Prolog tutor

Julian Verdurmen

Seminar Softw. tech. for teaching and learning

September 30, 2009
Outline

1. Introduction to Prolog
   - Basics
   - Simple example

2. Paper
   - The list technique examples
   - Representing knowledge
   - Tutoring
   - Experimental results
   - Conclusions & questions
1 Introduction to Prolog
   - Basics
   - Simple example

2 Paper
   - The list technique examples
   - Representing knowledge
   - Tutoring
   - Experimental results
   - Conclusions & questions
Prolog intro

- PROgrammation en LOGique (French) (programming with logic)
- Logic programming: One of the four major programming paradigms[2].

Commonalities with functional programming: both declarative¹
- What the program should accomplish, rather than describing how.
Used in the field of AI & parsing.

Compared with Haskell:
- Overloading ‘functions’ possible.
- More valuable pattern matching.
- Can also write higher-order programs (Map etc.)

\[1\] Even more declarative than functional programming[3, 4]
Basics
Prolog basics

Variables
Start with uppercase char or _.

Examples Variables
X, Y, ThisIsAVar, _

Atoms
Start with lowercase char or quoted.

Examples Atoms
x, blue, ’Red’, ’im an Atom’

Rules
<head> :- <body>.

Example Rule: Equals
between(X,Y,Z) :- X < Y, Y < Z.

Facts
<head>.
≡ <head> :- true.

Example Fact: Equals
equals(X, X).
Loops: recursion.
, conjunction (AND)
; disjunction (OR)
Simple example
Prolog simple example

Length of a list

% Fact: base case.
length([], 0).

% Rule: recursive case.
length([Head|Tail], L) :- length(Tail, T),
L is T + 1.

How to invoke:

Invoke examples 1

?- length([], Length).
   Length = 0.

?- length([1,2], Length).
   Length = 2.
Prolog simple example

Invoke examples 2

?- length(List,3).
List = [_G717, _G720, _G723].

?- length(List, Length).
List = [],
Length = 0 .

?- length([], 0).
true.

?- length([], 1).
false.
Outline

1. Introduction to Prolog
   - Basics
   - Simple example

2. Paper
   - The list technique examples
   - Representing knowledge
   - Tutoring
   - Experimental results
   - Conclusions & questions
Paper: Guided programming and automated error analysis in an intelligent Prolog tutor[1].

Prolog has simple syntax & few program constructs.

Less knowledge of **structured programming** can be a problem.

structured programming: not the paradigm but probably:
"Computer programming in which the statements are organized in a specific manner to minimize error or misinterpretation."[5]
Previous approach

Limitations:

1. Focused on construction/transformation instead of debugging/tutoring.
2. All possible implementations needed.
3. Relies on buggy versions.
4. Only code-related feedback, not on technique.

Used sometimes schematic representation.

```
schema_A([], []).
schema_A([H|T], [H, T]) :- 
  pre_pred([&1], H, [&3]), 
  schema_A(T, [&5]), 
  post_pred([&6], H, [&7]) .
```

**Figure:** Schemata example
**Approach**

- Sharing of common code pattern.
- Recognize and classify on pattern.
- Usage of programming technique grammar rules
  1. Program recognition,
  2. Program Construction,
  3. and program parsing.
- No buggy versions.
- Not all possible implementations needed.
The list technique examples
Four methods under List technique:

1. Naive
2. Accumulator
3. Railway-shunt
4. Inverse naive

We look at all the four implementations of `reverse` and see how they work.
Naive programming

1. Split the list in Head + Tail
2. Recursive call Tail.
3. Join list (with head reversed list)

```
reverse([], []).
reverse([Head|Tail], List) :- reverse(Tail, Temp), append(Temp, [Head], List).
```
Naive programming - execution

?- reverse([1,2,3], A).
A = [3,2,1].

