stuck with such a seemingly minor problem. On average this error occurred more than 5 times, often enough to keep you busy for hours chasing such "minor" typos. Such typos often lead to hectical "bug fixing" in completely unrelated parts of a program, since they are almost invisible for the unexperienced. Even worse, such typos may cause the first encounter with so called de-buggers or execution tracers.

To overcome these problems we have two choices:

1. Redesign Prolog's syntax.
2. Take a subset of Prolog's existing syntax.

Prolog's syntax has been redesigned in Prolog II where facts cannot be confused with a rule's last goal since in Prolog II a fact is written as a rule with an empty body. Unfortunately this idea has not caught on. Also smaller fixes to Prolog's syntax are not possible. Using the notation male(john) :- . for facts (by defining :- as a postfix operator) is impossible. The Prolog standard states: *There shall not be an infix and a postfix operator with the same name.* So :- cannot serve for both a rule atom and a postfix operator to denote facts. Second, :- is already used as a prefix operator. Even if we would relax the standard's restriction, read/1 cannot distinguish the directive :- male(john). from male(john) :- .

Given the fact that there is no chance to change Prolog's syntax, the only realistic choice we have is to use a subset of it. Spacing and indentation must be taken into account. The following restrictions are used in a specialized programming environment for beginners[1].

1. Each head each goal goes into a single line.
2. Goals are indented. Heads are not indented.
3. Only comma can separate different goals.
4. Different predicates are separated by blank lines.

Given these restrictions all such problems can be indicated precisely and delays caused by such typos are kept to the minimum.

2.2 Names of predicates

The rôle of choosing appropriate names for predicates cannot be overestimated. It is helpful to have in the beginning separate assignments that consist simply of finding good predicate names.

Common misnomers are names that

- describe the action performed for certain patterns of usage (prescriptive names) but other patterns are possible (or thinkable).
- leave the argument order open.
- pretend a relation that is too general or too specific.

[1] Actually many more restrictions are present. The programming environment is described in detail in http://www.complang.tuwien.ac.at/ulrich/gupu

2.2.1 Descriptive instead of prescriptive names

Prescriptive names suggest a certain way to execute a predicate instead of describing the relation the predicate implements (or approximates). Unfortunately, many idiomatic Prolog predicates have prescriptive names that suggest a certain execution order (append/3, reverse/2). Sometimes past participles help (reversed).

Using a noun instead of an "acting" verb (e.g. concatenation/3, reversal/2) sounds often very declaratively. But nouns are often confusing since they leave open the precise argument order.

2.2.2 Disambiguate argument positions

Even predicate names that do not suggest a certain execution manner may be deceptive. Consider the following two binary relations: "length of a list", and "child of a person". The relation length/2 is a function (each list has one length). So the result goes in the second argument: length(Xs, Length). But what are the arguments of the predicate child/2? Is the child the first or the second argument? Even textbooks do not agree on the argument order — most prefer the child to be the first argument: child(Child, Parent).

The following guidelines help to find an unambiguous name for a predicate.

1. Start with a name that describes only the intended types of the arguments.

   type1_type2_type_3_type4(Arg1, Arg2, Arg3, Arg4)

   E.g., to find a name for the relation "child of a person" start with person_person/2.

   In most cases such a name is certainly too general, but this starting point is essential because it helps to avoid thinking of arguments as inputs and outputs.

2. If the name describes a relation that is too large, try to refine the name by using more specific terms. E.g., child_person/2, list_reversedlist/2.

3. Emphasize the relation *between* arguments.

   - Use shortcuts like prepositions.
     E.g.: child_of/2
   - Use past participles alone.
     E.g.: list_reversed/2

For "length of a list" we would refine as follows: 1) number_list, 2) length_list. Equally well we can refine 1) list_number, 2) list_length.

At the first glance it seems that this scheme yields very verbose predicate names, e.g., list_length/2 instead of the common length/2, list_reversed/2 instead of reverse/2. But often the traditional names are too generic for the actual relation they are defined for. E.g., length/2 is only defined for list and not for e.g. atoms in contrast to LISP-systems where length is generic over many predefined data structures.

The naming scheme does not suggest any specific argument order. Very detailed guidelines to find a specific argument order are described by Richard A. O'Keefe in The Craft of Prolog. However, these guidelines mostly cover predicates where we can distinguish between inputs and outputs. Therefore, these guidelines are not suitable for beginners but should be used only at an advanced level.

