

## Laboration 2: Beviskontroll med Prolog

### Inledning

I denna laboration har målet varit att konstruera och implementera en algoritm i Prolog som kan kontrollera om ett givet bevis, skrivet enligt principerna för naturlig deduktion, är korrekt eller ej. Laborationen baseras på inmatning av en sekvens samt ett bevis. Programmet förväntas svara "yes" om beviset är korrekt och visar att sekvensen gäller, och "no" annars. Naturlig deduktion innefattar regler för härledning av nya formler, och Prolog valdes som ett lämpligt språk för att implementera dessa regler.

### Teori

**Naturlig deduktion** är en formell metod inom matematik och logik som används för att härleda sanningar från redan existerande premisser. Den grundläggande idén bakom naturlig deduktion är att bygga argument och bevis stegvis genom att tillämpa olika logiska regler. Metoden fokuserar på att visa att en slutsats är giltig genom att konstruera ett bevis i form av en serie logiskt korrekta steg. Grundläggande regler inom naturlig deduktion inkluderar införande och eliminering av logiska operationer, samt användning av universal- och existenskvantifikatorer.

Införandet av en logisk operator innebär att man visar att en proposition är sann om dess komponenter är sanna. Elimineringen av en operator innebär att man kan använda en sats som innehåller operatoren som en ny premiss. Naturlig deduktion tillåter även användning av bevis genom motsägelse, där man antar motsatsen till det man vill bevisa och visar att detta leder till en logisk motsägelse.

**Prolog**, som står för "Programming in Logic," är ett deklarativt programmeringsspråk som bygger på logiskt programmeringsparadigm. Prolog används för att beskriva relationer och regler snarare än att specificera exakta steg för att utföra en uppgift. Språket är särskilt lämpligt för att hantera kunskapsrepresentation och sökning.

I Prolog beskrivs programlogiken genom regler och fakta som definierar relationer mellan olika entiteter. Språket använder sig av mönstermatchning och rekursion för att söka igenom och manipulera dessa relationer. Prolog är känt för sin användning av backward-chaining, där målet specificeras och systemet arbetar bakåt för att hitta en lösning genom att matcha regler och fakta.

För denna laboration har Prolog valts som språk på grund av dess förmåga att representera och utföra logiska resonemang, vilket är särskilt passande för att implementera regler och kontroller för naturlig deduktion.

### Tillvägagångssätt

Metoden som användes för att implementera programmet kan beskrivas som inkrementell. Det första steget var att skriva ett program som kunde hantera bevis utan boxar, då detta uppfattades som komplicerat. Det skapades först en design som kunde hantera ett subset av de regler som inte kräver boxar för att se att det hela verkade fungera. Efter detta

implementerades alla regler som inte krävde boxar och programmet testades på den fullständiga testsvit som kom med labben. Det verifierades att de test som passerades var just alla dem som inte hade boxar. Efter detta implementerades en utökad version som kunde hantera boxar och programmet testades återigen på testsviter och alla test passerades.

Då testsviten inte var uttömmande gjordes några försök till att testa "edge cases". T.ex. lades det till ett test som innehöll ett korrekt bevis som fyra nästlade boxar. Det gjordes även en del test som bytte ut siffrorna som fungerar som etiketter till varje rad mot andra termer. Programmet godkände dessa typer av omskrivningar så länge man var konsekvent och refererade till rätt etikett, även om denna var i stil med *rad "den här bevisekontrollens suger"* istället för t.ex. *rad 7*. Detta är rimligt då radnumren faktiskt bara är till för tydlighet och att man ska kunna orientera sig lättare, och har inte med korrektheten av själva beviset att göra.

## Algoritm för bevisekontroll

### Utan boxhantering

Vi kunde använda oss av samma grundläggande bevisekontrollalgoritm för både fallet med och utan boxar. Det gick alltså att "bygga på" den ursprungliga algoritmen när hanteringen av boxar infördes.

