תכנות לוגי בפרולוג

ע"פ חוברת ההסברים של חסד-אומות-העולם

פרופ' אולה אנדריס

עם תרגום

"רי"ח טוב"

נערך בחסדי ה'

מאיתי יוחנן חאיק

שנת 01011010010010

Dear Yohanan,

I'm glad to hear you found my lecture notes useful and you are very
welcome to translate them. I would only like to ask you to include a
link to the original:

http://www.illc.uva.nl/~ulle/teaching/prolog/prolog.pdf

All the best,
Ulle

"כשם שאי אפשר לבר בלא תבן,
כך אי אפשר לתוכנית בלי שגיאות של הבודק האוטומטי"

להערות, הארות ותיקונים:
yohananha@gmail.com
בטלגרם - yohanan@
ניתן להשתמש בסיכום באופן חופשי לכולם!!
לסיכומים נוספים ועל מנת לוודא שיש בידכם את הגרסא המעודכנת ביותר - כנסו:

תוכן העניינים

פרק 1 – יסודות 3

A.1. הקדמה 3

A.2. מתחילים: דוגמא 3

A.3. סינטקס בפרולוג 5

1.1.1. ביטויים 5

1.1.2. טענה, תוכנית ושאילתות 6

1.1.3. מספר פרדיקטים מובנים במערכת 7

A.4. מענה לשאילתות 8

1.1.4. התאמה 9

1.1.5. ביצוע מטרה 10

1.5 עניין של סגנון 11

פרק 2 – רשימות 13

A.5. תיחום 13

A.6. ראש וזנב 13

A.7. מספר פרדיקטים מובנים ומניפלוציות על רשימות 15

פרק 3 – ביטויים אריתמטיים 17

A.8. אופרטור **is** להערכה אריתמטית 17

A.9. פונקציות ויחסים אריתמטיים מוגדרים מראש 18

פרק 4 – אופרטורים 19

A.10. קדימות ואסוציאטיביות 19

A.11. הגדרת אופרטורים בעזרת פרדיקט **op**. 21

פרק 5 – מעקב, CUT ושלילה 23

A.12. מעקב ו–**CUT** 23

A.13. מעקב 23

A.14. בעיות עם **Backtracking** 24

A.15. הכרות עם **CUT** 24

A.16. בעיות עם **CUT** 26

A.17. שלילה בכישלון 27

A.18. עולם ההנחות 27

A.19. אופרטור **+** 28

A.20. הפרדה 29

A.21. דוגמה: הערכה לוגית 30

פרק 6 – הבסיס הלוגי מאחורי פרולוג 32

A.22. תרגום של משפטי פרולוג לנוסחאות 32

נספח – רקורסיה 34

A.1. אינדוקציה..... 34
A.2. העיקרון הרקורסיבי..... 34
A.3. אילו בעיות ניתן לפתור..... 35
A.4. דיבאג..... 36

הבהרה:

החוברת שבידכם **איננה** סיכום שיעורים, אלא תרגום של החוברת ממנה נלקחו מצגות השיעור.

יש פה בחלק מהנושאים פירוט שלא דיברנו עליו בשיעור, וחסר פה חלק מהדברים החשובים ביותר (לבחינה לפחות) – צבעים של "קאט", וסוגי רקורסיות ועוד.

ראו הוזהרתם!

פרק 1 – יסודות

1.1. הקדמה

פרולוג – תכנות לוגי[1] היא אחת משפו התכנות הנפוצות ביותר בתחום מחקר הבינה המלאכותית. בניגוד לשפות אימפרטיביות (פקודתיות) כמו C או ג'אווה (שגם הן במקרה מונחות-עצמים), שפה זו הינה שפת תכנות *הצהרתית*. הכוונה היא, כאשר מיישמים פתרון לבעיה כלשהי, במקום לציין *כיצד* אנו הולכים להשיג מטרה מסוימת תחת התנאים נתונים, אנחנו ציינים *מה* המצב הנתון (*חוקים* ו*עובדות*) והמטרה בתוכנית היא לתת לאינטרפרטר של הפרולוג להסיק לנו את התוצאה. פרולוג שימושית מאוד בשדות מסוימים, כמו אינטליגנציה מלאכותית, לימוד שפה עצמאית, בסיסי נתונים... אבל חסרת שימוש בשדות אחרים כמו גרפיקה או אלגוריתמיים מספריים.

במהלך לימוד הקורס, נלמד בתחילה כיצד להשתמש בתכנות בפרולוג על מנת לפתור מספר בעיות במדעי המחשב ובינה מלאכותית, וכן נלמד כיצד המפענח של הפרולוג באמת עובד. החלק השני, יכלול בתוכו הצגה לבסיס הלוגיקה שעל פיו עובד הפרולוג.

מתחילים: דוגמא

בהקדמה אמרנו שפרולוג היא שפה הצהרתית (או תיאורית). תכנות בפרולוג משמעו בעצם לתאר את העולם. שימוש בתוכנת פרולוג, משמעותו לשאול את הפרולוג אלות בקשר לעולם הדיון שתואר קודם. הדרך הקלה ביותר לתאר את העולם הוא על ידי *עובדות* =, למשל:

```
bigger( elephant, horse).
```

הצהרות אלה, הינם אינטואיטיביות ופשוטות, במקרה זה – העובדה שפיל גדול יותר מסוס. (בהנחה כמובן שהעולם אותו אנחנו מתארים לפרולוג זהה לזה שבו אנו חיים, אך גם אחרת יכול להיות, תלוי כמובן במתכנת). כעת נוסיף עוד מספר הצהרות לתוכנית הקטנה שלנו:

```
bigger ( elephant , horse).
bigger ( horse , donkey).
bigger ( donkey, dog).
bigger ( donkey , monkey).
```

מבחינת הסינטקס מדובר בתוכנה תקינה לכל דבר, ואחר שנקמפל את התוכנית אנחנו נוכל לשאול את הפרולוג שאלות (או *שאילתות*[2], במונח המקצועי) לגבי ההצהרות השונות. למשל:

```
?- bigger (donkey, dog).
Yes
```

השאילתא bigger (donkey, dog), כלומר – השאלה "האם חמור גדול יותר מכלב?", עברה בהצלחה, מאחר וכבר קבענו זאת כעובדה שהוכרה כבר לתוכנית בשלב קודם. כעת נבדוק, האם קוף גדול יותר מפיל:

```
?- bigger ( monkey , elephant ).
No.
```

---

[1] Prolog – *programming in logic*

[2] *Queries*

אכן, זה לא נכון. אנחנו מקבלים בדיוק את התשובה לה ציפינו: השאילתא bigger (monkey , elephant). מחזירה תשובה שלילית. אך מה יקרה כאשר נשאל את אותה השאלה רק הפוך?

```
?- bigger (elephant , monkey).
No
```

לפי תוכנית הפרולוג, פילים אינם גדולים יותר מקופים. כמובן שזה טעות בכל קנה מידה, בהתחשב בעובדות של העולם האמיתי אליו אנחנו מתייחסים, אך אם נבדוק את התוכנית הקטנה שלנו שוב, נמצא שאין לנו שום הצהרה שיכול להצביע על איזשהו קשר ויחס בין הקופים לפילים. ועדיין, אנחנו יודעים בוודאות (ואף הצהרנו על זה בתוכנית) שפילים גדולים ותר מסוסים, והם בתורם, גדולים יותר מחמורים, שבהמשך אף הם גדולים יותר מקופים, ואזי, פילים גם הם צריכים להיות גדולים יותר מקופים. ובמונחם מתמטיים: יחס "גדול מ" הוא טרנזיטיבי. אך חוק שכזה לא הוגדר לנו בתוכנית. הפירוש המדויק של התשובה השלילית שהפרולוג ענה נו: מתוך כל האינפורמציה שניתנה בידי, אין לי יכולת להוכיח אם שאלה זאת נכונה או לא – אין ידע מה יותר גדול, פיל או קוף.

אם, בכל אופן, נרצה לקבל תשובה חיובית לשאלה ששאלנו .bigger (elephant , monkey), עלינו לספק מידע נוסף על מנת לגרום לעולם להיות יותר מדויק. דרך אחת תהיה להוסיף את שאר ההצהרות על מנת שיקיימו את כל הקומבינציות האפשריות, במקרה זה פשוט להוסיף את העובדה: , bigger (elephant .(monkey במקרה כזה, רק בשביל התוכנית הקטנה שבנינו, נצטרך להוסיף עוד 5 עובדות בשביל לסגור את כל האפשרויות. קל לראות שדבר זה הוא מסורבל ולא חכם במיוחד.

הפתרון המוצלח יותר יהיה להגדיר **יחס** חדש, שנקרא לו is_bigger, שיגדיר את היחס הטרנזיטיבי. חיה X תוגדר להיות גדולה יותר מחיה Y אם היא תוגדר להיות כזאת באחת מהעובדות שהוגדו בבסיס התוכנית, או במקרה בו קיימת חיה אחרת Z, הנמצאת ביניהם, המקיימת את התנאי שהוגדר באופן שחיה Z קטנה מX וגדולה מחיה Y. בפרולוג דברים אלו מוגדרים *חוקים* ומוגדרים באופן הבא:

```
is_bigger ( X, Y):- bigger(X,Y).
is_bigger (X , Y):- bigger (X , Z) , is_bigger (Z , Y).
```

הפרולוג במקרה זה עדיין לא יוכל להסיק את נכונות העובדה bigger (elephant , monkey) במסד הנתונים שלו, אז הוא ינסה את השיטה השנייה. דבר זה ייעשה על ידי *התאמה* של השאילתא עם ראש החוק, is_bigger (X , Y). בעת הביצוע שני המשתנים יאותחלו להיות: X = elephant, Y = monkey. החוק אומר שעל מנת לספק את החוק is_bigger (X , Y) (שעל ידי השת המשתנים שווה ערך ל is_bigger (elephant , monkey)) על הפרולוג להוכיח שתי הנחות משנה: 1. bigger (X , Z) .2. is_bigger (Z, Y). – במקביל להשמת המשתנים אותם הגדרנו. שאלה זאת נשאלת בצורה רקורסיבית עד לרמה בה ישנה שרשרת שמובילה היישר מהפיל כל הדרך על קוף וכל מחזורת השאילתא מצליחה. הדרך בה המטרה מושגת ודרך ההשמה ואיך פועל הכל יוסבר בצורה מפורטת בפרק 1.4.

כמובן, אנחנו יכולים לעשו גם דברים קצת יותר מרגשים מלשאול רק שאלות של כן ולא. נגיד ואנחנו רוצים לדעת *אילו* חיות גדולות יותר מחמור? השאילתא המתאימה תהיה:

```
?- is_bigger ( X, donkey).
```

שוב, X הוא משתנה. יכולנו גם לבחור כל שם משתנה אחר, ובלבד שיתחיל באות גדולה. האינטרפרטר יחזיר לנו את התשובה הבאה:

```
?- is_bigger ( X, donkey).
X = horse
```

סוס גדול יותר מחמור. השאילתא עברה בהצלחה, אך על מנת שהיא אכן תיושם באופן נכון הפרולוג נאלץ לעשות השמה של המשתנה X שיהיה עכשיו סוס. אם זה מספק את מה שאנחנו רוצים, אנחנו יכולים ללחוץ "חזור" ולסיים את התוכנית. במקרה בו אנחנו רוצים למצוא האם יש חיות נוספות שהינם גדולות יותר מחמור נלחץ על ; דבר שיגרום לפרולוג לחפש אלטרנטיבות נוספות לשאילתא זאת. אם נעשה זאת פעם אחת נקבל את הפתרון X = elephant. פילים גדולים יותר מחמורים. לחיצה נוספת על מקש זה תחזיר לנו No, וזאת מאחר ואין לנו עוד פתרונות.

```
?- is_bigger ( X, donkey).
X = horse ;
X = elephant ;
No
```

ישנם דרכים נוספות לשאול את המערכת לגבי בסיס הנתונים שלה. כדוגמא אחרונה, נשאל האם ישנה חיה המקיימת בו זמנית שני תנאים: קטנה מחמור *וגם* גדולה מקוף:

```
?- is_bigger ( donkey , X) , is_bigger ( X , monkey).
No
```

התשובה (הנכונה) היא לא. אף על פי שכל שאילתא בפני עצמה יכולה להחזיר לנו ערכים כלשהם בפני עצמם, חיתוך של האפשרויות (המיוצג על ידי הפסיק) לא מוצא תשובה מתאימה.