Call: (1) reverse([1,2,3], _G6728)
  Call: (2) reverse([2,3], _L191)
    Call: (3) reverse([3], _L211)
      Call: (4) reverse([], _L231)
        Exit: (4) reverse([], [])
        Call: (4) append([], [3], _L211)
          Exit: (4) append([], [3], [3])
          Exit: (3) reverse([3], [3])
          Call: (3) append([3], [2], _L191)
            Exit: (3) append([3], [2], [3,2])
            Exit: (2) reverse([2,3], [3,2])
            Call: (2) append([3,2], [1], _G6728)
              Exit: (2) append([3,2], [1], [3,2,1])
              Exit: (1) reverse([1,2,3], [3,2,1])

Accumulator programming

1. Define extra argument for intermediate results.
2. With each call, extend these intermediate results.

\[
\text{reverse}(\text{List1}, \text{List2}) :- \text{reverse}(\text{List1}, [], \text{List2}).
\]

% Overload of reverse with 3 args.
% If empty, result is intermediate result.
\[
\text{reverse}([], \text{List}, \text{List}).
\]
\[
\text{reverse}([\text{Head}|\text{Tail}], \text{AccumList}, \text{List})
\]
\[ :- \text{reverse}(\text{Tail}, [\text{Head}|\text{AccumList}], \text{List}).
\]
Accumulator programming - execution

?- reverse([1,2,3], A).
A = [3,2,1].

Call: (1) reverse([1, 2, 3], _G7459)
  Call: (2) reverse([1, 2, 3], [], _G7459)
    Call: (3) reverse([2, 3], [1], _G7459)
      Call: (4) reverse([3], [2, 1], _G7459)
        Call: (11) reverse([], [3, 2, 1], _G7459)
          Exit: (11) reverse([], [3, 2, 1], [3, 2, 1])
          Exit: (4) reverse([3], [2, 1], [3, 2, 1])
          Exit: (3) reverse([2, 3], [1], [3, 2, 1])
          Exit: (2) reverse([1, 2, 3], [], [3, 2, 1])
        Exit: (1) reverse([1, 2, 3], [3, 2, 1])
      Exit: (3) reverse([2, 3], [1], [3, 2, 1])
    Exit: (2) reverse([1, 2, 3], [], [3, 2, 1])
  Exit: (1) reverse([1, 2, 3], [3, 2, 1])
Analogous to previous (accumulator), but intermediate result as head of the list. (the head is also a list)

```prolog
reverse(List1, List2) :- reverseInner(List1, [List2]).
% No overload possible, same amount of args
% If empty, result is intermediate result.
reverseInner([], [List | List]).
reverseInner([Head | Tail], [Head2 | Tail2])
    :- reverseInner(Tail, [Head2, Head | Tail2]).
```

Julian Verdurmen
Prolog tutor
Railway-shunt programming - execution

?- reverse([1,2,3], A).
A = [3,2,1].
  Call: (1) reverse([1, 2, 3], _G7650)
    Call: (2) reverseInner([1, 2, 3], [_G7650])
      Call: (3) reverseInner([2, 3], [_G7650, 1])
        Call: (4) reverseInner([3], [_G7650, 2, 1])
          Call: (11) reverseInner([], [3, 2, 1])
            Exit: (11) reverseInner([], [3, 2, 1])
          Exit: (4) reverseInner([3], [3, 2, 1])
        Exit: (3) reverseInner([2, 3], [3, 2, 1])
      Exit: (2) reverseInner([1, 2, 3], [3, 2, 1])
    Exit: (1) reverse([1, 2, 3], [3, 2, 1])
Inverse naive programming