For predicates with large arities this scheme leads to very verbose names. In such cases names of the less important arguments can be omitted provided they follow the more important arguments: country(Country, Region, Population, ...).

2.3 Variable Names

Since Prolog has no type system it is particularly important to try to document a variable's type with the variable's name similar to the "Hungarian naming conventions" in other programming languages. In particular plural forms like Xs for lists are very helpful for beginners.

Void variables. Void variables have to start with _. In the head voids must have a name to document an arguments meaning. In the case below _Xs instead of _ is definitely preferable, just to underline, that the tail should be a list.

member(X,[X|_]).

3 Reading of programs

3.1 Informal reading

Learning to read a Prolog program in English helps to focus the student's attention to the meaning instead of operational details.

Many constructs of Prolog clauses translate directly into certain words. Shared variables are translated with various constructs like "of", "both", "they all", a relative pronoun, or a relative clause (in case the variable appears in different goals). Informal readings must be undertaken with some care, due to the ambiguities of the language.

```
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*

Special care is required to distinguish between conjunctive "and"s and disjunctive "and"s meaning "or". Often conjunctive "and"s can be described informally with relative phrases. Combining alternate clauses into a single sentence is often much more tedious. The resulting sentences are often not easy to understand. In the case above we could say either *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.*
or more elegantly: *Parents* **and** *their ancestors are ancestors.*

These phrases show the limits of reading a predicate aloud: The first sentence is extremely long —28 words— for expressing an idea that fits into 15 words (excluding punctuation) in Prolog. The second more elegant version, is almost too terse to be understandable. In particular the terse sentence is very difficult to analyze for errors. Also the plural is irritating because it is not clear whether this definition is about a single parent or a pair of parents.

To overcome this problem a very simple reading technique is presented that does not translate the whole predicate at once into English.

3.2 Declarative reading of programs

The basic idea is to consider only parts of a predicate. The remainder of the predicate will be neglected (visualized by $\equiv$). In this manner we do not need to understand the whole predicate in one fell swoop. Incomprehensible sentences of the length of paragraphs are thus avoided.

3.2.1 Analysis of clauses

A predicate consists of a sequence (or set) of clauses. Each clause describes a part of the solutions for the predicate. We will confine ourselves to read a single clause at a time. To remind ourselves, that there is more to the definition of the predicate, we add some remark like: *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).
At least parents are ancestors.

When covering the second clause we get:

```
~~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 is the parent of another person being an ancestor of the descendant. (But ...)
At least parents of ancestors are ancestors.

Erroneous clauses. In ancestor_of_too_general/2 a typo has crept into the first clause. The arguments have been exchanged accidentally. The rule now reads: *Children are ancestors of their parents.* Since this clause already describes solutions that are not part of the intended meaning of what an ancestor should be, we can identify this clause as erroneous — without considering the whole predicate.

```
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)~~.
```

3.2.2 Analysis of the rule body

Each goal in a rule restricts the set of solution described by the rule. By covering goals, we consider a generalized definition of the rule. While the original definition of father reads *Male parents are fathers*, we have now a generalized version.

```
father(Father) :-
    male(Father),
    ~~child_of(_Child, Father)~~.
```

Fathers are at least male. (But not all males are necessarily fathers)

If such a generalized clause is already too restricted, we have found an erroneous definition. We can see that the definition below is incorrect, without even considering any goal. Already the head contains the error.

```
father_toorestricted(franz) :-
    ~~male(franz)~~,
    ~~child_of(_Child, franz)~~.
```

3.2.3 Searching for errors

In case of an erroneous definition of a predicate we can now use the following strategy:

1. If the erroneous definition is too general, then one or more clauses involved are defined too general. Use: Analysis of clauses.
2. If the erroneous definition is too restricted, a single clause is too restricted. Use: Analysis of the rule body.

After getting used to this way of looking at a program further variations, e.g. combining both ways can be used.

3.3 Procedural reading of programs

The procedural reading technique is just a special case of the declarative reading technique. Instead of covering just some goals we start by covering all goals. Then, step-by-step, we uncover one goal after the other. In this manner the dependence of variables can be seen easily. E.g., in step two we see that the goal child_of will always been called with the first argument free. In step three we see the first usage of the variable Descendant.

The connection of variables is important for understanding termination of predicates. In this case we can see, e.g. that the variable Descendant will have no influence on termination, while Ancestor may prevent infinite computations (by causing child_of to fail).