חלק זה נועד לתת רום ראשוני לגבי תכנות בפרולוג. החלק הבא מכיל ראייה סיסטמתית על הסינטקס. ישנם מספר אינטרפרטרים לפרולוג. פתיחת תוכנית עלולה להיות שונה במקצת בין מערכת אחת לחרת. פרטים לגבי זה יובאו בהמשך.

סינטקס בפרולוג

חלק זה בא לתאר את רוב האפשרויות הבסיסיות בשפת התכנות פרולוג.

1.1.1. ביטויים

מבנה המידע המרכזי בפרולוג הוא ה*ביטוי*. ישנם ביטויים מארבעה סוגים: *אטומים, מספרים, משתנים, וביטויים מורכבים*. אטומים ומספרים לעיתים מאוחדים ביחד ונקראים *ביטויים אטומים*.

אטומים

אטומים הינם בדרך כלל מחרוזות המורכבות מאותיות גדולות וקטנות, מספרים, וקו תחתון. כל הדוגמאות הבאות הינם אטומים חוקיים בשפה:

```
elephant, b, abcXYZ, x_123, another_pint_for_me_please
```

בנוסף לכך, כל שילוב שרירותי של תווים מאוחדים תחת סגירה אחידה נחשב אטום.

```
'This is also a Prolog atom.'
```

לבסוף, מחרוזות שמורכבות באופן בלעדי מתווים מסוימים כגון: + - * = < > : & נחשבים גם הם אטומים. לדוגמא:

```
+, :: , <------>, ***
```

מספרים

כל ממשקי הפרולוג השונים תומכים באינטג'רים (מספרים שלמים כולל מספרים הקטנים מס). בחלק ניתן גם להשתמש במספרי float.

משתנים

משתנים אם מחרוזות של תווים, אותיות מספרים וקו תחתון, המתחילים באות גדולה או בקו תחתון בראשית המילה. לדוגמא:

X , Elephant , _4711 , X_1_2 , MyVariable, _

הדוגמא האחרונה המכילה רק קו תחתון, הינה מקרה מיוחד. משתנה זה נקרא *משתנה אנונימי* והוא בא לידי שימוש כמשתנה שאין לנו שום עניים מסוים במה הוא מכיל (don't care) באר מכניסים את המשתנה הזה במקומות מסוימים אנחנו בעצם מתעלמים מכל מה שיש שם, אפילו א מדובר בלעשות את זה מספר פעמים תחת רכים שהם שונים לגמרי. דבר זה יורחב בהמשך.

ביטויים מורכבים

ביטויים אלה נוצרים בעזרת פקטור (אטום) ומספר *טיעונים* (ביטויים בפרולוג. למשל, אטומים, מספרים או אפילו ביטויים מורכבים אחרים) סגורים תחת הכרזה כללית (טיעון אב). כל הביטויים הבאים הם ביטויים מורכבים:

Is_bigger(horse , X) , f(g (X , _), 7), 'My Functor'(dog)

חשוב ביותר לא לשים רווחים בין ההכרזה לאיחוד הטיעונים, או שהפרולוג לא יבין מה אנחנו מנסים לומר. במקרים אחרים, רווחים דווקא יעזרו לנו לעשות את התוכנה קריאה יותר, אך זה לא המקרה פה.

שילוב של ביטויים ורכסים ואטומים ביחד מייצרים את ה*פרדיקטים*.

ביטוי שאינו מכיל שום משתנה נקרא *ביטוי בסיסי* .

1.1.2. טענה, תוכנית ושאילתות

בפרק ההקדמה כבר ראינו כיצד בונים תוכנית פרולוג בעזרת *עובדות* ו*חוקים*. עובדות וחוקים נקראים גם *טענות*.

עובדות

עובדה היא פרדיקט שבסופו מונחת נקודה. לדוגמה:

bigger(whale, _).

Life_is_beutiful.

המשמעות האינטואיטיבית של עובדה היא שאנחנו מגדירים איזשהו קיום של יחס שהינו אמת.

חוקים

חוק מורכב מ*ראש* (פרדיקט) ו*גוף* (רצף של פרדיקטים מופרים על ידי פסיק). הראש והגוף מופרדים ביניהם על ידי הסימן –: וכמו כל ביטוי אחר בפרולוג, מסיימים את החוק על ידי הנחת נקודה בסוף המשפט. דוגמה:

```
is_smaller ( X , Y) :- is_bigger ( Y , X).
aunt (Aunt , Parent) :-
```

sister (Aunt , Parent),
parent (Parent , Child).

המשמעות האינטואיטיבית של חוק הוא שראש החוק מוגדר להיות אמת, אם נוכל להראות, שכל הביטויים (תתי המשימות) בגוף החוק גם הם אמת.

תוכנית

תוכנית פרולוג היא אוסף של טענות.

שאילתות

לאחר קומפילציה של תוכנית פרולוג, ההרצה שלה מתבצעת על ידי הרצת שאילתות לאינטרפרטר. לשאילתא יש את אותו מבנה של גוף החוק, כלומר, הוא אוסף חוקים מופרד על ידי פסיק ובסופו יש נקודה. ניתן לשאול את השאילתא בחלון הפקודה של פרולוג, כאשר המימוש הנפוץ ביתר הוא על ידי הסימון –? . כאשר אנו כותבים שאילתות אנחנו לרוב משתמשים בצורה הזו. למשל:

?- is_bigger (elephant , donkey).
?- small (X), green (X), slimy (X).

באופן אינטואיטיבי, כאשר מציבים שאילתא באופן דומה לדוגמה השנייה, אנחנו שואלים את פרולוג האם כל המשתנים מוכחים להיות אמת, א במילים אחרות האם ישנו X בו כל התנאים small (X), green (X), slimy (X) יתבררו להיות נכונים.

1.1.3. מספר פרדיקטים מובנים במערכת

מה שראינו עד כה מאפשר לנו ליצור תוכנית פשוטה על יד הגדרה של חוקים ועובדות, אבל הפרולוג מספק לנו גם טווח של פרדיקטים מובנים במערכת בהם ניתן להשתמש. חלק מהם יוצגו בפרק הזה, ולהרחבה לכל הפרדיקטים ניתן לראות את המדריך באתר.

הפרדיקטים המובנים ניתנים לשימוש באופן דומה לאלו המוגדרים ע ידי המשתמש. ההבדל החשוב ביניהם הוא, שבפרדיקט מובנה לא ניתן להשתמש בצורת פונקציה או עובדה או ראש של חוק. זה חייב להישאר ככה מאחר ושינוי בהם עלול להשפיע באופן שישנה להם את כל ההגדרה.

הפרדיקט שהוא אולי החשוב ביותר הוא = (שוויון). במקום לכתוב .(X , Y)= אנחנו כותבים בדרך כלל בצורה נוחה יותר Y=X. הצבה זו מקבלת אישור במידה והערך ב X והערך בY אכן מתאימים זה לזה. נדייק בכל זה בחלק 1.4.

לפעמים יהיה שימושי לקחת דווקא פרדיקטים שידועים לנו כבאלה שייכשלו או יצליחו בכל מצב. למצבים כאלה באופן פשוט יש לנו את הפרדיקטים true ומקבילו false.

קבצי תכונית מסוימת יכולים להתקמפל על ידי שימוש בפרדיקט consult/1[3]. הארגומנט הנשלח הוא חייב להיות קובץ מסוים פרולוג. למשל, על מנת לקמפל את הקובץ big_animals.pl עלינו לכתוב את הפרדיקט הבא:

?- consult ('big-animal.pl').

אם הקומפילציה עוברת בהצלחה, הפרולוג יגיב ב.Yes אחרת תופיע רשימה של שגיאות.

[3] הביטוי 1/ משמש לסמן את מספר הארגומנטים שהפרדיקט מקבל.

אם אתה מעוניין בפלט נוסף מלבד זה המובנה בפרולוג, ניתן להשתמש בפרדיקט write/1. כאשר הארגומנט יכול להיות כל ביטוי חוקי בפרולוג. במידה ומדובר במשתנה, אזי יודפס החוצה הערך שהוא מכיל. הוצאה לפועל של פרדיקט nl/0 גורם לתוכנית לקפוץ שורה לאחר ההדפסה. להלן שתי דוגמאות:

```
?- write ( 'Hello World!' ), nl.
Hello World!
Yes

?- X = elephant, write (X), nl.
elephant
X = elephant
Yes
```

בדוגמה השניה אנחנו קודם ל קשרנו את המשתנה X עם האטום elephant ואז ביקשנו להדפיס את הערך X (הווה אומר, elephant), בעזרת הפרדיקט write/1. לאחר שהפרולוג מוריד שורה, הוא מדווח על השמת המשתנה X והכבילה שלו X = elephant.

בדיקת סוג ביטוי בפרולוג.

ישנם מספר פרדיקטים מובנים שתפקידם הוא לבדוק את סוג הביטוי המתאים להגדרות פרולוג. להלן מספר דוגמאות:

```
?- atom ( elephant ).
Yes

?- atom (Elephant).
No

?- X = f(mouse), compound (X).
X = f(mouse)
Yes
```

השאילתא האחרונה מצליחה, מאחר והמשתנה X מקושר לביטוי המורכב f(mouse), ברגע שהפרדיקט compound(X), מבוצע.

רוב מערכות הפרולוג מספקות גם עזרה בצורת פרדיקטים, בדרך כלל תחת help/1. ברגע שמציבים אותו על ביטוי כלשהו (כמו שם של פרדיקט מובנה) המערכת תפלוט לנו תיאור קצר של הפרדיקט (במידה ואפשרי). למשל:

```
?- help (atom).
atom (+Term)
        Succeeds if Term is bound to an atom.
```

1.4 מענה לשאילתות

הזכרנו כבר את עניין התאמת הביטויים בחלקים קודמים. נושא זה הינו קריטי להבנה ששל איך פרולוג עונה לשאילתות, ולכן תיארנו זאת עוד לפני שהסברנו מה באמת קורה כשר שאילתא מבוצעת (או באופן כללי יותר: כאשר מוציאים לפועל את המטרה).

1.5.1. התאמה

אנו אמורים ששני ביטויים הינם *מתאימים* במידה והם זהים, או לחילופין ניתן להפוך אום להיות זהים על ידי משחק של השמת משתנים. השמת משתנים פירושה הצבת של ערך קבוע וברור. לאור זאת, שני משתנים ריקים גם הם מתאימים, מאחר והבסיס שלהם הוא שווה.

חשו לציין שמשתנה חייב להישאר אחיד בצורתו לאורך כל הביטוי. יוצא הדופן היחיד לחוק זה הוא ה*משתנה האנונימי* _, שהוא ייחודי כך שהוא יכול לקבל מספר שונה של ערכים וטיפוסים.

