גירסה 1.00 – 24.9.2004

PROLOG

ניר אדר

מסמך זה הורד מהאתר http://underwar.livedns.co.il.
אין להפיץ מסמך זה במדיה כלשהי, ללא אישור מפורש מאת המחבר.
מחבר המסמך איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

כל הזכויות שמורות ל**ניר אדר**

Nir Adar
Email: underwar@hotmail.com
Home Page: http://underwar.livedns.co.il

תוכן עניינים

תוכן עניינים **2**

פתיחה **4**

תחביר בסיסי **6**

פרדיקטים 6
הערות 7
קלט/פלט 7
עובדות 8
מבנים 9
שאילתות 9
עובדות ומשתנים 11
חוקים 12
תחביר ודוגמאות בסיסיות *12*
משמעות הצהרתית ומשמעות פרוצדורלית *13*
מופע ושינוי *13*
ספיקות *14*

פעולות אריתמטיות **16**

אופרטורים בסיסיים 16
האופרטור = והאופרטור =:= 17
דוגמאות 17
מחלק משותף גדול ביותר (GCD) *17*

רשימות **19**

תחביר בסיסי ועקרונות 19
דוגמאות 21
מספר האיברים ברשימה *21*
שייכות *22*
שרשור רשימות *22*
סכימה של איברי הרשימה *23*
הוספה ומחיקה של איברים *23*

BACKTRACKING AND CUT **24**

מוטיבציה ורעיונות בסיסיים 24
דוגמא 1 25
סוגים של CUT 27

שלילה 28
fail *28*
הפרדיקט not *28*

פתיחה

PROLOG היא שפת תכנות המשמשת לביצוע חישוביים על שמות סימבוליים.

תכנות ב-PROLOG מבוסס על הגדרת יחסים, ולאחר מכן ביצוע שאילתות על היחסים.

התחביר שנציג מתאים לגירסת ה-PROLOG בשם SWI-PROLOG. גירסה זו היא אינטרפרטר אינטראקטיבי, המאפשר לנו להקיש פקודה אחר פקודה.

בגרסה זו הסימון ?- הוא ה-prompt שסביבת העבודה מספקת לנו כאשר היא מוכנה לקבל פקודות.

נפתח בדוגמא, ולאחר מכן נעמיק בתחביר של השפה.

הגדרת עובדות:

```
assert( parent(bob, tom) ).
assert( parent(bob, dan) ).
assert( parent(liz, tom) ).
assert( parent(liz, dan) ).
assert( parent(tom, or)  ).
```

בחרנו להגדיר כי בוב וליז הם ההורים של טום ודן, וכי טום הוא ההורה של אור.

לאחר הגדרת העובדות, נוכל לבצע שאילתות:

```
?- parent(bob, tom).
Yes

?- parent(bob, or).
No
```

שאילתות מורכבות יותר הינן בקשות מידע, למשל נרצה לדעת מי הם ההורים של דן. נכתוב:

```
?- parent(X, dan).
```

האינטרפרטר יחזיר:

```
X = bob
```

נלחץ ; ו-enter ונקבל את התוצאה המתאימה הבאה:

```
X = liz
```

נוכל גם לברר מי ההורה של מי, על ידי:

```
?- parent(X, Y).
```

התוצאה תהיה זוגות של נתונים

```
X = bob
Y = tom ;

X = bob
Y = dan ;

...
```

ביצוע AND לוגי בין שאילתות הוא אפשר על ידי אופרטור פסיק:

```
parent(X, tom) , parent(X, dan).
```

שורה זו תפיק את כל האנשים המוגדרים כהורים של טום וגם של דן.

תחביר בסיסי

פרדיקטים

אחת מאבני הבניין החשובות ביותר של השפה הוא הפרדיקט. הפרדיקט משמש אותנו להגדרת עובדות, שאילת שאילתות ועוד.

התחביר של פרדיקט הינו:

$$pred(\arg_1, \arg_2, \arg_N).$$

כאשר $pred$ הינו שם הפרדיקט, ו- $\arg_1,...,\arg_N$ הם ארגומנטים.

פרדיקט מזוהה על ידי השם שלו ועל ידי מספר המשתנים שלו. שני פרדיקטים בעלי אותו שם אך מספר משתנים שונים נחשבים לשני פרדיקטים שונים.

פרדיקט ללא פרמטרים יכתב בצורה:

$$pred.$$

ארגומנט של פרדיקט יכול להיות:

- **מספר שלם** – שערכו המקסימלי קטן מחזקה כלשהי של 2 (המספר המדויק – תלוי מימוש).
- **מספר ממשי** – למשל, 22.51.
- **אטום** – קבוע טקסט המתחיל באות קטנה.
- **משתנה** – המתחיל באות גדולה או קו תחתון (_). המשתנה "_" (משתנה ששמו קו תחתון בלבד) מכונה משתנה אנונימי.
- **מבנה** (struct) – יודגם בהמשך.