Step 1

```
ancestor_of(Ancestor, Descendant) :-    % ⟸
    ~~child_of(Person, Ancestor)~~,
    ~~ancestor_of(Person, Descendant)~~.
```

Step 2

```
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),         % ⟸
    ~~ancestor_of(Person, Descendant)~~.
```

Step 3

```
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).    % ⟸
```

3.4 Termination

When reasoning about the termination of a predicate it is best to cover all irrelevant clauses. As a first approximation we can cover all non recursive parts. To understand termination of append/3, we can ignore its fact altogether.

```
~~append([], Xs, Xs)~~.
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs, Ys, Zs).
```

Since the second argument is just handed through, we can ignore it as well:

```
~~append([], Xs, Xs)~~.
append([X|Xs], ~~Ys,~~ [X|Zs]) :-
    append(Xs, ~~Ys,~~ Zs).
```

We can go even a step further and ignore the variable X. (By this we will miss some rare cases were append would still terminate in the case of incomplete lists like :- append([1|_],_,[2|_]). Our analysis is therefore only a safe approximation.)

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

When this somewhat distorted predicate will terminate (and fail) also the original definition of append/3 will terminate (and maybe even succeed). Our new predicate terminates if the first or the last argument contains a finite list (to be precise not even a proper list is required, a dotted pair will do as well).

Notice that this reading technique avoids some common misunderstandings concerning the rôle of the fact append([], Xs, Xs). Often the fact is called the "end/termination condition". But, as we see above, append([], Xs, Xs) is *not* responsible for the termination of append/3 at all.

A rather tricky example. The following well known example helps to illustrate the power of the presented procedural reading technique. Most real examples are much simpler and in this particular case I would rather recommend to use definite clause grammars, since they immediately lead to the correct solution.

The following predicates try to define the concatenation of three lists. It is certainly desirable that such a predicate terminates whenever the set of solutions is finite. So we would like to use this predicate reliably to split a given list into three parts and to join three given lists.

Which of the following definitions terminates in both cases?

```
append3A(As,Bs,Cs,Ds) :-      append3B(As,Bs,Cs,Ds) :-
    append(As,Bs,ABs),            append(As,BCs,Ds),
    append(ABs,Cs,Ds).            append(Bs,Cs,BCs).
```

append3A.

```
append3A(As,Bs,Cs,Ds) :-
    append(As,Bs,ABs), % ⟸
    ~~append(ABs,Cs,Ds)~~.
```

Our procedural reading technique tells us immediately that only As and Bs are connected with the head. The third argument ABs will always be free. We have seen before that Bs is of no importance for termination properties, so it is evident that this definition *only* terminates if As is a given list. This definition is therefore unsuitable for splitting a list. We can immediately reject this unstable definition.

Notice that it was not necessary to consider the second goal at all! Even better, it was not necessary to imagine the “magic” of Prolog’s execution mechanism at all. Not a single thought was necessary about backtracking, unifying, or handing terms through arguments.

Exchanging the goals would not help, since the second goal is also connected to the head only with a single variable. What we really need is goal that connects to the head with more variables.

append3B.

```
append3B(As,Bs,Cs,Ds) :-
    append(As,BCs,Ds), % ⟸
    ~~append(Bs,Cs,BCs)~~.
```

The first goal will terminate if As or Ds is a known list. BCs will be a known list only if Ds is known.

```
append3B(As,Bs,Cs,Ds) :-
    append(As,BCs,Ds),
    append(Bs,Cs,BCs). % ⟸
```

The last goal terminates if Bs or BCs (and therefore Ds is known). To summarize, this predicate will terminate if As and Bs together are known, or if Ds is known.

3.4.1 Fair enumeration of infinite sequences

Reasoning about the fair enumeration of infinite sequences is much more complex than the techniques presented so far. It is very seldom that a predicate which does not terminate is used as a generator for an infinite set of solutions. In such a case it is often preferable to use a simple to understand fair generator like length/2 and then connect this part to the real generator.

3.5 Resource consumption

Precise estimations of resource consumption require to understand the actual execution order, but exactly this should be avoided as often as possible. Frequently it is sufficient to consider the size of intermediate data structures or the number of intermediate solutions. To see the number of intermediate solutions it is often helpful to generalize a predicate.

```
list_double(Xs, XsXs) :-
    append(Xs, Xs, XsXs).

:- list_double(Xs, XsXs).
```

How many inferences are required to compute list_double(Xs, XsXs) with XsXs a list of n elements?