נתנו גם מספר דוגמאות. הביטוי is_bigger (X , dog) וכן הביטוי is_bigger (elephant , dog) מתאימים, מאחר והמשתנה X מוצה עם הערך elephant. אנחנו יכולים לבדוק זאת על ידי השמה של שני הביטויים האלה עם הסימן = ביניהם ולראות כיצד הפרולוג יציב את המשתנה:

```
?- is_bigger ( X, dog) = is_bigger (elephant, dog).
X = elephant
Yes
```

הדוגמה הבאה היא שאילתא שאינה מתקבלת, וזאת מאחר ו-X לא יכול להיות שווה גם ל-1 וגם ל-2 בו זמנית.

```
?- p( X, 2, 2) = p( 1, Y , _).
No
```

מצד שני, אם במקום X אנחנו נחליט להשתמש במשתנה האנונימי _, ההתאמה אפשרית, בגלל של מופע של _ מוצג בתור משתנה נפרד. וכל מה שמושם הוא הערך ב Y המוצב להיות 2:

```
?- p( _, 2, 2) = p( 1, Y, _ ).
Y = 2
Yes
```

דוגמה נוספת להתאמה:

```
?- f( a, g( X, Y)) = f( X, Z),  Z = g( W, h(X)).
X = a
Y = h(a)
Z = g( a, h(a))
W = a
Yes
```

עד כאן הכל בסדר. אך האם יכו להיות מקרה בו התאמה תצא לפועל גם במקרה שלא נציב ערכים במשתנים עצמם? נסתכל על הדוגמה הבאה:

```
?- X = my_functor ( Y).
X = my_functor (_G177)
Y = _G177
Yes
```

בדוגמה זו ההתאמה גם כן מצליחה, וזאת מאחר שאת X אנחנו מציבים להיות הפונקציה my_functor וזה הארגומנט שיוגדר עליו ביחד עם עוד משתנה פנימי שאינו מוגדר. Y כעת יוכל להיות כל ביטוי חוקי בפרולוג, אבל הוא חייב להיות מתאים לזה שמוצב בתוך ה-X. בפרולוג שימוש כזה מבוצע על ידי ערך

דומה ל G177_ שהוא מעין ערך ברירת מחדל שמחושב בכל פעם מחדש בזמן ריצה. השם הספציפי G177_ יחושב בכל פעם מחדש לשם טיפה שונה.

1.5.2. ביצוע מטרה

הגשת שאילתא משמעותה בקשה מהפרולוג לנסות ולהוכיח טענה כלשהי (X), המבוצעת על ידי בדיקה של טענות אמת המוצגות מראש והשמה של ערכים מתאימים. החיפוש אחר ההוכחה נקרא *ביצוע המטרה*. כל פרדיקט בשאילתא מכיל (תת) מטרה, שהפרולוג מנסה לספק בצורה כזו או אחרת. אם משתנה מפוזר על פני כמה תתי-ביטויים עליו להיות מושם בצורה אחידה לאורך כל הדרך.

אם המטרה להתאמה נמצאת בראש החוק, השמת המשתנים מבוצעת בתוך גוף החוק, מה שהופך להיות בתורו המטרה לסיפוק. אם הגוף מורכב ממספר פרדיקטים המטרה שוב מתפצלת בכל פעם ומספקת כל חלק בתורו. במילים אחרות, על מנת להוכיח שהראש אכן הוא ביטוי אמת, עלינו להוכיח שגם כל החלקים של הגוף הינם אמת.

אם בזמן חיפוש הביצוע נתקלים בעובדה המאששת לנו אן הביטוי הוא אמת מוחלטת, אנחנו חוזרים ישר באותו רגע ומתקשרים למבצע ומדוחים לו על התוצאה.

יש לשים לב, שסדר העובדות הוא שוב בשביל לבצע נכון את הפעולות. פרולוג תמיד ינסה להתאים קודם כל את הפרדיקט הנמצא אל מולו ורק אחר כך ימשיך הלאה.

אם הפונקציה שאנו מחפשים היא פרדיקט מובנה, הפעולה הקשורה אליו תבוצע ברגע שהמטרה תושלם. לדוגמה: כבל שמעניין את המערכת, הפרדיקט

```
write ('Hello World!')
```

יצליח באופן פשוט, כי זה נכון, ובאותו זמן הוא גם ידפיס למסך את ההודעה !Hello World.

כמו שנאמר כבר קודם, הפרדיקט המובנה true יתקיים תמיד (ללא כל תופעות לוואי כלשהן), בזמן שהפרדיקט fail תמיד ייכשל.

לפעמים ישנה יותר מדרך אחת על מנת לספק את המטרה. הפרולוג בוחר את האפשרות הראשונה (כפי שנקבע על ידי סדר כתיבת התוכנית), למרות שיכול להיות אלטרנטיבות נוספות בהמשך שיגיעו לתוצאה דומה. אם לחילופין הפרולוג ייכשל תחת אחד התהליכים, הוא לא יעצור, אלא יחוור למקום הבטוח האחרון ומשם ינסה להמשיך לכיוונים אחרים. תהליך זה נקרא *backtracking*.

נוכל לעקוב אחרי תהליך אימות על ידי המשפט המפורסם הבא:

All men are mortal.
Socrates is a man.
Hence, Socrates is mortal.

במונחים של פרולוג, המשפט הראשון מופיע כחוק X הוא בן-מוות, אם X הוא בן-אדם (עבור כל X). המשפט השני קובע עובדה: סוקרטס הוא אדם. את שני אלו ניתן לכתוב בפרולוג באופן הבא:

```
mortal (X) :- man (X).
man (Socrates).
```

יש לשים לב, ש-X הוא משנה. בעוד שסוקרטוס הוא אטום. המסקנה של הטיעונים הם ש סוקרטס הוא בן-מוות, יכול לבוא לידי ביטוי באופן אחר – mortal(Socrates). לאחר התייעצות עם התוכנית שכתבנו למעלה אנחנו יכולים לשאול את הפרולוג שאילתא, שתוציא את הפלט הבא:

```
?- mortal ( socrates).
Yes
```

הפרולוג מסכים עם ההיגיון הפשוט שלנו – שזה נחמד. אבל איך הוא הגיע למסקנה הזאת? נעקוב אחרי הביצוע צעד אחר צעד.

1. השאילתא mortal (socrates) נעשית המטרה אותה אנחנו רוצים לספק.
2. בסריקה על התוכנית שלנו, הפרולוג מנסה למצוא משהו שיתאים לטיעון של mortal (socrates) באפשרות הראשונה שנקרית בדרכו או בראש החוק. הוא מוצא את mortal (X), הראש של החוק הראשון (והיחיד). כאשר מאחדים את שני התנאים, ההשמה X =socrates צריכה להתקיים.
3. השמת המשתנה נבדקת עבור גוף החוק, כלומר, man(X) הופך להיות man(socrates).
4. ההשמה החדשה שלנו הופכת להיות המטרה החדשה שלנו: man(socrates).
5. פרולוג מנסה ליישם את המטרה החדשה שלו על ידי בדיקה מול כלל הראש או עובדה אחרת כל שהיא. ברור שנמצא כי המטרה שלנו man(socrates) מתאימה באורח פלא לעובדה man(socrates) וזאת מאחר והם זהים. וזה אומר שהמטרה הושגה.
6. דבר זה מחזיר לנו שכל השאילתא אכן נכונה.

1.5 עניין של סגנון

אחד היתרונות הגדולים בפרולוג, שהיא נותנת לנו לכתוב תוכנית קצרות וקומפקטיות שפותרות לא רק תוכניות מסובכות, אלא גם נשארות קריאות (כמובן: יחסית) וקלות להבנה.

כמובן שזה יכול לקרות רק בתנאי שהמתכנת (את/ה!) מתייחס לסגנון כתיבת הקוד. וכמו בכל שפה, גם כאן הערות עוזרות לנו. בפרולוג הערות נסגרות בין /* */ בצורה כזו:

```
/* This is a comment! */
```

הערות שרצות רק על פני שורה אחת יכול גם להתחיל על ידי סימן אחוזים %. דבר זה נעשה בדרך כלל בסימון של שורה בתוך הקוד:

```
aunt ( X, Z) :-
        sister (X, Y), %A comment on this subgoal
        parent (Y, Z).
```

מלבד שימוש בהערות, פריסה נכונה של הקוד תשפר גם את הקריאות באופן משמעותי. החוקים הבאים הם הבסיס עליהם מסכימים מרבי האנשים:

1. תפריד ביטויים על ידי רווח שורה אחד או שניים.
2. תכתוב פרדיקט אחד בלבד לכל שורה ותשתמש בריווח:

```
blond(X) :-
    father (X, Father),
    blond (Father),
    mother (X, mother).
    blond (mother).
```

(ביטויי קצרים כדאי להם שייכתבו גם הם בשורה נפרדת.)

3. תכניס רווח אחרי כל פסיק, ובתוך סוגריים בין כל המשתנים:

```
born ( mary , Yorkshire , '01/01/1980').
```

4. תכתוב ביטויים קצרים המורכבים ממספר קטן של תתי משימות. במידת הצורך הפרד את הביטוי לשניים שונים או יותר.
5. בחר שמות משמעותיים עבור האטומים והמשתנים השונים.

פרק 2 – רשימות

פרק זה מכיל מספר הערות חשובות גבי רשימות, אחד ממבני הנתונים הנוחים ביותר בפרולוג, ומכיל מספר דוגמאות כיצד לעבוד איתן.

תיחום

רשימות מוכלות בתוך סוגריים מרובעות כאשר האלמנטים השונים בתוכם מופרדים על ידי פסיקים. להלן דוגמה:

[elephant , horse , donkey , dog]

זוהי רשימה של ארבעה אטומי שונים: elephant , horse , donkey , dog. אלמנטים בתוך רשימה יכולים להיות כל ביטוי חוקי בפרולוג, למשל, אטומים, מספרים משתנים, או ביטויים מורכבים. וזה כולל גם רשימות אחרות. רשימה ריקה נכתבת באופן כזה – []. הדוגמא הבאה היא גם כן דוגמה של רשימה (קצת יותר) מורכבת:

[elephant , [] , X , parent(X , tom), [a , b , c] , f(22)]

ייצוג פנימי

רשימות פנימיות מיוצגות כביטויים מורכבים המאוחדים על ידי נקודה. הרשימה הריקה [] היא אטום שאליו מוכנסים הערכים אחד לאחר השני. הרשימה [a, b, c] יכולה להיכתב גם באופן הבא:

.(a, .(b, .(c, [])))

ראש וזנב

האיבר הראשון ברשימה נקרא *ראש* וכל שאר הרשימה מוגדר *זנב*. לרשימה ריקה אין זנב. לרשימה המכילה רק איבר אחד יוגדר האיבר הבודד בתור ראש הרשימה, והזנב יוגדר להיות רשימה ריקה.

פיצול הרשימה מתבצע בצורה נוחה במקביל גם לראש וגם לזנב. דבר זה נעשה בצורה נוחה על ידי שימוש בקו המפריד |. בכל מקום בו הוא מונח ברשימה, הוא מגדיר את כל החלק שאחריו להיות רשימה חדשה. הרשימה החדשה נבנית בצורה שתת הרשימה לפני הקו יהיה הראש. במידה ויש רק איבר אחד לפני הקו הוא יהיה הראש והשאר יהיה הזנב. הדוגמה הבאה המספר 1 יהיה הראש, והרשימה [2,3,4,5] יהיה הזנב, שחושב בקלות בפרולוג בצורה פשוטה של תבנית ראש|זנב:

```
?- [1,2,3,4,5] = [ Head | Tail].
Head = 1
Tail = [2,3,4,5]
Yes
```

יש לשים לב, שבמקרה זה הHead ו–Tail הם שמות של משתנים. אנחנו יכולים להשתמש גם בX או Y או כל דבר אחר שנרצה להגדיר בעצמנו. חשוב גם להדגיש שזנב הרשימה (או ליתר דיוק: החלק המופיע לאחר הסימון |) הוא תמיד יהווה רשימה בעצמו. יכול להיות שזה יהיה רשימה ריקה, אך גם זה בהגדרה רשימה. הראש, בכל אופן, הוא איבר ברשימה. הוא יכול להיות רשימה בעצמו, אך דבר זה לא מוכרח (כמו שניתן לראות בדוגמה מעל, בה הראש היה פשוט איבר בודד 1). אותו דבר חל על כל איבר שיופיע לפני הקו המפריד.