אטום הוא ביטוי הנוצר על ידי אחת הצורות הבאות:

1. רצף של אותיות, מספרים וקו תחתון, המתחיל באות קטנה.
2. רצף של תווים מיוחדים, למשל: <--> או ==:: או .:. וכדו'.
3. רצף כלשהו של תווים המופרד על ידי מרכאות מהסוג ' '. למשל: 'UnDerRwArior@#$'

פרדיקטים ישמשו אותנו למספר שימושים אותם נציג בהדרגה.

- פרדיקטים יכולים לשמש לצורך ייצוג עובדות.
- פרדיקטים יכולים לשמש כמבנים (structs) המאכסנים נתונים.
- פרדיקטים יכולים להתנהג כמעין פונקציות המבצעות חישובים שונים.

הערות

כמו בשפות אחרות, גם שפת PROLOG מכילה מנגנון המאפשר למתכנת לתעד את הקוד שלו.
ישנם שני סוגי הערות בשפת PROLOG, הדומים לאלה של C++:

1. הערות המתחילות בסימון */ ומסתיימות בסימון /*. כל מה שנמצא בין הסימן הפותח לסוגר הוא חלק מההערה. ההערות יכולות להופיע בכל מקום בקוד.
2. הערות המסומנות על ידי %. הערות אלו הינן הערות שורה (בדומה ל-\\ בשפות C++). כל דבר שמופיע אחרי הסימן % עד לסוף השורה נחשב חלק מההערה.

קלט/פלט

שני פרדיקטים עיקריים ישמשו אותנו לקלט ופלט בשפת PROLOG. הפרדיקט write(X) והפרדיקט nl. הפרדיקט write(X), כ-side effect, מדפיס למסך את X. הפרדיקט nl מעביר אותנו לשורה חדשה.

לדוגמא:

```
?- write('hello world') , nl , write('second line').

hello world
second line
Yes
```

עובדות

ציון עובדות מתבצע על ידי פרדיקטים.

נציין למשל את העובדה "השמיים הם כחולים". הפרדיקט המתאים הינו:

```
blue(sky).
```

ישנן שתי דרכים לגרום לסביבת העבודה להכיר את הפרדיקט:

הדרך הראשונה היא שימוש במילה assert בזמן העבודה האינטראקטיבית. נכתוב:

```
assert( blue(sky) ).
```

והעובדה הנ"ל תתווסף לבסיס הנתונים של PROLOG (בסיס נתונים זה מכונה **logicbase**).

הדרך השניה היא על ידי קובץ. נכתוב את העובדות, ובהמשך גם את החוקים והמבנים, שאנחנו רוצים להגדיר בתוך קובץ טקסט, ונשמור אותו.

לאחר מכן נשתמש בפרדיקט consult כדי לטעון את הקובץ, בצורה הבאה:

```
consult('filename').
```

העובדות יקראו מהקובץ ויכנסו אל בסיס הנתונים של PROLOG.

כאשר אנחנו מציינים עובדות בתוך קובץ טקסט, אין צורך להשתמש בפרדיקט assert.

כאשר נציין בדוגמאות בהמשך כי "קיימות העובדות הבאות", נתכוון שעל ידי שימוש ב-assert או בקובץ טקסט העובדות הוכנסו אל המערכת.

מבנים

תוכנית הכתובה ב-PROLOG תשתמש בפרדיקטים כדי לשמור נתונים ולציין עובדות. למשל, תוכנית עסקית תוכל להכיל את הנתונים הבאים:

```
customer('John Jones', boston, good_credit).
customer('Sally Smith', chicago, good_credit).
```

במקרה זה, הפרדיקט משמש להגדרת **מבנה (struct)** בעל שלושה שדות: שם, עיר וסטאטוס האשראי של בן האדם.

נשים לב שמדובר כאן על צורת הסתכלות. למעשה אין הבדל בין הצהרת המבנים להצהרת העובדות. ההבדל הוא בצורת החשיבה שלנו על הנתונים שאנו מוסיפים אל בסיס הנתונים של PROLOG.

כדי לגרום לשפה להכיר את שני המבנים (הפרדיקטים) לעיל, נכתוב:

```
assert( customer('John Jones', boston, good_credit) ).
assert( customer('Sally Smith', chicago, good_credit) ).
```

שאילתות

PROLOG מבצע שאילתות על ידי התאמת תבניות. תבנית השאילתה מכונה **מטרה (goal)**. אם קיימת בסביבת העבודה עובדה המתאימה למטרה, אז השאילתה מצליחה והמקשיב מקבל את התשובה 'yes'. אחרת, אם אין תבנית מתאימה, השאילתה נכשלת ומוחזרת התשובה 'no'.

התאמת התבניות ב-PROLOG נקראת **יוניפיקציה (unification)**.
במקרה כמו שראינו קודם, שישנן רק *עובדות* במאגר המידע, היוניפיקציה מצליחה אם מתקיימים שלושה תנאים:

1. לפרדיקט המטרה ולפרדיקט שבבסיס הנתונים יש שם זהה.
2. לשני הפרדיקטים יש אותו מספר ארגומנטים.
3. כל הארגומנטים של הפרדיקטים זהים.