We have seen in the previous example that the second argument in append/3 has not had any impact in the recursive rule. It is therefore safe to delay the sharing of the first two arguments:

```
list_double(Xs, XsXs) ←
    append(Xs, Ys, XsXs),
    Xs = Ys.

:- list_double(Xs, XsXs).
```

Now it is evident that halving a list requires n inferences and not (as often assumed) $n/2$ inferences. Again we consider the definition of append/3. This time we are interested in counting inferences, so it is again safe to ignore the first fact. The fact that the first and second argument are shared does not have an impact on the number of inferences, since the second argument has no effect in the rule in append/3.

3.6 Reading of definite clause grammars

Similar to reading ordinary Prolog clauses grammar rules can be read. Now the comma is read differently.

```
nounphrase ⟶              % A noun phrase consists of
    determiner,           % a determiner followed by
    noun,                 % a noun followed by
    optrel.               % an optional relative clause.
```

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

```
nounphrase ⟶              % A noun phrase (at least)
    determiner,           % starts with a determiner
    ~~noun~~,             % —
    optrel.               % and optionally ends with a relative clause.
```

Procedural reading. Reasoning about termination is slightly different to ordinary clauses. In addition to the arguments the implicit list must be taken into account.

```
seq([]) ⟶              seq3(Xs, Ys, Zs) ⟶        append3(As, Bs, Cs, Ds) :-
    [].                    seq(Xs),                  phrase(seq3(As, Bs, Cs), Ds).
seq([X|Xs]) ⟶              seq(Ys),
    [X],                   seq(Zs).
    seq(Xs).
```

seq//1 terminates if its argument is known or if the implicit list is known. seq3//3 terminates for given Xs, Ys, and Zs or when the implicit list is known. The predicate append3/4 can therefore be used both for splitting and joining.

4 Testing and executing programs

It is vital when learning Prolog to focus on the clean parts of the language. All of Prolog's add ons which are needed to integrate Prolog programs in an application must be left out completely. It is too much to ask of a beginner to understand both the meaning of a program and the precise way how this program is executed.

This part discusses briefly a programming environment (GUPU) that is used for the Prolog programming courses at TU-Wien. It is shown how GUPU provides side effect free interactions with a Prolog program.

GUPU's screen (http://www.complang.tuwien.ac.at/ulrich/gupu or the appendix) consists of two windows. The left contains the examples to be solved. The right contains help (read-only) texts with links to other texts. All interaction is performed through these two windows.

GUPU's metaphor is very simple: The left window is like a paper sheet containing at first only example statements. The student adds more text and saves it from time to time by pressing a button. Comments from the system or lecturer are written back into the text.

A simple familiy database will serve as an example.

```
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(marie_antoinette, maria_theresia).
:- kind_von(Kind, Elternteil).
:/- kind_von(joseph_II, friedrich_II).
```

The last lines above are goals which are inserted like ordinary program text. Such goals are called assertions (like assertions in procedural languages). They are part of the program. Every time an example is saved such goals are all tested immediately. (In our case the first assertion is true, since there are some children around, the second assertion states that the goal should fail). If a goal does not behave as specified an error message is inserted into the text. Typical example/problem statements often involve phrases like ... P is true for ..., is false for ... . Such parts can be formulated directly as goals in the program text.

A predicate can thus be specified before being implemented. In contrast, when using a toplevel shell it does not make sense at all to enter queries about predicates that do not yet exist. When using a toplevel shell a separate pass for testing programs is needed. Too often programs are written first and then test cases are conceived - if at all.

Goals serve two purposes. First, they are tested after each compilation serving their role as assertions — and in our case the goal was in fact true. Second, when clicking on the colon of the goal ...

```
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(marie_antoinette, maria_theresia).
:- kind_von(Kind, Elternteil).
@@@ % Elternteil = leopold_I, Kind = karl_VI.
@@@ % Elternteil = karl_VI, Kind = maria_theresia.
```

```
@@@ % Elternteil = maria_theresia, Kind = joseph_II.
@@@ % Elternteil = franz_I, Kind = joseph_II.
@@@ % Elternteil = maria_theresia, Kind = leopold_II.
@@@ ?  Weitere Lösungen mit SPACE
:/- kind_von(joseph_II, friedrich_II).
```

... they are used to query predicates.

Answer substitutions are displayed in chunks of five. In this manner also long sequences of solutions can be inspected with ease. Redundant answer substitutions are labeled separately. The traditional toplevel shell requires one to type two keys ; RETURN for each solution which is somewhat tedious and causes most students to stick with the first solution. But many errors in predicates show up only on backtracking. These errors are much more difficult to find. Thus seeing several solutions helps to detect errors. Additionally, GUPU enables one to inspect also infinitely long sequences.