צורה זאת מאפשרת לנו, למשל, לקבל גם את האיבר השני ברשימה. בדוגמה הבאה אנחנו משתמשים במשתנה אנונימי גם עבור האיבר הראשון ברשימה וגם עבור הזנב, מאחר ואנחנו מעוניינים אך ורק באיבר השני:

```
?- [ quod , licet , jovu , non , licet , bovi ] = [_, X | _ ].
X = licet
Yes
```

תבנית הראש | זנב מאפשרת לנו ליצור פרדיקטים ברשימה בצורה מאוד מתוחכמת. נדגים זאת על ידי שימוש בפרדיקט המובנה המשמש לאיחוד שני רשימות[4]. אנחנו נקרא לפרדיקט concat_lists/3 כאשר פרדיקט זה נקרא, אנחנו רוצים לאחד שתי רשימות קיימות המופיעות ראשונות אל תוך המשתנה השלישי המוגדר. במילים אחרות, נרצה ליצור משהו שייראה כך:

```
?- concat_lists ( [ 1, 2, 3] , [d, e, f, g] , X).
X = [1, 2, 3, d, e, f, g ]
Yes
```

הגישה הכללית לפתרון בעיה מסוג כזו היא ל ידי רקורסיה. אנחנו מתחילים עם תנאי בסיס ואז בונים ומפרטים את שאר המקרים על מנת לפרק את הבעיה האחת המורכבת למספר בעיות קטנות ופשוטות יותר עד שנגיע למקרה הבסיס. למקרה שלנו המקרה הבסיסי יהיה כאשר אחת מהרשימות (למשל בדוגמה שהבאנו, הרשימה הראשונה) תהיה ריקה. במקרה זה, התוצאה (הארגומנט השלישי) יהפוך להיות זהה בדיוק לרשימה הנותרת (השנייה). דבר זה יכול לבוא לידי ביטוי באופן הבא:

```
concat_lists ( [] , List , List).
```

לכל שאר המקרים (כלומר כל המקרים בהם השאילתא של ה concat_lists/3 לא מתאימה לעובדה זאת) ברשימה הראשונה יש לפחות איבר אחד. ולכן, ניתן לכתוב אותה בצורה הבאה כתבנית של ראש | זנב: [Elem| List1]. בהנחה שאת הרשימה הבאה נגדיר להיות List2, אז נדע שהראש הוא Elem ושאר הרשימה הוא List1 או List2 בהתאמה. שים לב כיצד זה מפשט לנו את הבעיה: אנחנו לוקחים את האיבר הראשון ברשימה הראשונה ומנסים לחבר אותו עם הרשימה השנייה (שלא עברה שום שינוי כלל). אם נחזור על פעולה זו באופן רקורסיבי, לבסוף נגיע לכך שהרשימה הראשונה תהיה ריקה, שזה בדיוק מקרה הבסיס שיכול להיות מטופל על ידי העובדה שהגדרנו קודם. כעת נהפוך את פישוט האלגוריתם המילולי שעשינו לכדי תוכנית בפרולוג:

```
concat_lists( [Elem | List1], List2, [Elem | List3]) :-
        concat_lists( List1, List2, List 3).
```

וזהו! הפונקציה concat_lists/3 כעת מובנה לשימוש ויכולה לאחד שני רשימות לאחת חדשה. אבל למעשה מה שעשינו הרב יותר גמיש ממה שחשבנו. אם נקרא לה כעת כאשר נגדיר את שני הארגומנטים הראשונים להיות משתנים ואת האיבר השלישי להיות רשימה, פרולוג ידפיס לנו את כל האפשרויות כיצד הרשימות הנתונות יכולות לצאת לפועל.

```
?- concat_lists ( X, Y, [a, b, c, d]).

X = []
Y = [a, b, c, d];
```

---

[4] ברוב סביבות העבודה של פרולוג ישנו את הפרדיקו המובנה הזה תחת השם append/3 (יופיע בהתמשך)

```
X = [a]
Y = [b, c, d];

X = [a, b]
Y = [c, d];

X = [a, b, c]
Y = [d];

X = [a, b, c, d]
Y = [];

No
```

שים לב שה No בסוף הפלט משמעו שאין עוד אלטרנטיבות לשאילתא שהגדרנו.

מספר פרדיקטים מובנים ומניפולציות על רשימות

פרולוג מגיע עם טווח רחב של אפשרויות מוגדרות מראש על מנת ליצור מניפולציות על רשימות. חלק מהפרדיקטים החשובים ביותר יוצגו כאן, אך יש לשים לב שאת כולם ניתן ליצור על ידי שימוש בצורת הראש|זנב.

length/2

הארגומנט השני יוצב להיות אורך הרשימה שנמצא בארגומנט הראשון. למשל:

```
?- length( [elephant ,[], [1, 2, 3]], Length).
Length = 3
Yes
```

ניתן להשתמש בפונקציה זאת גם על ידי שימוש ברשימה לא מאותחלת. דבר זה יחשב את המקומות הריקים על פי האורך שהגדרנו:

```
?- length( List, 3).
List = [_G228, _G251, _G254]
Yes
```

השמות של המשתנים הללו יהיו שונים בכל פעם שנקרא לשאילתא הזאת מחדש מאחר שהם מחושבים בזמן ריצה ברגע שפרולוג מתחיל לעבוד.

member/2

היעד member(Elem, List) יושג, במידה והאיבר Elem יימצא מתאים לאחד האיברים הנמצאים ברשימה List. לדוגמה:

```
?- member( dog, [elephant, horse, donkey, dog, monkey]).
Yes
```

append/3

מחבר שני רשימות. פונקציה זאת עובדת בדיוק כמו שהגדרנו את concat_lists/3 בחלק הקודם.

last/2

פרדיקט זה מקבל ערך אמת, במידה והאלמנט המתקבל הוא האיבר האחרון ברשימה המוצעת כארגומנט השני של last/2.

reverse/2

פרדיקט זה משמש להיפוך סדר אלמנטים בתוך רשימה. הארגומנט הראשון צריך להיות רשימה (מאותחלת) והאיבר השני משתנה, כך שהוא יוחזר בתור הרשימה ההפוכה. לדוגמה:

```
?- reverse( [1, 2, 3, 4, 5], X).
X = [5, 4, 3, 2, 1]
Yes
```

select/3

בהינתן רשימה ואיבר אחד המופיע בתוכה, המשתנה שמופיע שלישי, יאותחל להיות שאר הרשימה ללא אותו איבר. לדוגמה:

```
?- select( [mouse, bird, jellyfish, zebra], bird, X).
X = [mouse, jellyfish, zebra]
Yes
```

פרק 3 – ביטויים אריתמטיים

אם ניסית להשתמש במספרים בפרולוג ודאי כבר נתקלת בכל מיני התנהגויות לא צפויות של המערכת. החלק הראשון של הפרק הזה מסביר את התופעה הזאת. לאחר מכן ניתן הבר על שאר האפשרויות האריתמטיות הקיימות לנו במערכת.

אופרטור is להערכה אריתמטית

ביטויים אריתמטיים פשוטים כגון + או * הינם, כמו שנאמר כבר קודם, נחשבים בתור ביטויים אטומיים בפרולוג. מסיבה זאת, גם ביטוי כמו (3, 5)+ הינו ביטוי חוקי בפרולוג. באופן נוח יותר, ניתן לרשום אותם גם בצורת infix כמו 3+5.

אם לא נאמר לפרולוג בצורה מפורשת שאנחנו מעוניינים בהערכה אריתמטית של ביטוי, הוא יסתכל על ביטוי בצורה הפשוטה ביותר. הכוונה היא שסימון של = לא יעבוד בצורה לה ציפית:

```
?- 3+5 = 8
No
```

הביטויים 3+5 ו-8 *אינם תואמים* – הביטוי הראשון הינו ביטוי מורכב, והשני הוא מספר. על מנת לבדו האם הסכום של 3 ו-5 שווה אכן 8, אנחנו צריכים להגיד קודם כל לפרולוג לתת הערכה אריתמטית לשאלה "כמה זה 3+5?". דבר זה נעשה על ידי שימוש באופרטור ההערכה האריתמטית – is. אנחנו יכולים להשתמש בו גם על מנת לתת השמה עבור משתנה בתוצאת הביטוי האריתמטי. לאחר מכן אנחנו יכולים להשוות את התוצאה שבמשתנה למספר אחר. נכתוב מחדש את הביטוי הקודם באופן נכון:

```
?- X is 3+5, X = 8.
X = 8
Yes
```

אנחנו יכולים לבדוק את נכונות תרגיל החיבור פשוט על ידי הצבה של הספרה ישירות ללא המשתנה מצד לשמאל לאופרטור ה-is באופן הבא:

```
?- 8 is 3+5.
Yes
```

אך יש לשים לב, שדבר זה הינו רק חד-כיווני:

```
?- 3+5 is 8.
No
```

דבר זה קורה מאחר והאופרטור is עושה הערכה רק של הביטוי הנמצא לימינו, ואז מנסה לבדוק אם זה מתאים גם לביטוי השמאלי. ההערכה האריתמטית של 8 מחזירה 8, מה שלא מתאים (לא מוערך) לביטוי הפרולוג 3+5.

לסיכום, אופרטור is מוגדר באופן הבא: הוא מקבל שני ארגומנטים, שבו הביטוי השני הינו ביטוי אריתמטי בו כל המשתנים מאותחלים. הביטוי הראשון צריך להיות מספר, או לחילופין משתנה המייצג ספר. הקריאה מצליחה, אם ההערכה של הביטוי השני מתאימה לזה של הארגומנט הראשון (או במקרה שבו המספר הראשון הינו מספר, אם הם זהים).

שים לב שהתוצאה של החישוב האריתמטי תהיה במידת האפשר מספר אינטג'רי שלם (לפחות ב-SWI Prolog). מה שאומר, לדוגמא, שאם נשאל לגבי 1.0 is 0.5+0.5 נקבל תשובה שלילית, מאחר והתוצאה של

0.5+0.5 תחזיר את הערך 1 ולא את הערך 1.0. בגדול, עדיך להשתמש באופרטור =:= (שיוצג בהמשך הפרק) במקרה שבו הביטוי השמאלי מוגדר כבר כמספר.

פונקציות ויחסים אריתמטיים מוגדרים מראש

האופרטורים האריתמטיים בפרולוג יכולים להתחלק בין *פונקציות* לבין *יחסים*. חלק מהם יוצגו כעת, אך בשביל הרשימה המלאה ניתן לראות במדריך הרשמי של הפרולוג.

פונקציות

חיבור או הכפלה הם דוגמאות נפוצות של פונקציה מתמטית. בפרולוג כל הפונקציות האלו נכתבות באופן הטבע ביותר. הביטוי הבא מציג לנו דוגמה:

```
2 + (-3.2 * X - max( 17, X)) / 2**5
```

הביטוי max/2 נותן לנו הערכה של המספר הגדול בין שני ערכים נתונים, והביטוי 2**5 הינו בפשטות בזקת 5. פונקציות מוכללות נוספות כוללות את min/2 (מינימום)ת abs/1 (ערך מוחלט), sqrt/1 (שורש ריבועי), ו – sin/1 (סינוס)[5]. האופרטור // משמש לחלוקה שלמה (ללא שארית). על מנת לקבל את השארית של החלוקה (מודולו) נשתמש באופרטור mod. סדר קדימויות של ביטויים הינו בדיוק כמו שאנו מכירים מהחיים, לדוגמה, 4+3*2 שווה ערך ל4(3*2) וכו'.

ניתן להשתמש בround/1 על מנת לעגל מספר צף למספר השלם הקרוב אליו, וכן על מנת להפוך מספר שלם לתצוגה של מספר צף.

כל הפונקציות האלו יכולות לשמש בתור צד ימין של הביטוי is.

יחסים

יחסים אריתמטיים משמשים להשוואה אריתמטית של שני ביטויי. הביטוי X>Y, למשל, יצליח במידה והערך המושם ב X מוערך להיות גדול באמת מהביטוי Y. שים לב שהביטוי is אינו נצרך פה. הארגומנט האריתמטי מוערך במקביל לשימוש בבדיקת היחס בין המשתנים.