Denna ursprungliga algoritm har sin utgångspunkt i predikatet *valid\_proff/3* som har två klausuler. Den ena *valid\_goal/2* testar om sista satsen i beviset är samma som målet i sekvensen; den andra *valid\_lines/4* går sedan rekursivt igenom rad för rad i beviset, uppifrån och ner, och kontrollerar så att raden går att matcha mot någon giltig regel. Detta görs genom hjälppredikatet *check\_line/4* som har en klausul per regel för att kunna mönstermatcha mot denna i beviset och sedan kontrollera att regeln är korrekt applicerad. Om raden godkänns sparas den i en ackumulator (*checked\_lines*) för alla godkända rader. Denna ackumulator används sedan som databas vid kontroll av kommande regler för att se om den innehåller de rader som måste finnas ovan i beviset för att få använda en viss regel.

### Med boxhantering

Boxhanteringen implementerades genom att lägga till en klausul av predikatet *check\_line/4*. Denna nya klausul ligger sist av alla då det den mönstermatchar mot är en lista, och därmed skulle vilken bevisrad som helst kunna matchas mot den, då dessa är formaterade som listor också, men på detta sätt fångas de upp av tidigare klausuler. ‘

Denna sista klausul av *check\_line/4* startar igång en ny rekursion med predikatet *valid\_box\_lines/4* för kontroll av alla rader i boxen, på liknande sätt som *valid\_lines/4*. Ett viktigt steg innan detta är dock att den använder predikatet *append/3* med *checked\_lines* och en tom lista för att se till att denna inre rekursion använder en ny variabel som databas och inte *checked\_lines* från den yttre rekursionen. På detta sätt får man med kravet att man inte får referera till rader inifrån en box när man väl är ute ur den. Denna mekanism fungerar även för hela boxar då dessa också läggs till i *checked\_lines* ackumulatören om de godkänns av predikatet *valid\_box\_lines/4*. Då denna design är rekursiv kan denna process göras med ett godtyckligt antal nästlade boxar.

För hantering av boxar krävdes det även en implementering av de fyra regler som använder sig av boxar, samt förekomsten av antaganden. För de nya reglerna skapades hjälppredikatet *find\_box/3*, som går igenom *checked\_lines* och letar efter en box som har första och sista raden i enlighet med de krav som är definierade i för regeln i dess *checked\_lines*-klausul. Om en sådan box hittas godkänns *find\_box* klausulen och därmed även bevisaden.

För hantering av antagande krävdes en speciallösning. Kravet för ett korrekt antagande är att det måste stå som första raden i en box. För att kontrollera detta finns det en andra ackumulator förutom `checked_lines`. Denna kallas för `Box_lines_only` och ackumulerar helt enkelt alla godkända rader i en box. Syftet med detta är att `check_line`-klausulen för antagande kan kontrollera om `Box_lines_only` är tomma listan, och då godkänna regeln, eftersom detta, tillsammans med det faktum att vi är inne i en box betyder att kraven för antagande är uppfyllda. Detta är orsaken till att det separata predikatet `valid_box_lines/4`; det är endast här som godkända rader läggs till i `Box_lines_only` och inte bara i `checked_lines`. För att omöjliggöra att ett antagande godkänns om vi inte är inne i en box binds `Box_lines_only` till en icke-tom lista (`[dummy]`) i `valid_lines/4`.

## Tabell över predikat

Predikat	Argument	Sann om: (falsk annars)
<code>verify/1</code>	1. Filnamnet	- Beviset stämmer som helhet, dvs om <code>valid_proof/3</code> stämmer. (Tar in en fil)
<code>valid_proof/3</code>	1. Beviset "Proof" 2. Premisser "Premis" 3. Målet "Goal"	- " <code>valid_goal/2</code> " är sann (jämför sista raden med målet) - " <code>valid_lines/4</code> " är sann (kontrollerar varje rad i beviset)
<code>valid_goal/2</code>	1. Målet "Goal" 2. Beviset "Proof"	- satsen i sista bevisraden är samma som målet i sekventen. Detta kontrolleras genom de två hjälppredikaten - <code>last_egen/2</code> - <code>second_element_equals</code>
<code>last_egen/2</code>	1. inputlista 2. Variabel att binda resultatet till.	Basfall - Inputlistan innehåller ett element, detta unifieras då till argument 2. Rekursiv - <code>last_egen</code> för svansen av inputlistan är sann.
<code>second_element_equals</code>	1. inputlista på formen <code>[_E_]_</code> 2. ett element att matcha mot	- Sann om <code>E = element</code> .
<code>valid_lines/4</code>	1. Premisser "Premis" 2. Beviset "Proof" 3. Ackumulator för godkända rader i beviset 4. Box-ackumulator	Basfall: - Listan med bevisrader är tom. Rekursiv: - Varje rekursiv kallelse är sann - En rekursiv kallelse är sann om <code>check_line/4</code> är sann (kontrollerar en rad)
<code>valid_box_lines/4</code>	som ovan	Basfall: - Listan med bevisrader är tom.