דוגמא:

נניח כי קיימות העובדות הבאות במערכת:

```
room(kitchen).
room(office).
room(hall).

door(office, hall).
door(kitchen, office).

location(desk, office).
location(apple, kitchen).
location(computer, office).
```

ונביט בשאילתות הבאות:

```
?- location(apple, kitchen).
Yes

?- location(kitchen, apple).
No
```

אכן, התפוח ממוקם במטבח ולא המטבח בתוך התפוח. ההתאמה נעשתה בהתאם לחוקים שצויינו לעיל.

נשים לב לבעיה היכולה להתעורר בצורה כזו. נביט בשאילתות הבאות:

```
?- door(office, hall).
yes

?- door(hall, office).
no
```

לפי אותם חוקים, קיבלנו שבין המשרד לחדר האוכל יש דלת, אולם לא להפך.

מטרות יכולות להיות מוכללות על ידי שימוש *במשתנים*.

משתנים מוסיפים מימד ליוניפיקציות. עם משתנים, יוניפיקציה תצליח אם:

1. לפרדיקט המטרה ולפרדיקט שבבסיס הנתונים יש שם זהה.
2. לשני הפרדיקטים יש אותו מספר ארגומנטים.
3. כל הארגומנטים (שאינם משתנים) של הפרדיקטים זהים.

ארגומנטים בפרדיקט המטרה שהם משתנים יתאימו לכל ערך בבסיס הנתונים.

אחרי יוניפיקציה מוצלחת, המשתנה הלוגי מקבל את הערך של הביטוי שהוא הותאם איתו. פעולה זו נקראת **קישור המשתנה (binding the variable)**. כאשר מטרה עם משתנים מאוחדת בהצלחה עם עובדת בבסיס הנתונים, PROLOG מחזירה את הערך של המשתנה החדש.

לדוגמא:

```
?- room(X).

X = kitchen ;
X = office ;
X = hall ;
No
```

המשמעות של ה-No בסוף הוא שאין עוד תשובות.

דוגמאות נוספות:

```
?- location(Thing, office).

Thing = desk ;
Thing = computer ;
No


?- location(Thing, Place).

Thing = desk
Place = office ;
Thing = apple
Place = kitchen ;
Thing = computer
Place = office ;
No
```

עובדות ומשתנים

ניתן להשתמש במשתנים גם כאשר מגדירים עובדות.

למשל אם נגדיר:

```
?- assert(sleeps(X)).
```

אז אנחנו מציינים בעצם ש-"כולם ישנים".

נבדוק זאת:

```
?- sleeps(nir).
Yes

?- sleeps(eyal).
Yes
```

חוקים

מלבד עובדות, רכיב עיקרי נוסף של PROLOG הוא **החוקים (Rules)**. עובדות שימשו אותנו לייצוג מידע, או ציון עובדה, למשל "בקיץ חם".

חוקים יאפשרו לנו לנסח משפטים בסגנון "אם יש עננים, אז יורד גשם".

חוקים הם המקבילים ל-"פסוקים" מתחום הלוגיקה, ולעתים אף נכנם בשם זה.

תחביר ודוגמאות בסיסיות

התחביר של חוק הינו:

```
head :- body.
```

כאשר head הינו הגדרת פרדיקט (עובדה), הסימן -: נקרא כ-"אם" ו-body מכיל מטרה אחת או יותר.

לדוגמא, הביטוי

```
offspring(Y, X) :- parent(X, Y).
```

מגדיר את הקשר הבא:

```
If parent(a, b) then offspring(b, a);
```

כעת בהתאם לדוגמא שראינו בפרק הפתיחה, יתקיים:

```
?- offspring(tom, bob).
Yes
```

הערה: גם במקרה זה יש לזכור שאם איננו מכניסים את החוקים בתוך קובץ, אלא כותבים אותם ישירות אל סביבת העבודה, יש להשתמש בפקודת assert.

על ידי חוקים, נוכל לפתור את הבעיה שראינו קודם עם הדלתות. להזכיר, הבעיה היתה שכאשר הגדרנו שקיימת דלת בין המשרד לחדר האוכל, לא הגדרנו באותו משפט שהדלת הינה דו כיוונית.

נביט בחוקים הבאים:

```
connect(X,Y) :- door(X,Y).
connect(X,Y) :- door(Y,X).
```

חוקים אלה אומרים "החדר X מקושר לחדר Y אם יש דלת בין X ל-Y או יש דלת בין Y ל-X".

כעת, הפרדיקט connect מתנהג כפי שאנו מצפים:

```
?- connect(X, Y).

X = office
Y = hall ;
X = hall
Y = office ;
...
```

משמעות הצהרתית ומשמעות פרוצדורלית

בהינתן החוק: **P :- Q, R.** נוכל לאמר:

1. P הוא אמת אם Q וגם R הם אמת.
2. מ-Q ומ-R נובע P.
3. כדי לפתור את הבעיה P, יש לפתור ראשית את תת-הבעיה Q ולאחר מכן את תת-הבעיה R.
4. כדי לספק את P, יש לספק ראשית את Q ולאחר מכן את R.