Goals that do not terminate within a certain amount of time are reported as errors. With the help of time-outs also infinite computations give almost immediate feed-back to the student.

Absence of a debugger.

5 Understanding differences

5.1 Misleading name

The first avoidable obstacle when understanding the notion "difference list" is the name itself. The misleading name "difference list" should be avoided. There is not a single list involved, as the name suggests, but there are two lists. And the quintessential word is not "list" but "difference". A frequent misconception when being exposed to "difference lists" is that these are something different from ordinary lists. The following alternate names are preferable: list difference, difference of lists or even — albeit again difficult to understand — differential list (in analogy to the differential calculus).

5.2 Differences too early in syllabus

The second mistake is to introduce differences too early in a course. It is tempting to present list differences first and only then definite clause grammars, but beginners are much more comfortable with grammar rules. Grammar rules are a very compact formalism and less error-prone than differences. List differences require a very rigid and tedious coding style. In particular variables must be named in a consistent manner, e.g. using the naming convention that numbers variables. Otherwise a program becomes virtually unreadable and in the case of beginners almost always wrong.

The power of grammar rules often amazes students. After having written rather interesting grammars, this gives a nice hook on which to hang the implementation techniques required for grammar rules and their awkward verbosity of list difference programs.

Learning grammar rules before understanding list differences also reflects the historical development more accurately. Context free grammars were well known long before Prolog was conceived. Prolog was born *after* it became clear how to encode grammar rules in logic in an efficient manner.

5.3 Differences as incomplete data structures

Often list differences are presented as a technique using incomplete data structures and "holes". This is a very procedural motivation which is not even accurate for commonly used execution patterns. When analyzing a given list of characters with a grammar, not a single hole appears: all lists are ground. Motivating differences with ground lists helps to visualize the notion of differences.

5.4 Differences are not specific to lists

A simple assignment to underline that differences do not depend on lists uses differences of very simple terms. E.g., the sum of nodes in a tree. More complex but somewhat contrived assignments are datalog grammars.

6 Example domains

6.1 The family database

There are some objections against using the family database as the very first example: a) recursion should rather be motivated along with recursive terms b) (possible) infinite loops appear already in the first week, giving room for early prejudice about Prolog's purported inefficiency c) some students complain initially about data base examples that they cannot see how Prolog can be used to compute something "real" like in other programming languages.

On the other hand this domain has too many advantages to be not the first example:

- Motivational advantages, because most students prefer to type in their own family. A domain where they are certainly experts.
- The relation of Prolog clauses to English sentences is much simpler (e.g. uncle John ...) if the domain is well known in advance.
- Notions commonly taken for granted in conversations (e.g., siblings, cousins) have to be defined precisely. Since there are different notions e.g. of the degree of a relationship (legal, genealogical, sacral) clarifications of their meaning can go hand-in-hand with the coding of Prolog rules.
- Problems of incompleteness can be discussed in a natural setting.
- Problems of various degrees of inconsistency of information can be discussed (e.g. illegtim, illegal, biologically impossible, chronologically impossible)

6.2 Maps

Databases like the CIA-World fact book (http://www.ic.gov/94fact/fb94toc/fb94toc.html).

6.3 Stories

Mapping small stories or fairy tales into Prolog, so that questions can be asked.

6.4 Grammars of programming languages

6.5 RNA-analysis

Complex but still pure examples of DCGs are used based on David B. Searls' article *Investigating the Linguistic of DNA with DCGs*, NACLP89. While using native DCGs to analyze RNAs is not very efficient, searching for certain structures (e.g. clover leaves) is still feasible. This example while still 100% pure helps to understand the weak points of Prolog's backtracking mechanism. Also some strategies to improve Prolog's backtracking can be outlined:

- constraining variables as early as possible (e.g. constraining a list to at least three elements)
- reordering of the parsing process putting easily determinable sequences first

6.6 Analyzing larger text

E.g. extracting the words used etc.