		Rekursiv: <ul style="list-style-type: none"> <li>- Varje rekursiv kallelse är sann</li> <li>- En rekursiv kallelse är sann om check_line/4 är sann (kontrollerar en rad)</li> </ul>
check_line/4	<ul style="list-style-type: none"> <li>- Premisser "Premis"</li> <li>- En rad: [Index, Påstende, Regel]</li> <li>- Checkade rader</li> <li>- Box-ackumulator</li> </ul>	<ul style="list-style-type: none"> <li>- Regeln den matchar med och påståendet är möjliga med hänsyn till "Premis"</li> <li>- Regeln och påståendet är möjliga med hänsyn till tidigare rader</li> </ul> (Predikatet är implementationen av olika regler inklusive premiss och antagande)
find_box/3	<ol style="list-style-type: none"> <li>1. Första raden i boxen</li> <li>2. Sista raden i boxen</li> <li>3. Ackumulator för godkända rader i beviset</li> </ol>	<ul style="list-style-type: none"> <li>- Om en sekvens som matas in (i vårt fall en box) har den första och sista rad som söks enligt vad som är definierad för regeln i check_line/4</li> </ul>

## Resultat

```

valid01.txt passed
valid02.txt failed. The proof is valid but your program rejected it!
valid03.txt failed. The proof is valid but your program rejected it!
valid04.txt failed. The proof is valid but your program rejected it!
valid05.txt failed. The proof is valid but your program rejected it!
valid06.txt failed. The proof is valid but your program rejected it!
valid07.txt failed. The proof is valid but your program rejected it!
valid08.txt failed. The proof is valid but your program rejected it!
valid09.txt passed
valid10.txt failed. The proof is valid but your program rejected it!
valid11.txt failed. The proof is valid but your program rejected it!
valid12.txt passed
valid13.txt passed
valid14.txt passed
valid15.txt passed
valid16.txt failed. The proof is valid but your program rejected it!
valid17.txt failed. The proof is valid but your program rejected it!
valid18.txt failed. The proof is valid but your program rejected it!
valid19.txt failed. The proof is valid but your program rejected it!
valid20.txt failed. The proof is valid but your program rejected it!
valid21.txt passed
valid22.txt failed. The proof is valid but your program rejected it!
valid23.txt passed
invalid01.txt passed
invalid02.txt passed
invalid03.txt passed
invalid04.txt passed
invalid05.txt passed
invalid06.txt passed
invalid07.txt passed
invalid08.txt passed
invalid09.txt passed
invalid10.txt passed
invalid11.txt passed
invalid12.txt passed
invalid13.txt passed
invalid14.txt passed
invalid15.txt passed
invalid16.txt passed
invalid17.txt passed
invalid18.txt passed
invalid19.txt passed
invalid20.txt passed
invalid21.txt passed
invalid22.txt passed
invalid23.txt passed
invalid24.txt passed
invalid25.txt passed
invalid26.txt passed
invalid27.txt passed
invalid28.txt passed
invalid29.txt passed
invalid30.txt passed
invalid31.txt passed
invalid32.txt passed

```

Resultat för utan användning av boxar

```
valid01.txt passed
valid02.txt passed
valid03.txt passed
valid04.txt passed
valid05.txt passed
valid06.txt passed
valid07.txt passed
valid08.txt passed
valid09.txt passed
valid10.txt passed
valid11.txt passed
valid12.txt passed
valid13.txt passed
valid14.txt passed
valid15.txt passed
valid16.txt passed
valid17.txt passed
valid18.txt passed
valid19.txt passed
valid20.txt passed
valid21.txt passed
valid22.txt passed
valid23.txt passed
invalid01.txt passed
invalid02.txt passed
invalid03.txt passed
invalid04.txt passed
invalid05.txt passed
invalid06.txt passed
invalid07.txt passed
invalid08.txt passed
invalid09.txt passed
invalid10.txt passed
invalid11.txt passed
invalid12.txt passed
invalid13.txt passed
invalid14.txt passed
invalid15.txt passed
invalid16.txt passed
invalid17.txt passed
invalid18.txt passed
invalid19.txt passed
invalid20.txt passed
invalid21.txt passed
invalid22.txt passed
invalid23.txt passed
invalid24.txt passed
invalid25.txt passed
invalid26.txt passed
invalid27.txt passed
invalid28.txt passed
invalid29.txt passed
invalid30.txt passed
invalid31.txt passed
invalid32.txt passed
```