מלבד הביטוי < קיים כמובן גם הערך ההפוך לו < (קטן מ), >= (קטן או שווה), =< (גדול או שווה), =\= (לא שווה), ו-=:= (שיוויון מתמטי) כביטויים לכל דבר. ההבדל בין =:= ובין = הוא קריטי הראשון מבצע הערכה מתמטית ובודק האם הערכים שווים, בעוד השני מבצע הערכה האם הביטויים דומים. להבהרת העניין:

```
?- 2**3 =:= 3+5
Yes

?- 2**3 = 3+5
Mo
```

ניתן לראות שבניגוד לאופרטור is, כאן ההשוואה המתמטית =:= כן פועלת גם במידה ואחד המספרים יהיה שלם והשני יהיה מספר צף (בהנחה שהם באמת אותו מספר.

[5] בדומה לדוגמה עם max/2 כל הפונקציות האלה אינם נכתבים בצורה טבעית את כפונקציות

פרק 4 – אופרטורים

בפרק הקודם על אריתמטיקה ראינו כבר מספר אופרטורים. חלק מהפרדיקטים המיוחסים לפעולות אריתמטיות גם הם אופרטורים מודרים מראש. בפרק זה נתעסק כיצד להגדיר בעצמנו אופרטורים, שבהם נוכל להשתמש אחר כך בתור פרדיקטים עצמאיים.

קדימות ואסוציאטיביות

קדימות

אנחנו מכירים כבר מהאריתמטיקה וגם מלוגיקה *שקדימות* של אופרטור קובעת אך יפורש הביטוי. לדוגמה שהאופרטור הלוגי ∧ חזק יותר וקודם לאופרטור ∨, מה שגורם לכך שהנוסחה P∨Q∧R מקבלת את הפירוש P∨(Q∧R) ולא להיפך. בפרולוג נאמר שלהפרדה יש יותר כוח מאיחוד.

בפרולוג כל אופרטור מקושר למספר אינטג'רי (ב- SWI-prolog למשל המספרים בין 0 ל 1200) הקובעים את רמת הקדימות. ככל שמספר נמוך יותר, כך הוא גבוה יותר ברמת העדיפות האופרטור האריתמטי *, לדוגמה, מקבל את ערך הקדימות 400, בעוד ש+ מקבל 500. מסיבה זאת, אם נקבל בפרולוג את הביטוי 2+3*5 פרולוג יחשב קודם את הכפל של 3 ו-5 ורק לאחר מכן יוסיף את הערך 2.

קדימות של ביטוי תוגדר בברירת מחדל כ-0, אלא אם הפרולוג יראה שיש סיבה להציב לו ערך אחר של קדימות. לדוגמה:

- הקדימות של 3+5 היא 500.
- הקדימות של 3*5+5*3 גם היא 500.
- הקדימות של sqrt(3+5) היא 0.
- הקדימות של elephant היא 0.
- הקדימות של (3+5) היא 0.
- הקדימות של (5,6)+*3 היא 400.

אסוציאטיביות

קונספט נוסף שחשוב להתייחס אליו הוא עניין ה*אסוציאטיביות* של האופרטורים. אנחנו כבר מכירים את ההבדל בין אופרטורים שמוגדרים פנימיים infix (כמו למשל +), אופרטורים תחיליים prefix (כמו ר), ולפעמים גם אופרטורים סופיים postfix (כמו למשל אופרטור עצרת ! המתמטי). בפרולוג האסוציאטיביות היא חלק בלתי נפרד מההגדרה.

הבעיה היא, שהגדרה של קדימות והכרעה האם מדובר באופרטור תוכי, תחילי או סופי לא מספיק בשביל להגדיר את האופרטור. ניקח לדוגמה "חיסור". זהו אופרטור תוכי המוגדר תחת קדימות 500 ב -SWI Prolog. האם המידע הזה מספיק לנו בשביל להבין את הפעולות שפרולוג עושה כאשר הוא עונה לנו על שאילתא?

```
?- X is 10-5-2.
X = 3
Yes
```

למה בעצם החישוב הוא כזה ולא מתבצע בצורה ש 5-2 = 3 ולאחר מכן 10-3 = 7 והחזרה של X = 7?
כמובן, שהפרולוג פועל נכון בכך שהוא עובד בסדר הזה הוא קודם מחסיר את ה-5 מה-10 ורק לאחר מכם מחסיר 2. דבר זה מוגדר בבסיס האופרטור. האופרטור – מוגדר כאופרטור תוכי, מה שאומר שהוא לוקח את הארגומנט הימני שצריך להיות קטן או שווה בקדימות ל 500 (הקדימות של אופרטור ה- בעצמו), כל

עוד הארגומנט השמאלי הוא ברמה שווה או נמוכה אליו. ברגע שמתייחסים לכלל זה, לא ניתן שסדר הקדימות של 2-5-10 תבוצע כ (2-5)-10 מאחר שהקדימות של הצד הימני הוא 500 שאינו קטן מוחלט מהצד השמאלי שהוא גם 500. בנוסף האופרטור מוגדר להיות יותר מקושר לשמאל או 'אסוציאטיבי שמאלי'.

בפרולוג האסוציאטיביות (יחד עם הבלות נוספות של קדימות ארגומנטים) מוצגת על ידי אטומים בדוגמת yfx. בדוגמה זאת, f מייצג את מיקום האופרטור (במקרה זה מדובר על אופרטור תובי) ו-x,y מסמנים את מיקום הארגומנטים. ה-y צריך להיקרא בעצם *במיקום זה צריך להיות מספר קדימות שווה או נמוך לזה של האופרטור*, בעוד שמשמעות ה-x הוא *במיקום זה הקימות צריכה להיות נמוכה ממש מזה של האופרטור*.

בדיקת קדימות ואסוציאטיביות

ניתן לבדוק את קדימות ואסוציאטיביות של אופרטור, על ידי שימוש באופרטור current_op/3. אם נכניס את הארגומנט השלישי להיות אופרטור כלשהו, נוכל לקבל בחזרה את הקדימות שלו בארגומנט הראשון, ואת דרך הקישור שלו בשני. הדוגמה הבאה של סימן הכפל * מראה את המספר שלו בקדימות 400 ואת תבנית הקישור שלו שהיא דומה לזה של חיסור.

```
?- current_op(Precedence, Associativity, *).
Precedence = 400
Associativity = yfx
Yes
```

הנה עוד מספר דוגמאות. שים לב ש- מוגדר פעמיים, אחד בתור אופרטור תובי (חיסור), והשני תחילי (מינוס)[6].

```
?- current_op( Precedence. Associativity, **).
Precedence = 200
Associativity = xfx ;
No

?- current_op( Precedence. Associativity, -).
Precedence = 500
Associativity = yfx ;
Precedence = 500
Associativity = fx ;
No

?- current_op( Precedence. Associativity, <).
Precedence = 700
Associativity = xfx ;
No

?- current_op( Precedence. Associativity, =).
Precedence = 700
Associativity = xfx ;
No
```

[6] על מנת לקבל את כל האופציות נלחץ במובן על ; ולבן בסוף חלק מהבדיקות יש שלילה בסוף.

```
?- current_op( Precedence. Associativity, :-).
Precedence = 1200
Associativity = xfx ;
Precedence = 1200
Associativity = fx ;
No
```

כמו שניתן לראות לא מדובר רק באופרטורים מתמטיים אלא גם כאלה כמו = ואפילו :- מוגדרים כאופרטורים. מהדוגמה האחרונה ניתן לראות ש :- יכול להיות מוגדר גם כאופרטור תחילי. נראה לזה דוגמה מתאימה בפרק הבא.

הטבלה הבאה מביאה סקירה של התבניות האסוציאטיביות. שים לב שלא ניתן "לקונן" אופרטורים שאינם אסוציאטיביים. לדוגמה, is מוגדר להיות אופרטור בתבנית xfx , מה שאומר שביטוי בצורת X is Y is Z יגרום בעיית סינטקס. זה כמובן הגיוני, מאחר והביטוי עצמו לא מניב שום משמעות.

תבנית	קישור אסוציאטיבי		דוגמה
yfx	תוכי	אסוציאטיבי שמאלי	+, -, *
xfy	תוכי	אסוציאטיבי ימני	, (עבור תתי-ביטויים)
xfx	תוכי	לא אסוציאטיבי	=, is, <
yfy	חסר היגיון, לא ניתן לבנות זאת.		
fx	תחילי	לא אסוציאטיבי	- (למשל 5- אינו הגיוני)
fy	תחילי	אסוציאטיבי	
xf	סופי	לא אסוציאטיבי	

הגדרת אופרטורים בעזרת פרדיקט op.

כעת אנחנו רוצים להגדיר פרדיקטים משל עצמנו. נחזור בשביל זה לדוגמה שבה הגדרנו גודל של חיות בחלקים הראשונים של ההסבר. נניח, ובמקום לכתוב את הביטוי בצורת is_bigger(elephant, monkey) אנחנו מעדיפים לבטא את זה בצורה של אופרטור תוכי:

```
elephant is_bigger monkey
```

דבר זה אפשרי, אך בשביל זה אנחנו צריכים להגדיר את is_bigger כאופרטור. בתור קדימות נוכל לבחור למשל 300. למעשה, זה לא משנה איזה מספר נבחר, כל עוד זה יהיה נמוך מ-700 (המקדם של =) וגדול מ -0. מה נגדיר בתור האסוציאטיביות? כבר אמרנו שאנחנו מעוניינים לעשות את זה בתור אופרטור תוכי. בתור ארגומנטים אנחנו מעוניינים רק באטומים או משתנים, כמו בתנאים של קימות 0. לבן, נבחר xfx על מנת להימנע מהיקלעות למצב בו הוא מקונן וצמוד לביטויי is_bigger אחרים.

הגדרת אופרטורים נעשית באמצעות האופרטור op/3, שלמה מוגדר בדיוק באותו אופן כמו current_op/3. ההבדל הוא, שכאן אנחנו ממש *מגדירים* את האופרטור במקום לשאול מה הערכים שלו. ולבן כל המשתנים הנכנסים לאופרטור זה חייבים להיות מאותחלים. שוב, הארגומנט הראשון מבטא את הקדימות של הפרדיקט, והשני את התבנית האסוציאטיבית, והארגומנט השלישי יהיה שם הארגומנט. כל אטום בפרולוג יוכל להיות שם של האופרטור, אלא אם הוא כבר מוגדר מראש. האופרטור is_bigger יוגדר באופן הבא:

```
?- op(300, xfx, is_bigger).
Yes
```

עכשיו הפרולוג מכיר את האופרטור, אבל עדיין אין לו שום מושג לגבי כיצד להעריך את הביטוי בעצמו. דבר זה צריך להיות מוגדר בצורה של עובדות וחוקים באופן הרגיל. ברגע שמשתמשים בהם תהיה לנו אפשרות לבחור בצורת הגדרה כמו שרצינו למעלה, וכן באופן הרגיל כמו שהגדרנו בחלק הקודם. ולכן ניתן יהיה "להתייעץ" עם הפרולוג באופן הבא:

```
?- elephant is_bigger donkey.
Yes
```

בכל הנוגע להתאמה, הפרדיקטים של האופרטור הם זהים, כמו שניתן לראות מהתשובה לשאילתא:

```
?- (elephant is_bigger tiger) = is_bigger( elephant, tiger).
Yes
```

ביצוע שאילתא בזמן ריצה

כמובן שזה לא יהיה נוח להגדיר מחדש כל דבר כזה בכל פעם שמאתחלים את הפרולוג. למזלנו אנחנו יכולים לבקש מהפרולוג לעשות את ההגדרות בזמן ריצה. במילים אחרות, ניתן להגדיר כל שאילתא ישירות מתוכך קובץ תוכנית, מה שיגרום לו להיות מוגדר מחדש בכל פעם שיתייעצו עם הקובץ הזה. הסינטקס של דבר כזה דומה לחוקים, רק ללא הגדרת ראש. אם למשל, מוגדרת בתוכנית השורה הבאה:

```
:- write( 'Hello, have a beautiful day!').
```

בכל פעם שנתייעץ עם הקובץ, תודפס לנו השורה באופן הבא:

```
?- consult ( 'my-file.pl').
Hello, have a beautiful day!
my-file compiled, 0.00 sec, 224 bytes.
Yes
?-
```

כעת ניתן לעשות בדיוק אותו דבר, רק עם הגדרת אופרטור. כלומר, ניתן להגדיר בקובץ באופן הבא עבור is_bigger:

```
:- op( 300, xfx, is_bigger).
```

בראש קובץ ההשוואה של החיות, והאופרטור יהיה זמין מאותו רגע של הקומפילציה של הקובץ.