Analogous to naive programming, but splitting the 'result'. (2nd argument)

```
%inverse naive: first append, head of 2nd list.
reverse([],[]).
reverse(List,[Head|Tail]) :- append(Temp,[Head],List), reverse(Temp,Tail).
```
**Inverse naive programming - execution**

```
?- reverse([1,2,3], A).
A = [3,2,1].

Call: (1) reverse([1, 2, 3], _G7650)
  Call: (2) append(_L191, [_G7723], [1, 2, 3])
  Exit: (2) append([1, 2], [3], [1, 2, 3])
  Call: (2) reverse([1, 2], _G7724)
    Call: (3) append(_L271, [_G7735], [1, 2])
    Exit: (3) append([1], [2], [1, 2])
    Call: (3) reverse([1], _G7736)
      Call: (4) append(_L330, [_G7744], [1])
      Exit: (4) append([], [1], [1])
      Call: (4) reverse([], _G7745)
      Exit: (4) reverse([], [])
    Exit: (3) reverse([1], [1])
  Exit: (2) reverse([1, 2], [2, 1])
Exit: (1) reverse([1, 2, 3], [3, 2, 1])
```
Representing knowledge
Need on two levels:

1. Syntactic level
2. Semantics level
   - Also used for exercises.
   - Mapping between code & technique.
Syntactic level

- Technique grammar rules.
  - General form of Prolog clauses
  - Language level instead of grammar-level.
  - Needed for every 'most specialized technique' (Naive, Accumulator, railway-shunt,..)

- Or, Standard programs at coding-level.
  - Paper a bit unclear if reference program isn’t needed:

    "Syntactic knowledge in our Prolog tutor is represented by either programming technique grammar rules for programming techniques at various levels of abstraction, or standard programs at the coding level."

  (Section 3, page 515)

- But in experimentation they have both.

- Can be used also for template generation.
Syntactic level

<list technique program>{Pred} ::= <base case>{Pred}
    <recursive case>{Pred}.
<base case>{Pred} ::= <pred>{Pred}(<list>{{[]},<arguments>>).
<arguments> ::= <argument>|<argument>,<arguments>.
<recursive case>{Pred} ::= <rule head>{Pred,H,T}:-
    <rule body>{Pred,H,T}.
<rule head>{Pred,H,T} ::= <pred>{Pred}(<list split>{H,T},
    <arguments>).
<list split>{H,T} ::= [<head>{H}|<tail>{T}].
<rule body>{Pred,H,T} ::= <pre-predicate>{H},
    <recursive call>{Pred,T},
    <post-predicate>{H}
    |<recursive call>{Pred,T},
    <post-predicate>{H}
    |<pre-predicate>{H},
    <recursive call>{Pred,T}.
<pre-predicate>{H} ::= <pre_pred>({arguments},<variable>{H},
    <arguments>).
<recursive call>{Pred,T} ::= <pred>{Pred}(<variable>{T},
    <arguments>).
<post-predicate>{H} ::= <post_pred>({arguments},
    <processed_head>{H},<arguments>).
Frames: connecting technique-related & code knowledge.

Needed for every *Most specialized technique*.

**Semantics level + exercises**

- **Frames**:
  - Frame type: technique
  - Name: naïve technique
  - Description: To process a list until it is empty by splitting it into the head and the tail, making a recursive call with the tail and putting the result from the recursive call and the processed head together as required.
  - Child nodes: nil

**Frames: program**

- Frame type: program
- Name: naïve program
- Invocation format: `reverse(List, Revlist)`
- Predicate description: Check or get the reversal of a list. If `List` and `Revlist` are instantiated, a check is made to see if one is the reversal of the other. If either `List` or `Revlist` is a variable, it is bound to be the reversal of the other.
- Base cases: naïve clause 1
- Recursive cases: naïve clause 2
- Programming technique: naïve technique

**Frames: recursive case**

- Frame type: recursive case
- Name: naïve clause 2
- Head:
  - Code: `reverse([H|T], L)`
  - Description: Reversing a non-empty list consisting of a head, H, and a tail, T, gets a list, L.
- Body:
  - Recursive call:
    - Code: `reverse(T, L1)`
    - Description: The tail, T, is reversed to get a list, L1.
  - Post-predicate:
    - Code: `append(L1, [H|T], L)`
    - Description: The list, L, is obtained from appending the list, L1, and the list, [H].
Semantics level + exercises

Frame type: exercise
Name: list reversal
Description: Write a program which reverses the elements of a list into another list.
Reference programs: naive, accumulator, inverse, railway_shunt