Resultat för användning av boxar

## Appendix A - Koden

```
verify(InputFileName) :-
    see(InputFileName),
    read(Premis), read(Goal), read(Proof),
    seen,
    valid_proof(Premis, Goal, Proof).

valid_proof(Premis, Goal, Proof) :-
    valid_goal(Goal, Proof),
    %gå igenom rad för rad och kolla att beviset är korrekt
    valid_lines(Premis, Proof, [], _). %accumulator som fungerar som bokföring för
    alla lines som gåttts igenom hittills.

%Basfall - när man kommit fram till slutet av beviset så är Proof-listan tom helt
enkel.
valid_lines(_, [], _, _).
%Rekursiv
valid_lines(Premis, [Current_Line|Rest], Checked_Lines, _) :-
    %Kontrollera att Current_line är godkänd enligt den regel som står där.
    check_line(Premis, Current_Line, Checked_Lines, [dummy]),
    %Gör samma för resten av beviset
    valid_lines(Premis, Rest, [Current_Line|Checked_Lines], _).

valid_box_lines(_, [], _, _).
valid_box_lines(Premis, [Current_Line|Rest], Checked_Lines, Box_Lines_Only) :-
    %Kontrollera att Current_line är godkänd enligt den regel som står där.
    check_line(Premis, Current_Line, Checked_Lines, Box_Lines_Only),
    %Gör samma för resten av beviset
    valid_box_lines(Premis, Rest, [Current_Line|Checked_Lines],
    [Current_Line|Box_Lines_Only]).

valid_goal(Goal, Proof) :-
    last_egen(Proof, LastLine), %LastLine = lista med tre termer
    second_element_equals(LastLine, Goal).
last_egen([R], R).
last_egen([_|T], R) :- last_egen(T, R).
second_element_equals([_, E|_], Element) :- E = Element.

% PREMISS
check_line(Premis, [_, P, premise], _, _) :- %[Rad, Proposition, regel]
    member(P, Premis).
% ASSUMPTION
check_line(_, [_, _, assumption], _, Box_Lines_Only) :-
    Box_Lines_Only = [].
% COPY
check_line(_, [_, P, copy(X)], Checked_Lines, _) :-
    member([X, P, _], Checked_Lines).
% AND INTRODUCTION
check_line(_, [_, and(P, Q), andint(X,Y)], Checked_Lines, _) :-
    member([X, P, _], Checked_Lines),
    member([Y, Q, _], Checked_Lines).
% AND ELIMINATION 1
check_line(_, [_, P, andel1(X)], Checked_Lines, _) :-
    member([X, and(P, _), _], Checked_Lines).
% AND ELIMINATION 2
check_line(_, [_, P, andel2(X)], Checked_Lines, _) :-
    member([X, and(_, P), _], Checked_Lines).
% OR INTRODUCTION 1
```

```

check_line(_, [_ , or(P,_) , orint1(X)] , Checked_Lines, _) :-
    member([X, P, _] , Checked_Lines).
% OR INTRODUCTION 2
check_line(_, [_ , or(_,Q) , orint2(X)] , Checked_Lines, _) :-
    member([X, Q, _] , Checked_Lines).
% OR ELIMINATION
check_line(_, [_ , R, orel(X,Y,U,V,W)] , Checked_Lines, _) :-
    member([X, or(P,Q) , _] , Checked_Lines) ,
    find_box([Y, P, assumption] , [U, R, _] , Checked_Lines) ,
    find_box([V, Q, assumption] , [W, R, _] , Checked_Lines).
% IMPLICATION INTRODUCTION
check_line(_, [_ , imp(P,Q) , impint(X,Y)] , Checked_Lines, _) :-
    find_box([X, P, assumption] , [Y, Q, _] , Checked_Lines).
% IMPLICATION ELIMINATION
check_line(_, [_ , Q, impel(X,Y)] , Checked_Lines, _) :-
    member([X, P, _] , Checked_Lines) ,
    member([Y, imp(P, Q) , _] , Checked_Lines).
% NEGATION INTRODUCTION
check_line(_, [_ , neg(P) , negint(X,Y)] , Checked_Lines, _) :-
    find_box([X, P, assumption] , [Y, cont, _] , Checked_Lines).
% NEGATION ELIMINATION
check_line(_, [_ , cont, negel(X,Y)] , Checked_Lines, _) :-
    member([X, P, _] , Checked_Lines) ,
    member([Y, neg(P) , _] , Checked_Lines).
% CONTRADICTION ELIMINATION
check_line(_, [_ , _ , contel(X)] , Checked_Lines, _) :-
    member([X, cont, _] , Checked_Lines).
% DOUBLE NEGATION INTRODUCTION
check_line(_, [_ , neg(neg(P)) , negnegint(X)] , Checked_Lines, _) :-
    member([X, P, _] , Checked_Lines).
% DOUBLE NEGATION ELIMINATION
check_line(_, [_ , P, negnegel(X)] , Checked_Lines, _) :-
    member([X, neg(neg(P)) , _] , Checked_Lines).
% MODUS TOLLENS
check_line(_, [_ , neg(P) , mt(X,Y)] , Checked_Lines, _) :-
    member([X, imp(P, Q) , _] , Checked_Lines) ,
    member([Y, neg(Q) , _] , Checked_Lines).
% PROOF BY CONTRADICTION
check_line(_, [_ , P, pbc(X,Y)] , Checked_Lines, _) :-
    find_box([X, neg(P) , assumption] , [Y, cont, _] , Checked_Lines).
% LEM
check_line(_, [_ , or(P, neg(P)) , lem] , _ , _).
% BOX
check_line(Prem, Box, Checked_Lines, _) :-
    append([], Checked_Lines, Box_Lines) , %så att checked lines inte får massa
box-lines på sig när man är klar med denna boxrekursion
    valid_box_lines(Prem, Box, Box_Lines, []).

find_box(First_Row, Last_Row, [Current_Line|Rest]) :-
    nth0(0, Current_Line, First_Row) ,
    last_egen(Current_Line, Last_Row) ;
    find_box(First_Row, Last_Row, Rest).