שתי האינטרפרטציות הראשונות מכונות ***המשמעות ההצהרתית* (declarative meaning)**.

שתי האינטרפרטציות האחרונות מכונות ***המשמעות הפרוצדורליות* (procedural meaning)**.

מופע ושינוי

הגדרה: יהא C פסוק. **מופע (instance)** של הפסוק C הינו הפסוק C בו כל אחד מהמשתנים שבו מוחלף על ידי עצם כלשהו. לדוגמא: יהא הפסוק C: has_a_child(X) :- parent(X, Y).

אזי ההצהרות הבאות הינן מופעים של C:

```
has_a_child(peter) :- parent(peter, Z).
has_a_child(dan) :- parent(dan, fat(oz)).
```

<u>הגדרה</u>: יהא C פסוק. **שינוי (variant)** של פסוק C הינו מופע של C בו כל משתנה של C מוחלף על ידי משתנה אחר.

לדוגמא, עבור הפסוק .has_a_child(X) :- parent(X, Y) הביטוי הבא הינו שינוי:

```
has_a_child(X1) :- parent(X1, X2).
```

ספיקות

הערך של מטרה G הוא אמת אם ורק אם קיים פסוק C ב-logicbase וקיים מופע I של C המקיימים:

1. הראש (head) של I זהה ל-G.
2. כל המטרות בגוף (body) של I הם אמת.

סימן פסיק בין מטרות מסמן חיתוך של המטרות (AND).

סימן נקודה-פסיק בין מטרות מסמן איחוד של מטרות (OR).

לדוגמא, המשמעות של הביטוי הבא הינה: "P הוא אמת אם Q הוא אמת או R הוא אמת":

```
P :- Q;R.
```

כאשר PROLOG מקבלת מטרה או רצף של מטרות, היא מנסה **לספק** אותם. אם היא מצליחה, היא עונה Yes ומציינת איך זה נעשה. אחרת, השפה מחזירה No.

עובדות וחוקים מתקבלים **כאקסיומות**. המטרה/השאלה נחשבת **למשפט (theorem)** אותו צריך להוכיח.

PROLOG מתחילה עם המטרות הנתונות, ועל ידי שימוש *בחוקים* ממירה את המטרות במטרות חדשות, עד שהיא מגיעה למצב בו המטרות החדשות הינן עובדות (ללא משתנים).

PROLOG משתמשת בטכניקת **חיפוש לעומק (backtracking)** כדי לייצר את כל המופעים האפשריים תוך כדי הנסיון לספק מטרה. אנחנו משתמשים בחיפוש לעומק, בכך שאם מופע נכשל (הוא אינו מתאים למטרה), PROLOG חוזרת לאחור למקום האחרון בו נבחרה עובדה או חוק, ומחפשת את החוק או העובדה הבאה בהם ניתן להשתמש. במידה ואין כאלה, PROLOG הולכת שלב נוסף אחורה, וכך הלאה.

דוגמא:

נתונים העובדות והחוקים הבאים:

```
1) parent(pam,bob).
2) parent(tom,bob).
3) parent(tom,liz).
4) parent(bob,ann).
5) parent(bob,pat).
6) parent(pat,jim).

offspring(Y,X) :-
      parent(X, Y).
predecessor(X,Z) :- % rule R1
      parent(X, Z).
predecessor(X,Z) :- % rule R2
      parent(X,Y),
      predecessor(Y,Z).
```

נניח כי ניתנת המטרה

```
predecessor(tom, pat).
```

החוק הראשון שנמצא שיכול להתאים הוא R1.

```
X = tom, Z= pat
```

מתקיים:

```
parent(tom, pat) => fail
```

ולכן אנחנו חוזרים אחורה אל המטרה predecessor(tom, pat).

החוק הבא המתאים הוא R2:

```
X = tom, Z = pat. (Y not yet instantiated)
parent(tom, Y), predecessor(Y, pat)
```

PROLOG כעת תנסה לספק את המטרה בסדר בו הן מופיעות.

parent(tom, Y) מתאים לעובדה parent(tom, bob) ומכן נסיק Y = bob. התוצאה:

```
predecessor(bob, pat)
```

החוק הראשון המתאים הוא R1:

```
X' = bob, Z' = pat
parent(bob, pat) => yes
```

הגענו להצלחה. מתחילים לעלות למעלה ומחזירים הצלחה ביציאה מה-backtracking.