Figure: Exercise frame
Tutoring
Modes

Two modes:

1. Guided programming
2. Automated error analyze.

For this we need *Program technique recognition.*

- Naive, Accumulator, Railway-shunt, Inverse naive
Mode selection

Try to recognize programming technique

- Tutor on programming technique level (Guided programming)
- Technique correct applied?
  - Yes: Tutor on coding level. (With help of error analysis)
  - No: Tutor on programming technique level (Guided programming)
Parse most abstract, then recursively more specialized.
Combining selection + recognition

Try recognize most abstract technique

Try recognize next level of abstraction. (can be multiple)

Program can be parsed?

Yes

Most specialized technique recognized?

Yes

Tutor on coding-level

No

No

Tutor on programming technique level (Guided programming)
Error analysis & correction

When on tutor level ⇒ error analysis.

- Needed a reference program.
- Compare parsing reference program with to-be-checked program.
- Not to normal form, but generate permutations.
Experimental results
Tried on 125 programs, handed in by students.

- Recognized: 120; 96%
- Unrecognized: 5; 4%

Total results
One level deeper:

- Recursive list technique most abstract, then list technique.
- All abstract techniques were incorrect applied.

Unrecognized:
- Unrecognized, but correct; 1; 20%
- Unrecognized, incorrect; 4; 80%

Recognized:
- Most specialized techniques recognized; 100; 83%
- List technique; 18; 15%
- Recursive list processing technique; 2; 2%
Used techniques

<table>
<thead>
<tr>
<th>Used techniques</th>
<th>Incorrect</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive technique</td>
<td>47</td>
<td>17</td>
</tr>
<tr>
<td>Accumulator technique</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Railway-shunt technique</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Inverse naive technique</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>
Results error detection

- Every detected error could be corrected.
- All unrecognized errors were not in 'most specialized technique'.

![Error detection & correction chart]

- Errors undetected: 52; 31%
- Error detected & corrected: 115; 69%
Conclusions & questions
Conclusions

- Some nice test results
- Liked to see in action ⇒ download not found.
- Still needs a lot of implementations.
  - Reverse has now 7 implementations.
  - Also needed for parameter reordering. (less nice)

On paper:
- Sometimes unclear what belongs to new and old-approaches.
- guided programming/tutor on technique level, error-analysis/tutor on code level, tutoring ⇒ Used inconsistent.

- The feedback in the examples is a (bit) verbose.
  - Sentences of 4 lines
  - Check questions (‘is this right?’)
Questions

Do you think...

- This approach is suitable for larger programs?

- This approach is suitable for other languages?

- Skipping buggy versions is a smart choice?
Questions

Do you think...

- This approach is suitable for larger programs?

- *There will be many permutations, the permutations aren’t linear (but the normal form is?). And also another problem with the order of parameters.*

- This approach is suitable for other languages?

- Skipping buggy versions is a smart choice?
Questions

Do you think...

- This approach is suitable for larger programs?

- *There will be many permutations, the permutations aren’t linear (but the normal form is?). And also another problem with the order of parameters.*

- This approach is suitable for other languages?

- *Maybe other declarative languages but there should not too many programming techniques. Also refactoring is missing for other languages (if there are multiple ways to construct the same).*

- Skipping buggy versions is a smart choice?
Questions

Do you think...

- This approach is suitable for larger programs?
- *There will be many permutations, the permutations aren’t linear (but the normal form is?).* And also another problem with the order of parameters.
- This approach is suitable for other languages?
- *Maybe other declarative languages but there should not too many programming techniques. Also refactoring is missing for other languages (if there are multiple ways to construct the same).*
- Skipping buggy versions is a smart choice?
- *The feedback with buggy versions would be better. Also give error-analysis would be less a problem for more abstract then ‘most-specialized’ techniques.*
[1] Guided programming and automated error analysis in an intelligent Prolog tutor, Jun Hong, accepted 5 February 2004


[3] lambda-the-ultimate.org/node/2410