```



# Appendix B - exempelbevis

$P \vee q, q \vee r, P \rightarrow \neg r \vdash q$

Bevis:

1	$P \vee q$	premiss
2	$q \vee r$	— 11 —
3	$P \rightarrow \neg r$	— 11 —

4	$P$	antagande
5	$q$	— 11 —
6	$q$	copy 5

7	$r$	antagande
8	$\neg r$	$\rightarrow e$ 3, 4
9	$\perp$	$\neg e$ 7, 8
10	$q$	$\perp e$ 9

11	$q$	$\vee e$ 2, 5-6, 7-10
----	-----	-----------------------

12	$q$	antagande
13	$q$	copy 12

14	$q$	$\vee e$ 1, 4-11, 12-13
----	-----	-------------------------

Exempel på ett korrekt icke-trivialt bevis som använder boxar.



1. Bevisa sekventen  $p \vee q, q \vee r, p \rightarrow \neg r \vdash q$  med naturlig deduktion. Använd bara bevisreglerna på sid 27 i boken.

Bevis:

- |     |                        |                       |
|-----|------------------------|-----------------------|
| 1.  | $p \vee q$             | premiss               |
| 2.  | $q \vee r$             | premiss               |
| 3.  | $p \rightarrow \neg r$ | premiss               |
| 4.  | $p$                    | antagande             |
| 5.  | $\neg r$               | $\rightarrow e, 3, 4$ |
| 6.  | $q$                    | $\vee i, 2$           |
| 7.  | $r$                    | antagande             |
| 8.  | $q \vee r$             | $\vee i, 2$           |
| 9.  | $p$                    | $\vee i, 1$           |
| 10. | $\neg r$               | $\rightarrow e, 3, 9$ |
| 11. | $\perp$                | $\neg e, 7, 10$       |
| 12. | $q$                    | $\vee i, 7-11$        |

Exempel på ett felaktigt icke-trivialt bevis som använder boxar.