פעולות אריתמטיות

אופרטורים בסיסיים

על מנת להיות שפה שימושית, שפת PROLOG תומכת בפעולות אריתמטיות.
פעולות אריתמטיות אינן מתיישבות ממש עם התפיסה של שפת PROLOG, מכיוון שהשפה מתבססת על התאמת תבניות (pattern matching) ופיענוח של ביטויים אריתמטיים אינו צורה של התאמת תבניות.
לפיכך, הפיענוח מתבצע רק אם אנחנו מציינים במפורש, על ידי מילת המפתח is, שאנחנו מעוניינים בו.
אם נצהיר, למשל, את ההצהרה הבאה:

```
?- X = 1 + 2.
X = 1 + 2
```

נראה כי לא בוצע כל פיענוח. לעומת זאת, ההצהרה הבאה כן תפוענח:

```
?- X is 1 + 2.
X = 3
```

הפעולות האריתמטיות הבסיסיות המוגדרות על ידי השפה הינן: +, -, *, /, mod.

בנוסף לפעולות האריתמטיות מוגדרים בשפה אופרטורים של השוואה. אופרטורים של השוואה גורמים לפיענוח של הביטוי האריתמטי בו הם מופיעים. למשל:

```
?- 23 * 21 > 111.
Yes
```

האופרטורים להשוואה הקיימים בשפה הינם:

X > Y	X גדול מ-Y.
X < Y	X קטן מ-Y.
X >= Y	X גדול או שווה ל-Y.
X =< Y	X קטן או שווה מ-Y.
X =:= Y	הערכים של X ושל Y זהים.
X =\= Y	הערכים של X ושל Y אינם זהים.

האופרטור = והאופרטור =:=

הביטוי X=Y גורם ל-matching בין X ל-Y, וייתכן שגם לקישור משתנים.

הביטוי Y =:= X גורם לפיענוח אריתמטי של X ושל Y והשוואה ביניהם, ואינו מסוגל ליצור מופעים של משתנים.

נציג את ההבדלים על ידי דוגמאות:

```
?- 1 + 2 =:= 2 + 1.
Yes

?- 1 + 2 = 2 + 1.
No

?- 1 + A = B + 2.
A = 2
B = 1

?- 1 + A =:= B + 2.
[WARNING: Unbound variable in arithmetic expression]
   Fail:  (   6) 1 + _G179 =:= _G181 + 2 ?
```

דוגמאות

מחלק משותף גדול ביותר (GCD)

נדגים פרדיקט המחשב את המחלק המשותף הגדול ביותר של שני מספרים.

בהינתן מספרים X ו-Y, המכנה המשותף הגדול ביותר D יכול להימצא בעזרת הכללים הבאים:

1. אם X ו-Y שווים, אז ה-GCD הינו X.
2. אם Y < X אז D שווה ל-GCD של X ושל (Y – X).
3. אם Y > X אז נחליף בין X ל-Y ונפנה לכלל 2.

בשפת PROLOG נכתוב זאת כך:

```
gcd(X, X, X).
gcd(X, Y, D) :-
      X < Y,
      Y1 is Y - X,
      gcd(X, Y1, D).

gcd(X, Y, D) :-
      X > Y,
      gcd(Y, X, D).
```

דוגמאות לשימוש:

```
?- gcd(348, 80, D).
D = 4
Yes

?- gcd(160, 80, D).
D = 80
Yes
```

רשימות

תחביר בסיסי ועקרונות

רשימות הינן מבנה נתונים המאפשר להחזיק מספר איברים ולבצע עליהם פעולות.
ב-PROLOG רשימה היא אוסף של ביטוים. כל ביטוי יכול להיות כל אחד מסוגי הנתונים של שפת PROLOG, כולל מבנים או רשימות אחרות בעצמו.
מבחינת תחבירית, רשימה מזוהה על ידי סוגריים מרובעים, כאשר האיברים בה מופרדים ביניהם על ידי פסיקים. לדוגמא, נוכל להגדיר רשימה של איברים היכולים להופיע במטבח:

```
[apple, broccoli, refrigerator]
```

בהמשך לדוגמא שראינו קודם, בו הגדרנו בית ובו חפצים, רשימות יאפשרו לנו כעת ייצוג חדש לחפצים בכל מקום:

```
loc_list([desk, computer], office).
loc_list([flashlight, envelope], desk).
```

רשימה מיוחדת היא **הרשימה הריקה.** הרשימה הריקה מיוצגת על ידי סוגריים מרובעים ריקים [] והיא מכונה גם nil. דוגמא לשימוש:

```
loc_list([], hall).
```

יוניפיקציה עובדת על רשימות כמו על כל איבר אחר בשפה, למשל:

```
?- loc_list(X, office).
X = [desk, computer]
```

שני מנגנונים בשפת PROLOG מאפשרים לנו לעבוד בנוחות עם רשימות. הראשון הוא ייצוג הרשימות בשפה כצמד של ראש+זנב. הראש הוא האיבר הראשון ברשימה, והזנב הוא שאר האיברים ברשימה (גישה זו זהה לגישה המוצגת בשפות פונקציונליות כגון ML ו-LISP).
הכלי השני הוא רקורסיה המאפשר לנו מעבר על איברי הרשימה.