פרק 5 – מעקב, CUT ושלילה

בפרק זה נלמד יותר לעומד ביצד פרולוג עונה על שאילתות. נציג גם מנגנון שליטה (cut) שנותן לנו אפשרות להשמה יעילה יותר. יותר מזה, נציג גם הרחבות נוספות לשפת הפרולוג. מלבד ה"וגם" שנוצר בעזרת פסיק, נציג גם את השלילה ואת ההפרדה.

מעקב ו-CUT

מעקב

את הביטוי מעקב backtracking הזכרנו כבר בפרק הראשון. במהלך הוכחת הביטויים הפרולוג שומר רישום של האפשרויות השונות, כלומר, מקומות בהם יש יותר מאפשרות אחת. ברגע שדרך אחת מבוררת להיות fail (או אם המשתמש מבקש דרך אחרת), המערכת קופצת בחזרה לנקודת הפיצול וממשיכה שוב את הדרך האלטרנטיבית. דבר זה הוא נקודה מאוד חשובה בפרולוג שמסייעת לפתור בעיות רבות.

בואו נסכל על דוגמה. אנחנו רוצי לכתוב פרדיקט שיכתוב את כל התמורות האפשרויות לרשימה נתונה. היישום של זה כפי שכתבנו אותו משתמש בפרדיקט המובנה select/3, שלוקח רשימה ומשווה אותה לאלמנט נתון. המשתנה השלישי יהיה אותה רשימה רק ללא אותו אלמנט.

להלן הגדרה רקורסיבית פשוטה של הפרדיקט permutation/2:

```
permutation([] ,[]).

permutation(List, [Element | Permutation]) :-
        select (List, Element, Rest),
        permutation( Rest, Permutation).
```

מקרה הבסיס הוא כאשר מקבלים רשימה ריקה. לדבר זה יש רק תמורה אחת אפשרית והיא להחזיר את הרשימה הריקה בעצמה. אם ברשימה המתקבלת ישנם מספר אלמנטים, תת-התנאי select (List, Element, Rest) יתקבל ויקשור את Element לאיבר הראשון ברשימה. דבר זה יכניס את הרשימה המוחזרת בצורה רקורסיבית ביחד עם הזנב של הרשימה. הערך המוחזר הראשון יהיה פשוט הרשימה המובנסת, מאחר שElement יוצב להיות ראש הרשימה. במידה ודרושות אלטרנטיבות נוספות, עושים מעקב חוזר על הselect, כלומר, בכל פעם שElement מקבל ערך שונה מהרשימה List. דבר זה יחשב את כל התמורות האפשריות של הרשימה הנתונה. דוגמה:

```
?- permutation( [1, 2, 3], X).

X = [1, 2, 3] ;

X = [1, 3, 2] ;

X = [2, 1, 3] ;

X = [2, 3, 1] ;

X = [3, 1, 2] ;

X = [3, 2, 1] ;

No
```

ראינו דוגמאות נוספות החלקים הקודמים על אפשרויות בהם יש לנו באקטראקינג, כמו בחלק 2.2. שם השתמשנו באפשרו הזאת על מנת לממש את concat_lists/3 על מנת למצוא את כל החיבורים האפשריים לרשימה.

בעיות עם Backtracking

ישנם מקרים בהם אנחנו לא רוצים לבצע את החזרה לאחור. ניקח לדוגמה את הגדרת הפרדיקט remove_duplicates/2 על מנת להסיר איברים כפולים מרשימה נתונה.

```
remove_duplicates ([], []).

remove_duplicates ( [Head | Tail], Result) :-
        member( Head, Tail),
        remove_duplicates( Tail, Result).

remove_duplicates( [Head | Tail], [Head | Result]) :-
        remove_duplicates( Tal, Result).
```

ה*משמעות ההצהרתית* של הפרדיקט הזה הא בדלקמן – הסרת האיברים הכפולים מרשימה ריקה היא כמובן רשימה ריקה בחזרה. בזה אין שום פסול. התנאי השני אומר שאם ניתן למצוא את ראש הרשימה איפשהו בזנב שלה, אזי נכנסים בצורה רקורסיבית לתוך remove_duplicates/2 רק עם הזנב, ובהתעלמות מהראש. אחרת נכנס שוב לתוך הפרדיקט רק שנצרך את הראש לתוך הרשימה של התוצאה.

זה עובד כמעט טוב. הפתרון הראשון תמיד יעשה את המוטל עליו. אבל כאשר נבקש אלטרנטיבות נוספות הדברים יתחילו להיהרס. שתני החוקים מהווים צמתים. לענף הראשון של עץ החיפוש פרולוג ייקח תמיד את האפשרות הראשונה מבין השניים, כל עוד זה אפשרי, ואם הראש הוא חלק מהזנב אז זה ייפול. אבל בגלל המעקב חזרה, כל ענפי העץ יקבלו ביקור נוסף. אפילו אם מצאנו את מה שאנחנו רוצי כבר בתנאי הראשון, אנחנו ניכנס לתנאי השני והראש הכפול יישאר ברשימה. הפלט השגוי (לפחות באופן סמנטי) ייראה כמו בדוגמה הבאה:

```
?- remove_duplicates( [a, b, b ,c, a], List).

List = [b, c, a] ;

List = [b, b, c, a] ;

List = [a, b, c, a] ;

List = [a, b, b, c, a] ;

No
```

על מנת לפתור את הבעיה הזאת אנחנו צריכים למצוא דרך להגיד לפרולוג, שאפילו אם המשתמש (או פרדיקט אחר שעלול לקרוא לremove_duplicates/2) יבקש משמעויות נוספות, לא נקבל שום אלטרנטיבות שיביאו לנו תוצאות שייחשבו כטעויות.

הכרות עם CUT

פרולוג בהחלט מביא לנו פתרון לבעיה שנידונה מקודם. ישנה אפשרות 'לחתוך' את האפשרות לחזור על צמתים ולמנוע בדיקת קיימות עבור החיפוש לפתרונות האלטרנטיביים לשאילתא.

כתיבה של *cut* היא באמצעות סימן קריאה - !. הקאט הוא פרדיקט מובנה במערכת אותו ניתן להציב בגוף של חוק (או באופן דומה, להיות חלק מרצף של תתי ביטויים של שאילתא). הוצאה לפועל של הביטוי !

תמיד יצליח, אך פעולת ה-backtracking לתתי הביטויים שנמצאים *לפני* הקאט בתוך אותו גוף החוק לא יהיו נגישים יותר.

נגדיר את זה באופן מדויק יותר בהמשך. נסתכל כעת על הדוגמה של מחיקת האיברים הכפולים מתוך הרשימה. נשנה את התוכנית שהוצעה קודם, על ידי הכנסה של קאט אחרי תת התנאי הראשון. כל שאר הקוד נשאר בדיוק אותו דבר כמו קודם.

```
remove_duplicates ([], []).

    remove_duplicates ( [Head | Tail], Result) :-
            member( Head, Tail), !,
            remove_duplicates( Tail, Result).

    remove_duplicates( [Head | Tail], [Head | Result]) :-
            remove_duplicates( Tal, Result).
```

כעת, ברגע שראש הרשימה יהיה חלק מזנב הרשימה, כלומר member (Head, Tail) יחזיר אמת, ייכנס לפעולה הביטוי הבא, !, וגם הוא יצליח. ללא הקאט הזה תהיה לנו אפשרות להמשיך את המעקב, ולהמשיך לביטוי השני על מנת למצוא פתרונות אלטרנטיביים לבעיה. אבל ברגע שפרולוג הגיע לקאט, דבר זה אינו אפשרי יותר: לא תינתן האפשרות לחפש פתרונות אלטרנטיביים.

שימוש בפרדיקט המחודש remove_duplicate/2 תניב לנו את התוצאה המבוקשת. אם נבקש את הפתרונות האלטרנטיביים על ידי לחיצה על ;, נקבל מיד את התשובה – לא.

```
?- remove_duplicates( [a, b, b ,c, a], List).

List = [b, c, a] ;
No
```

עכשיו אנחנו מוכנים להגדרה מדויקת יותר של *הקאט* בפרולוג: ברגע שcut מופיע בגוף החוק, כל האפשרויות שנעשו בין התאמת ראש החוק למטרה הראשית להגעה לקאט הגיעו לרמה סופית, כלומר כל הצמתים שהיו באמצע פשוט נעזבות.

נפשט את העניין הזה בסיפור. נניח שהיה נסיך שרצה להתחתן. בימי קדם הוא פשוט היה צריך לרתום את הסוס הטוב יותר שלו ולצאת לרכיבה לכיוון העמק, לצורך העניין של אסקס, ולמצוא לעצמו את הנערה הצעירה, יפה וחכמה בה הוא מעוניין. אבל, כמובן, הזמנים השתנו, החיים בכלליות נעשו הרבה יותר מסובכים היום, וחשוב מזה, הנסיך היום עסוק בלהגן על המולדת מפני קומוניזם/ אנרכיה/ דמוקרטיה (ניתן לבחור מה שמתאים לך). למזלו, היועצים המלכותיים שלו מכילים את מיטב המוחות הפסיכולוגיים ומתכנתי הפרולוג המוצלחים ביותר שהעולם הזה יכול להציע. הם מכנסים וועדה שמטרתה ליצור בפרולוג תוכנית שתעזור לנסיך למצוא את הכלה המיודעת בצורה אוטומטית. מהפסיכולוגיים המדופלמים אנחנו מקבלים את העובדות הבאות:

- הנסיך מחפש בעיקר בחורה *יפה*. אבל, על מנת שתתאים לתפקיד אשת הנסיך, היא צריכה גם להיות *חכמה*.
- הנסיך צעיר ודווקא גבר רומנטי. לכן, הוא ייפול ברשתה של הבחורה הראשונה בה ייתקל, יאהב אותה לנצח, ולא ישקול אפילו אישה אחרת עד סוף ימיו. דבר זה תקף אפילו אם לא יתחתן בסוף עם אותה נערה.

שירות הביטחון מספק לוועדה מסד נתונים של נשים שמתאימות לטווח הגילאים המתאים. הנתונים מובנסים בסדר המתאים ביותר לאפשרות שהנסיך ייתקל באחת מהן במהלך רכיבתו לאורך הארץ. הרשימה נראית כ עובדות פרולוג באופן הבא:

```
beautiful(claudia).
beautiful (sharon).
beautiful( denise).
...
intelligent( Margaret).
intelligent( sharon).
...
```

לאחר מחשבה מאומצת, וועדת המשנה לענייני פרולוג מגיעה עם התוכנית הבאה:

```
bride( Girl) :-
        beautiful( Girl), ! ,
        intelligent (Girl).
```

נשאיר כרגע את הקאט בשורה השנייה ללא מחשבה כרגע. ונכניס את השאילתא הבאה:

```
?- bride (X).
```

השאילתא אמורה להצליח במידה ונמצא בחורה X במסד הנונים, שהיא מקיימת את שני העובדות והיא גם יפה וגם חכמה. וכך, נקיים את התנאי הראשון עליו הפסיכולוגים הסכימו שראויה להיות כלה לנסיך. שם הבחורה יוכל להיות מושם היטב עבור המשתנה X.