A GUPU's screen

```
## 1. Beispiel #############################################  |Bitte lesen Sie zuerst die Beschreibung dieser
# Stellen Sie eine Frage (mit < ).                            |Programmierumgebung in Anhang A und B!
:- student_vorläufigenote(S,2).                              |Auf dieser Seite können Sie allgemeine Hinweise
# Beachten Sie bitte den Unterschied zwischen einer           |lesen. Um einen Hinweis zu lesen, mit dem
# Anfrage wie z.B.                                            |Cursor vor einen Hinweis und DO drücken.
:- ocean(Ozean).                                              |          \hinweis{init9495last} (Vom WS)
# und einer < Frage. Siehe Anhang A. Verwenden Sie die        |
# < Fragen nur, wenn Sie Hilfe brauchen. Siehe auch           |          \hinweis{Tastatur}        (Allgemein)
# \hinweis{Tastatur}.                                         |          \hinweis{Reservierung}    (Allgemein)
                                                              |          \hinweis{Übungsmodus}     (Allgemein)
## 2. Beispiel #############################################  |          \hinweis{Maschinenwahl}
# Schreiben Sie eine kleine Datenbasis (mit zumindest         |          \hinweis{ÜberlasteteMaschinen}
# 10 Personen), die familiäre Beziehungen beschreibt:         |          \hinweis{Konsistenzprüfung}
# (In den folgenden Beispielen werden einige komplexere       |          \hinweis{Bewertungsmodus}
# Verwandtschaftsbeziehungen definiert, formulieren Sie       |          \hinweis{KompakteListen}
# daher bitte eine Datenbasis, die komplex genug ist.         |          \hinweis{Suffix}
                                                              |ad Bsp.26 \hinweis{Zahlenpaare}
# -- Hier können Sie die Funktionstasten zum raschen          |ad Bsp.29 \hinweis{Datenstrukturdefinition}
# Kopieren von Funktoren verwenden. Siehe Anhang B. --        |          \hinweis{AufbauendeLVAs} (SommerS.95)
                                                              |          \hinweis{Wozu_Prolog}
kind_von(joseph_I, leopold_I).                                |ad Bsp.28 \hinweis{appendnachsuffix}
kind_von(karl_VI, leopold_I).                                 |ad Bsp.53 \hinweis{Instanzierungsmuster} Erkl.
kind_von(maria_theresia, karl_VI).                            |ad Bsp.57 \hinweis{Frosch} Die ganze Geschichte
kind_von(joseph_II, maria_theresia).                          |ad Bsp.58 \hinweis{Variablen_in_DCGs}
kind_von(joseph_II, franz_I).                                 |ad Bsp.62 \hinweis{Mögliche_Instanzierungen}
kind_von(leopold_II, maria_theresia).                         |ad Bsp.67 \hinweis{Diagonalen}
kind_von(leopold_II, franz_I).                                |          \hinweis{PrologAllgemein}
kind_von(marie_antoinette, maria_theresia).                   |
kind_von(franz_II, leopold_II).                               |Abgabetermine sind nun mittwochs 24h00.
                                                              |1. Abgabetermin ist Mittwoch 22. März. Bis
                                                              |dahin: 54-57 u. 67-69 sowie das 1. Übungsblatt
                                                              |(kommt erst nächste Woche). (58,66 einmal zu
:- kind_von(Kind, Elternteil).                                |machen wär auch nicht schlecht).
:- männlich(Mann).                                            |
! ! Prädikat :männlich/1: nicht oder in nicht geladenem Beisp\|Paßwörter im Sekretariat.
iel definiert. \hinweis{laden}                                |
:- weiblich(Frau).                                            |Die Terminals ncd13, ncd14, ncd27, ncd29
! ! Prädikat :weiblich/1: nicht oder in nicht geladenem Beisp\|bleiben für die Übung gesperrt, um
iel definiert. \hinweis{laden}                                |Rechnerüberlastungen aufgrund des Ausfalls
:- verheiratet(Mann, Frau).                                   |einer DecStation zu vermeiden.
! ! Prädikat :verheiratet/2: nicht oder in nicht geladenem Be\|
ispiel definiert. \hinweis{laden}                             |
                                                              |
# Diese Zusicherungen liefern anfangs -- wenn Sie die         |
# entsprechenden Prädikate noch nich definiert haben --       |
# Fehlermeldungen. Kommentieren Sie die Zusicherungen         |
# deshalb NICHT aus!                                          |
                                                              |
# Geben Sie zumindest vier eigene positive Anfragen an.       |
# Z.B., Wer sind meine Eltern, Großeltern?                    |
# Schreiben Sie mit (%) Kommentaren die entsprechenden        |
# deutschen Sätze.                                            |
                                                              |
# Schreiben Sie die folgenden Sätze als negative              |
# Zusicherungen:                                              |
-----          n599 server 100% 20:18 Freie Zeit xterm  (GUPU) --%%-Emacs: init.hlp         (Hinweise)--All---
```