PROLOG מאפשרת לנו לייצג את הרשימות במספר דרכים, לצורך הנוחות.

הראנו קודם את הסוג האינטואיטיבי ביותר:

```
[aaa, bbb, ccc]
```

על ידי סוגריים מרובעות אנו מציינים את האיברים כרשימה.

דרך נוספת היא על ידי אופרטור נקודה בתחילת סוגריים:

```
.(head, tail)
```

תיווצר רשימה שהאיבר הראשון שלה הוא head והאחרון הוא tail.

רשימה בת 3 איברים תיכתב בצורה הבאה:

```
.(aaa, .(bbb, .(ccc, [])))
```

צורת הכתיבה המבוססת על סוגריים מלבניות היא למעשה קיצור:

הביטוי [tom, jerry] זהה לביטוי .(tom, .(jerry, [])).

ניתן לייצג גם רשימות בצורה הבאה: [X | Y] כאשר X מקושר אל האלמנט הראשון ברשימה המכונה ראש הרשימה (head) ו-Y מקושר אל זנב הרשימה (tail). זנב הרשימה הוא תמיד רשימה בפני עצמו, וראש הרשימה הוא איבר. למשל נביט בזהויות הבאות:

```
?- [a | [b,c,d]] = [a,b,c,d].
Yes

?- [a | b,c,d] = [a,b,c,d].
No
```

הסיבה לכך שהביטוי הראשון נכון והשני לא, הוא שהביטוי אחרי האופרטור | חייב להיות איבר יחיד שהוא רשימה.

מספר דוגמאות נוספות:

```
?- [H|T] = [apple, broccoli, refrigerator].
H = apple
T = [broccoli, refrigerator]

?- [H|T] = [a, b, c, d, e].
H = a
T = [b, c, d, e]

?- [H|T] = [apples, bananas].
H = apples
T = [bananas]
```

נשים לב למקרה שזנב הרשימה הוא רשימה ריקה:

```
?- [H|T] = [apples].
H = apples
T = []
```

כמו כן נשים לב לבדיקה הבאה:

```
?- [H|T] = [].
No
```

העובדה שיוניפיקציה זו נכשלה היא חשובה. על ידי שימוש בתנאי זה בפונקציות רקורסיביות נוכל לאתר את הרגע בו נגיע אל סוף הרשימה.

דוגמאות

נציג כעת דוגמאות רבות ושימושיות לפעולות על רשימות.

מספר האיברים ברשימה

נשתמש בכללים הבאים:

1. האורך של רשימה ריקה הוא 0.
2. האורך של רשימה שלה ראש H וזנב T הוא 1 + האורך של T.

```
size([],0).
size([H|T],N) :- size(T,N1), N is N1+1.
```

קיימת פונקציה בעלת פעולה זהה הבנויה כבר בתוך השפה. פונקציה זו הינה length.

דוגמא לשימוש בפונקציה:

```
?- size( [1, 2, 3], Len).
Len = 3
```

שייכות

נציג כעת פונקציה אשר בהינתן איבר X בודקת האם X שייך לרשימה.

הרעיון:

1. אם X הוא ראש הרשימה, החזר Yes.
2. אחרת החזר האם X נמצא בזנב הרשימה.

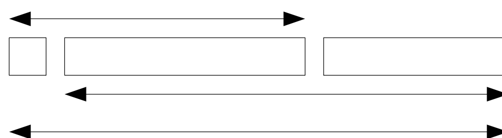
```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

שרשור רשימות

שרשור רשימות יוגדר כך: conc(L1, L2, L3) הוא אמת אם L3 הוא השרשור של L1 ו-L2.

```
conc([], L, L).
conc([X|L1], L2, [X|L3]) :- conc(L1, L2, L3).
```

ההגיון של חוקים אלו יובהר על ידי השרטוט הבא:



ובמילים: אם L3 הוא שרשור של L1 ושל L2, אזי אם נוסיף איבר X להתחלה, נוכל להגיד כי [X|L1] ו-L2 זהים יחד ל-[X|L3].

דוגמא לשימוש:

```
?- conc( [a, b, c], [1, 2, 3], L).
L = [a, b, c, 1, 2, 3].
```

סכימה של איברי הרשימה

אם נתון כי איברי הרשימה הינם מספרים בלבד, נוכל לסכום אותם.

נשתמש בכללים הבאים:

1. הסכום של רשימה ריקה הוא 0.
2. הסכום של רשימה שראשה H וזנבה T הינו H + הסכום של T.