על מנת לקיים את התנאי השני, הוספנו את הקאט. אם התנאי beautiful (Girl) מתקיים, כלומר ניתן למצוא ערך שיוכל להיות מוצב ב beautiful(X) (וזה יהיה האיבר הראשון מסוגו בו הוא ייתקל), בחירה זאת תהיה סופית, גם אם אותה X לא תקיים במקביל גם את התנאי השני- Intelligent(X).

בהתחשב במסד נתונים זה יש בעיה שעלולה להיווצר אצל הנסיך. הבחורה הראשונה שהיא גם יפה אותה הוא יפגוש תהיה קלאודיה, והוא יתאהב בה באותו רגע ועד עולם. בפרולוג זה יתאים עבור התנאי beautiful(Girl) כאשר ההשמה המתאימה תהיה Girl = Claudia. וזה יישאר כך לנצח, מאחר וברגע והקאט יצא לפועל, הבחירה כבר לא יכולה להשתנות. במקרה זה, קלאודיה אינה הבחורה הכי מבריקה שאנחנו מקווים למצוא, מה שאומר שהם אינם יכולים להתחתן. מבחינת פרולוג זה אומר, שעבור התנאי intelligent(Girl) הוא אנחנו מנסים להציב עבור Girlאת קלאודיה יוחזר ערך שקר – לא מקוים, מאחר ואין עובדה מתאימה לה בתוכנית. זה אומר שכל השאילתא תיכשל. כל זה למרות שיש שם בכל הנתונים שיהא גם יפה וגם חכמה (שרון), משימת הנסיך להתחתן נידונה לכישלון:

```
?- bride(X).
No
```

בעיות עם CUT

קאטים הינם יעילים במיוחד על מנת "להדריך" את האינטרפרטר לכיוון פתרון. אך דבר זה לא בא בחינם. על ידי הכנסת קאטים, אנחנו מוותרים מרצון על מספר אופציות (נחמדות) שמועילות לנו במערכת. דבר זה מוביל לפעמים לתוצאות בלתי רצויות.

על מנת לתאר זאת, בוא נתאר פרדיקט בשם add/3 שמטרתו להכניס איבר לתוך רשימה, בתנאי שאיבר זה לא מופיע כבר במקום אחר ברשימה. האיבר המוכנס יהיה הארגומנט הראשון בפרדיקט, הרשימה בתור האיבר השני, והמשתנה המוכן להצבה בתור הארגומנט השלישי. לדוגמה:

```
?- add( elephant, [dog, donkey, rabbit], List) :-
List  = [elephant, dog, donkey, rabbit] ;
No

?- add( donkey, [dog, donkey, rabbit], List) :-
List  = [dog, donkey, rabbit] ;
```

הדבר החשוב לציון כאן, הוא שאין באלטרנטיבות תשובות שאינן נכונות. תוכנית הפרולוג הבאה תעשה את העבודה:

```
add ( Element, List, List) :-
        member( Element, List), ! .

add (Element, List, [Element | List]).
```

אם האיבר המיועד להכנסה נמצא כבר ברשימה, הרשימה המוחזרת אמורה להיות בדיוק כמו רשימת הקלט. מאחר וזו התשובה הנכונה, אנחנו מונעים מהפרולוג להמשיך ולחפש אפשרויות נוספות על ידי הקאט. במידה והאיבר לא ברשימה, אנחנו משתמשים בתבנית ה[Heat | Tail] על מנת לבנות רשימת פלט.

במקרה זה הקאט עלול להיות בעייתי. אם נשתמש בו באופן שציינו, על ידי השמה של משתנה לא מאותחל בתור הארגומנט השלישי, הכל יעבוד בצורה שרצינו, והפרדיקט add/3יעבוד כמתוכנן. אך אם נכניס לארגומנט השלישי משתנה מאותחל, התגובה של פרולוג עלולה להיות שונה ממה שאנחנו מצפים. דוגמה:

```
?- add( a, [a, b, c, d], [a, a, b, c, d]).
Yes
```

יש צורך להשוות ולעבור על התוכנית על מנת להבין איפה טעינו. בקיצור, יש להיזהר עם הקאט!

שלילה ככישלון

בדוגמה הקודמת הסקנו כי מאחר ואין לנו את העובדה intelligent (claudia), אזי קלאודיה היפה איננה חכמה במיוחד. דבר זה נוגע לעניין חשוב בפרולוג, נקרא לו שלילה.

עולם ההנחות

על מנת לתת תשובת אמת לשאלה נתונה, על פרולוג לבנות בסיס הוכחה מתוך כל העובדות והחוקים הקיימים המובילים ישירות את התשובה. לכן, אם נדייק בעניין, המענה Yes בתשובה לשאילתא אין משמעו רק שהשאילתא נכונה, אלא שהיא *הוכחה כנכונה*. בהתאמה ניתן לומר, כי תשובת No לא אומרת בהכרח שתוצאת השאילתא היא שקר, אלא שהיא *לא הוכחה כנכונה*: פרולוג נכשל מלספק לנו רצף הוכחה שלם.

הגישה של שלילת כל מה שלא מצוין באופן ברור בתוכנית (או לחילופין, בר הוכחה על ידי מהלך התוכנית הרגיל) מצוין בדרך כלל תחת המושג *עולם ההנחות הסגור*. הכוונה היא, אנחנו מתייחסים לפרולוג כמין עוולם קטן בפני עצמו, ויוצאים מנקודת ההנחה שכל דבר שאנחנו לא רואים, אינו קיים (אינו אמת).

בחיי היום-יום שלנו אנחנו בדרך כלל לא נוטים לעשות קפיצות לוגיות כאלה. אם נחפש הגדרה של ברווזן (duckbill) בספר טבע ענק ולא נמצא, אנחנו לא נסיק שהברווזן פשוט אינו חיה. לעומת זאת בפרולוג, אם יוצגו לנו העובדות הבאות:

```
animal( elephant).
animal( tiger).
animal( lion).
...
```

העובדה animal(duckbill) אינה מופיעה ברשימת העובדות הזאת (ואין שום חוק בו ניתן להגדיר animal/1 שאפשר יהיה להגדיר אותו שוב. ולכן בתגובה לשאילתא האם הברווזן הוא חיה התשובה תהיה בדלקמן:

```
?- animal( duckbill).
No
```

עולם ההנחות המצומצם אולי ייראה לנו בהתחלה כצרות-מחשבה, אבל במחשבה עמוקה יותר, זו הדרך היחידה בה הפרולוג יוכל לפרש נכונה באמצעות תנאים שהם מוכרחים והחלטיים. יש לשים לב, שאם הגדרנו בוודאות את כל הפניות האפשריות וכיסינו את כל האופציות בצורה שלפרולוג תמיד יש דרך ברורה להסיק את המסקנות הרצויות, במקרה כזה אכן No מתכוון ל"לא".

אופרטור +\

לפעמים לא נרצה לדעת האם שאלה מסוימת מחזירה אמת, אלא דווקא אם היא נכשלת. למעשה, אנחנו רוצים תוצאת *שלילה*. בפרולוג זה אפשרי על ידי האופרטור +\. זהו אופרטור תחילי שיכול להתחבר לכל ביטוי בפרולוג. היעד מסוג Goal +\ יצליח, במידה שאכן Goal החזיר שלילה, ולהיפך. במילים אחרות, +\ Goal יחזור עם תוצאה חיובית, אם"ם הפרולוג לא הצליח להוכיח את Goal.

הסמנטיקה של אופרטור השלילה ידוע בשם *שלילה של כישלון*. כישלון בפרולוג משמעותו שלא הצלחנו לספק הוכחה לדרוש. בחיים האמתיים זה בדרך כלל לא ופס (מלבד הכלל המשפטי 'זכאי עד שתוכח אשמתו'). ובמתמטיקה האמתית בוודאי שאי אפשר לקבוע הוכחות על ידי זה שפשוט לא מצאנו שום דבר אחר.

בואו נסתכל ברגע על דוגמה של אופרטור +\ . נניח ויש לנו רשימת עובדות פרולוג של אנשים המסומנים כנשואים האחד לשני:

```
married ( peter, lucy).
married ( paul, mary).
married (bob, Juliet).
married ( harry, Geraldine).
```

במקרה זה נוכל להגדיר את הפרדיקט single/1 המחזיר תשובת אמת במידה והארגומנט הניתן לא שייך לארגומנט הראשון או השני של married/2. אנחנו יכולים להשתמש במשתנה האנונימי לארגומנט השני מאחר והוא לא רלוונטי לנו:

```
single ( Person) :-
        \+ married( _, Person),
        \+ married( Person, _).
```

שאילתות לדוגמה יהיו:

```
?- single( mary).
No

?- single( Claudia).
Yes
```

שוב, עלינו לקרוא את התשובה לשאלה בתור 'קלאודיה בנראה רווקה, מאחר ואנחנו לא יכולים להוכיח שהיא נשואה'. אנחנו יכולים גם לקצר את התשובה פשוט ל'קלאודיה *היא* רווקה', אך ורק במידה ואנחנו בטוחים שכיסינו את כל האפשרויות תחת רשימת העובדות married/2, כלומר אם עולם הדיון הסגור שלנו הוא אכן אמיתי.

בעת נסתכל על השאילתא הבאה ועל תגובת הפרולוג:

```
?- single( X).
No
```

המשמעות היא, שהפרולוג לא יכול להוכיח לכל X האן הוא רווק או לא.

איפה להשתמש ב +

הזכרנו כבר את האופרטור +\ אנחנו יכולים לשייך לכל תנאי וביטוי בפרולוג. צריך להבין מה זה אומר. מטרות-להשגה הם או (תתי) מטרה של שאילתא או תתי מטרה של גוף החוק. עובדות וראשי חוקים אינם מטרות. לכן, אין זה אפשרי לשלול עובדה או ראש של חוק. דבר זה מתאים בדיוק למה שאמרנו לגבי עולם הדיון של ההנחות ושלילת הכישלון: אין אפשרות להכריז על משהו בתור שקר.

הפרדה

הפסיק המופיע בין שני תתי ביטויים בגוף החוק נקרה *הפרדה (conjunction)*. הוא אומר שההכרזה על הצלחה תהיה אם"ם שני התנאים שהפסיק נמצא ביניהם יתקיימו.

אנחנו כבר מכירים דרך אחת לבצע הפרדה. אם ישנם שני חוקים בעלי אותו ראש הם מיוצגים בעזרת ההפרדה הזו, מאחר ובמהלך של ביצוע התנאים פרולוג בוחר אחת מהדרכים על מנת לבדוק תאימות ואז עובר לשני. כמובן שהוא ינסה קודם כל את החלק הקודם בשורה מבין השניים, ואת השני הוא יבצע רק במידה והתנאי הראשון נכשל, או במידה והמשתמש ביקש תוצאות אלטרנטיביות.

ברוב המקרים צורה כזו של הפרדה היא זאת שאנחנו נשתמש, אך לפעמים יהיה לנו שימושיות דווקא ללכת על סימון אחר ; על מנת להפריד בין שני המקרים המוצעים. לדוגמה ניקח את ההגדרה הבאה עבור parent/2:

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

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

המשמעות של התוכנית הזאת היא – 'X הוא הורה של Y, בתנאי שאנחנו נוכל להראות שX הוא אבא של Y או שנראה שהוא אמא של Y'. את אתה הגדרה אנחנו יכולים לרשום בפשטות גם בצורה הבאה:

```
parent(X, Y) :-
        father( X, Y) ;
        mother( X, Y).
```

יש לשים לב שהקדימות של ; גבוהה יותר מזו של , (פסיק). ולכן כאשר עושים הפרדה בתוך הפרדה יש לשים לב לקדימויות השונות.

כדאי למעט בשימוש בסימן ; מאחר וקל מאוד להתבלבל בינו ובין הפסיק והתוכנית עלולה לאבד מהקריאות שלה.