```
sumlist([],0).
sumlist([H|T],N) :- sumlist(T,N1), N is N1+H.
```

הוספה ומחיקה של איברים

הוספת איבר בראש הרשימה יכולה להתבצע בצורה הבאה:

```
add(X, L, [X | L]).
```

מחיקת איבר תוגדר כך:

```
del(X, [X | Tail], Tail).
del(X, [Y | Tail], [Y | Tail1]) :- del(X, Tail, Tail1).
```

Backtracking and Cut

מוטיבציה ורעיונות בסיסיים

נציג כעת תוכנית ב-PROLOG המקבלת ציון בבחינה ומחזירה ציון מילולי. התוכנית תוכל להראות כך:

```
grade(N,excellent) :- N>=85.
grade(N,very_good) :- N<85, N>=75.
grade(N,good) :- N<75, N>=65.
grade(N,pass) :- N<65, N>=55.
grade(N,fail) :- N<55.
```

למרות שתוכנית זו עובדת, היא איננה יעילה. השאילתה grade(90, G) תחזיר G=excellent כמצופה, אולם לאחר שהמטרה סופקה, PROLOG תחזור בחיפוש לעומק כדי לחפש פתרונות אחרים. כדי לעשות זאת, PROLOG תעבור על כל האפשרויות האחרות, ותכשל בגוף של כל אחד מהחוקים האחרים.

בשפת C או C++ למשל היינו מממשים תוכנית זו בצורה יעילה יותר על ידי מבנה switch, למשל.

```
int excellent(int n)    { return n>=85; }
int very_good(int n)    { return n<85 && n>=75; }
int good(int n)   { return n<75 && n>=65; }
int pass(int n)   { return n<65 && n>=55; }
int fail(int n)   { return n < 55; }


switch(n)
{
    case(excellent(n)): cout << "Excellent"; break;
    case(very_good(n)): cout << "Very good"; break;
    case(good(n)): cout << "Good"; break;
    case(pass(n)): cout << "Pass"; break;
    case(fail(n)): cout << "Fail";
}
```

ברגע שאחד התנאים מתקיים, שאר התנאים לא נבדקים עקב ה-break.

אנחנו מסוגלים לבצע משהו דומה לאפקט זה של ה-break בשפת PROLOG על מנת לשפר את היעילות של התוכנית. בגדול – אנחנו רוצים להגיד ל-PROLOG שברגע שהיא הצליחה לספק גירסה אחת של הפרדיקט, היא איננה צריכה להסתכל על פרדיקטים נוספים. המקביל של שפת PROLOG ל-break בשפת C מכונה cut והוא מסומן על ידי סימן קריאה – "!".

נשנה את התוכנית שהדגמנו על מנת שתשתמש ב-cut. אנחנו מנצלים בדוגמא את מנגנון ה-cut, וכן את הסדר ש-PROLOG מבצעת את ההוראות.

```
grade(N,excellent)       :- N>=85, ! .
grade(N,very_good)       :- N>=75, ! .
grade(N,good)            :- N>=65, ! .
grade(N,pass)            :- N>=55, ! .
grade(N,fail)            :- N<55.
```

פרדיקט ה-cut אומר ל-PROLOG שלא להמשיך הלאה לאחר נקודה זו כדי לחפש פתרונות נוספים. כאשר PROLOG מגיעה ל-"!" כל הפתרונות שנמצאו עד אותו רגע נקבעים להיות הפתרונות היחידים. מבחינת תחביר, cut יכול להופיע בכל מקום בו פרדיקט יכול להופיע. הפרדיקט "!" תמיד מחזיר הצלחה.

לסיכום, הפעולה של cut הינה כלהלן:

1. כל המשתנים המקושרים לערכים בנקודה זו אינם יכולים לקבל ערך נוסף.
2. לא יקראו גרסאות נוספות של פרדיקטים לפני שה-cut יובא בחשבון.
3. לא יקראו גרסאות עוקבות של הפרדיקט שבראש החוק הנוכחי.
4. ה-cut תמיד מצליח.

כל פתרון נוסף של השאילתה חייב להגיע מחיפוש לעומק בין ה-cut לסוף החוק הנוכחי.

דוגמא 1

דוגמא זו נלקחה ממסמך שחיבר James Power בנושא. נגדיר את העובדות הבאות ואת החוקים הבאים:

```
holiday(friday,may1).

weather(friday,fair).
weather(saturday,fair).
weather(sunday,fair).

weekend(saturday).
weekend(sunday).

% We go for picnics on good weekends and May 1st
picnic(Day) :- weather(Day,fair), weekend(Day).
picnic(Day) :- holiday(Day,may1).
```

כעת נשאל – מתי הולכים לפיקניק?

```
picnic(When).
```

ניתן לראות כי שאילתה זו תחזיר 3 תשובות.

נשנה כעת את ההגדרה של picnic להיות ההגדרה הבאה:

```
picnic(Day) :- weather(Day,fair), !, weekend(Day).
picnic(Day) :- holiday(Day,may1).
```

ונשאל שוב – מתי הולכים לפיקניק??

```
picnic(When).
```

כאשר נשאל זאת, PROLOG תנסה לספק את הפסוק

```
weather(When,fair), !, weekend(When).
```

העובדה הראשונה המתאימה היא:

```
weather(friday,fair)
```

PROLOG עוברת את ה-cut ואז מנסה לספק את:

```
....., !, weekend(friday).
```

פסוק זה אינו נכון והיוניפיקציה נכשלת. קודם PROLOG היתה ממשיכה לאחור ומגיעה אל הביטוי

```
weather(saturday,fair)
```

אבל כעת ה-cut נועל את פרולוג בין סימן ה-cut לסוף הפסוק, ולכן PROLOG תחזיר No.

נשנה שוב את החוקים לגבי picnic כדי לראות התנהגות נוספת של cut:

```
picnic(Day) :- weather(Day,fair), weekend(Day), !.
picnic(Day) :- holiday(Day,may1).
```

כעת כאשר נבצע את השאילה, נגיע כמו קודם למצב:

```
....., weekend(friday), !.
```

במקרה זה, weekend(friday) ייכשל, ולכן לא נגיע אל ה-cut.

נחזור לאחור. מכיוון שקיימת העובדה:

```
weather(saturday,fair).
```

אזי תת המטרה החדשה תיקבע להיות:

```
....., weekend(saturday), !.
```

כעת אנחנו עוברים את המטרה בהצלחה, ו-PROLOG מגיעה אל ה-cut.

מכיוון שמטרת העל הושגה בהצלחה, השפה תדפיס:

```
When = saturday.
```

עם זאת, מכיוון שהגענו אל cut אז PROLOG לא תחפש פתרונות נוספים, וזה יהיה הפתרון היחיד.

המקרה האחרון הוא כלהלן:

```
picnic(Day) :- !, weather(Day,fair), weekend(Day).
picnic(Day) :- holiday(Day,may1).
```

במקרה זה יהיו שתי תוצאות:

```
When = saturday.
When = sunday.
```

את ההוכחה לכך נשאיר כתרגיל לקורא.

סוגים של cut

השימוש ב-cut מאפשר לנו לכתוב תוכניות בצורה יותר אפקטיבית. אם זאת, cut הופך את התוכניות לקשות יותר להבנה, ובעלות טבע פחות "לוגי". ניתן לחלק את ה-cut השונים לשני סוגים, בהם כבר נתקלנו בדוגמאות הקודמות:

Green cuts: cuts אלו פשוט גורמים לתוכנית לעבוד בצורה יעילה יותר. דרך פעולתם היא ביטול פעולות מיותרות שהמתכנת יודע שלא ישפיעו על תוצאת החישוב. Cuts אלו אינם מבטלים פתרונות מהתוצאה הסופית. ריצה של תוכנית ללא cuts כאלה עדיין תעבוד, למרות שהיא עלולה לקחת זמן רב יותר.

Red cuts – cuts הגורמים לתוכנית לרוץ בצורה שונה. הם עושים זאת על ידי ביטול חלק מהפתרונות שהיו עלולים להתקבל. על ידי כך הם משנים את המשמעות הלוגית של התוכנית.

בעוד ש-green cuts יעילים לשיפור ביצועי התוכנית, אנו ננסה להמנע מ-red cuts במידת האפשר.

שלילה

fail

כאשר אנחנו מבקשים מ-PROLOG לספק מטרה P ו-PROLOG עונה לנו ב-No, אנחנו מבינים את התשובה במשמעות: P אינו יכול להיות מסופק.

במקרים מסויימים נרצה להגדיר פרדיקטים במונחים של שלילת פרדיקטים אחרים.

אחת הדרכים לעשות זאת היא על ידי שימוש ב-cut ובפרדיקט נוסף הבנוי בתור השפה – fail.

fail הוא פרדיקט מיוחד שתמיד נכשל. באופן דומה, true הוא פרדיקט שתמיד מצליח.

דוגמאות לשימוש: "Mary אוהבת את כל החיות חוץ מנחשים" ייכתב כך:

```
likes(mary, X) :-
      snake(X), !, fail;
      animal(X).
```

נגדיר פרדיקט diff שיחזיר אמת אם האיברים שהוא מקבל שונים ושקר אם הם זהים:

```
diff(X, X) :- !, fail.
diff(X, Y).
```

הפרדיקט not

המטרה Goal מצליחה אם not(Goal) נכשלת.

```
not(P) :- P, !, fail; true.
```

הפרדיקט not עצמו כבר בנוי בתוך שפת PROLOG, ומוגדר כאופרטור prefix. נוכל למשל לכתוב את הפרדיקט diff אותו כבר ראינו בצורה הבאה:

```
diff(X, Y) :- not (X, Y).
```

נעדיף במידת האפשר להשתמש ב-not במקום ב-cut&fail במידת האפשר. אופרטור זה שומר יותר על התוכנית כתוכנית לוגית. נשים לב כי אם המטרה not(A) מצליחה זה לא אומר כי A אינו אמת, אלא זה אומר כי "בהינתן בסיס הנתונים הנוכחי, לא ניתן להוכיח את A".

Prolog פועלת לפי "**הנחת העולם הסגור**" – השפה מניחה שבידה כל המידע הרלוונטי. אם לא ניתן להוכיח טענה, סימן שהיא אינה נכונה.