דוגמה: הערכה לוגית

בתור דוגמה, ננסה לכתוב תוכנית המנסה להעריך תוצאה של שורה בטבלת אמת. נצא מנקודת הנחה שהאופרטורים הדרושים הוגדרו כבר קודם. בעזרת שימוש באופרטורים האלה, אנחנו רוצים להקליד תנאים שיתאימו לנוסחה שבשאלה (כאשר הסימונים הלוגיים יוחלפו בסימני אמת) והתוכני תחזיר לנו את הערכים הסופיים האם הם אמת או שקר.

על מנת לקבל את טבלת האמת עבור A∧B נצטרך לבצע את ארבעת השאילתות הבאות:

```
?- true and true.
Yes

?- true and false.
No

?- false and true.
No

?- false and false.
No
```

לפי זה, טבלת האמת תיראה כמו בדוגמה הבאה:

A	B	A∧B
T	T	T
T	F	F
F	T	F
F	F	F

דוגמה אחת נוספת לפני שנתחיל:

```
?- true and (true and false implies true) and neg false.
Yes
```

בדוגמאות השתמשנו באטומים המוגדרים בפרולוג true ו-false. למעשה זה מוגדר בדיוק כמו שאנחנו רוצים, וזה בסדר מבחינתנו. ההבדל היחיד הוא שאת falseאנחנו צריכים שיהיה בדיוק כמו fail מלבד השם שלו. אז את החוק של falseנגדיר באופן הבא:

```
false:- fail.
```

עכשיו יש נו הפרדה ואיחוד. שהם כמובן יגיבו לאופרטורים ',' ו';' בהתאמה:

```
and( A, B) :- A, B.

or( A,B) :- A;B.
```

כלל השלילה שלנו יהיה האופרטור neg שהוא כמובן יצרו שימוש באופרטור +\:

```
neg( A) :- \+ A.
```

הגדרת השתמעות (גרירה) היא טיפה יותר טריקית. דרך אחת תהיה פשוט כלל השקילות A=>B ≡¬A∨B ובעת אנחנו יכולים להגדיר אותו בעזרת שלילה ו"או". קצת יותר נוח אבל להשתמש כאן ב"קאט". באופן הבא:

```
implies( A, B) :- A, ! ,B.
implies( _, _).
```

איך זה עובד? נניח וA הוא false. במקרה זה הכלל הראשון ייכשל, ופרולוג ידלג הלאה לבדוק מה B יצא. זה בדיוק מה שאנחנו רוצים: השמה של T במידה ולפני זה הערכנו F. במקרה שהוא יצליח, הקאט יצליח אם"ם B יצליח. שוב, בדיוק מה שאנחנו רוצים.

שים לב

ידוע שמבחינת לוגיקה קלאסית ¬A שקול ל A=> F. באופן דומה, במוקם להשתמש ב \+ אנחנו יכולים להגדיר בפרולוג את השלילה שלנו באופן הבא:

```
neg( A) :- A, !, fail.
neg( _).
```

פרק 6 – הבסיס הלוגי מאחורי פרולוג

במהלך ההסברים עד כה, השתמשנו בביטויים כמו 'אטום', 'פרדיקטים', 'אמת', 'הוכחה' וכו' בהקשרים של תכנות בפרולוג והדרך בה מוציאים לפועל מהלכים, בהם הפרולוג מנסה לענות על שאילתות, וניתן כבר לראות באופן די ברור שישנו קשר חזק בין לוגיקה לפרולוג. לא רק שפרולוג היא שפת התכנות המתאימה ביותר על מנת לבטא היגיון מתמטי, אלא כל תהליך ה"חשיבה" והסקת המסקנות בפרולוג מבוססים על הסקה לוגית. חלק זה של ההסברים ייתנו לנו רושם ראשוני על הלוגיקה מאחורי פרולוג.

נתחיל בלהראות דוגמה (פשוטה) כיצד תוכנית פרולוג מיתרגמת למערכת של צירופים לוגיים. הנוסחאות יהיו בצורה הלוגית אתה למדנו בעבר, ונראה כיצד פותרים את הנוסחאות על ידי האינטרפרטר של פרולוג.

תרגום של משפטי פרולוג לנוסחאות

בחלק זה נתמקד בעניין כיצד משפטים של פרולוג (כלומר – עובדות, חוקים, ושאילתות) יכולים להיתרגם לנוסחאות לוגיות. אנחנו נדון בבסיס הסינטקס של הפרולוג, או ליתר דיוק לא נדון ב"קאט", שלילה, הפרדה, משתנה אנונימי, או הערה אריתמטית של ביטויים בנקודה זו. שים לב שבהתחשב בייצוג הפנימי שלהם, לרשימות יש התייחסות שונה שלא תדון כאן.

פרדיקטים בפרולוג תואמים לסימני פרדיקטים בלוגיקה, תנאים בתוך פרדיקטים תואמים גם הם לתנאים הפנימיים בפרדיקטים בלוגיקה. התנאים נכתבים על ידי קבועים ('אטומים' בפרולוג), משתנים, וסימנים פונקציונליים (פונקציות פרולוג). כל המשתנים בתנאים של פרולוג הם אוניברסליים (כלומר, כל משתנה יוכל לקבל כל ביטוי בפרולוג).

בהתחשב בפרדיקטים בפרולוג והתרגום שלהם לתוך חוקים לוגיים יש לזכור שמדובר בתרגום ישיר.
הכוונה היא ש -: יכול להיקרא כ'אם', כלומר כהשמה מימין לשמאל, והסיק המפריד בין תתי ביטויים מתפקד כמפריד. שאילתות בפרולוג יכולות להיחשב כחוקים בפרולוג ללא ראש. הראש הריק מתורגם כ-$\bot$.הסיבה לכך תוסבר בהמשך. כאשר מתרגמים ביטוי, לכל משתנה X המופיע בביטוי אנחנו נמיר אותו ל-$\forall x$ בתחילת הנוסחה.
לפני סיכום תהליך התרגום בצורה פורמלית יותר נביא דוגמה. ניקח את התוכנית הקטנה הזאת המכילה שתי עובדות ושני חוקים:

```
bigger( elephant, horse).
bigger( horse, donkey).
is_bigger( X, Y) :- bigger (X, Y).
is_bigger( X,Y):- bigger( X, Z) , is_bigger( Z,Y).
```

נראה כעת את המימוש בנוסחאות לוגיות:

{ bigger(elephant, horse),
bigger(horse, donkey),
$\forall x.\forall y.(bigger(x,y) \Rightarrow is_bigger(x,y))$,
$\forall x.\forall y.\forall z.(bigger(x,z) \wedge is_bigger(x,y))$ }

שים לב כיצד החלק הראשון נכתב כהשמה. כמו כן, יש לשים לב, שכל השמה מבוצעת מחדש ובצורה עצמאית. זה מתאים לחוק שכל ביטוי הוא מופיע ומתפקד בצורה עצמאית, אפילו אם יש לו את אותו השם. לדוגמה, הX בחוק הראשון הוא עצמאי ואין שום קשר בינו לבין הX בחוק השני. יותר מזה, כבל הנראה יושמו בכל אחד דברים שונים לגמרי. אם נציב בX הראשון למשל פיל, ובשני נצי כלב, זה לא ישפיע על מהלך התוכנית. דבר זה נקרא *השמה מחדש של משתנים קשורים*.

אם מספר תנאים מרכיבים תוכנית, אותה תוכנית מגיבה למספר תנאים ולסט של נוסחאות וכל אחד מהתנאים הללו מגיב ישירות לנוסחה אחת בלבד של המערכת. כמובן, אנחנו יכולים גם לתרגם תנאי יחיד. למשל, השאילתא הבאה

?- is_bigger(elephant, X), is_bigger(X, donkey).

מגיב לנוסחה הבאה:

$\forall x.(is_bigger(elephant,x) \wedge is_bigger(x,donkey) \Rightarrow \bot)$

כמו שאנחנו יודעים, שאילתות יכולות להיות גם חלק מתוכנית פרולוג (במקרה זה הם מובלים על ידי :-), בלומר, נוסחה שכזאת יכולה להיות חלק ממערכת שמגיבה לכל התוכנית.

לסיכום, כאשר מתרגמים תוכנית פרולוג (או לחילופן – רצף של תנאים) לסט של נוסחאות לוגיות, עלינו לעבוד לפי הסדר הבא:

- כל פרדיקט בפרולוג יומר לנוסחה אטומית לוגית (בסופו של דבר, שניהם אותו דבר: ניתן פשוט לכתוב את זה מבלי לשנות כלום).
- פסיקים בין תתי תנאים מתורגמים להיות איחוד לוגי (בלומר, עלינו להחליף כל פסיק בין שני פרדיקטים בסימן $\wedge$ בתוך הנוסחה).
- חוקים של פרולוג מתורגמים להיות הצבה של משתנים, כאשר גוף החוק הוא הגורר, והראש הוא הנגרר (בלומר את :- נשכתב בתור $\Rightarrow$ ונשנה את הסדר של הראש והזנב – $x \Rightarrow y \equiv$ X :-).
- כל משתנה שיוגדר באחד מהתנאים יקבל תנאי "לכל" בנוסחה (בלומר, כאשר יש לנו משתנה X בלשהו, אנחנו ממירים אותו ל-$\forall x$ וכותבים אותו בראש הנוסחה.

נספח – רקורסיה

רקורסיה הוזברה לא מעט בין דפי ההסברים הללו. דבר זה במובן לא תופעה ייחודית לפרולוג, אלא חלק מהבסיס של כל שפת תכנות כלשהי באופן כללי.

לחלק מהאנשים עניין הרקורסיה קצת קשה לתפיסה בתור התחלה. במקרה והקורא/ת רואה את זה כקביעה נכונה, החלק הבא יכול להיות מאוד שימושי.

1.1. אינדוקציה

תהליך הרקורסיה מקביל בעצם לתהליך האינדוקציה המתמטית. על מנת להראות נכונות לכל המספרים הטבעיים, נראה זאת למקרה הבסיס (n=1) ונראה שמנקודה זאת והלאה, התנאי הוא אמת לכל n על ידי קביעה שהחוק מתאים גם לn+1. דבר זה יוכיח את הטענה עבור כל המספרים הטבעיים.

נסתכל כעת על דוגמה. אנחנו בוודאי זוכרים את הנוסחה לחישוב סכום של n מספרים טבעיים. לפני שנשתמש בנוסחה, אנחנו קודם יוצאים מנקודת הנחה שהיא אכן נכונה.

טענה: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ (הצגת ההיפותזה)

הוכחה על ידי אינדוקציה:

עבור n=1 (מקרה הבסיס): $\sum_{i=1}^{n} i = 1 = \frac{1(1+1)}{2}$ √

עבור n~n+1 (צעד האינדוקציה, שימוש בהשערה): $\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) = \frac{n(n+1)}{2} + n + 1 =$ $\frac{(n+1)(n+2)}{2}$ √

העיקרון הרקורסיבי

הרעיון הבסיסי מאחורי תוכנית רקורסיבית, *העיקרון הרקורסיבי* הוא כדלקמן: על מנת לפתור בעיה מורכבת, עלינו לספק את הפתרון עבור הבעיה הקטנה ביותר מסוגה, ולספק כלל שעוקב אחרי השינויים כך שהבעיה הגדולה (והמורכבת9 תהיה הרבה יותר פשוטה.

במילים אחרות, עלינו לספק פתרון עבור המקרה הקל ביותר, ולספק *חוק רקורסיבי* (או *צעד*). באותו רגע נקבל אלגוריתם (או תוכנית) שפותרת את כל הבעיה הנתונה.

בעזרת שימוש ב*אינדוקציה*, אנחנו מוכיחים הצהרה על ידי שימוש במקרה הבסיס כלפי "מעלה" לעבר כל האפשרויות הנתונות. בעזרת ה*רקורסיה*, אנחנו מחשבים פונקציה עבור מקרה שרירותי שעובר "מט" עד מקרה הבסיס.

הגדרה רקורסיבית של פונקציות

סימן העצרת !n מוגדר להיות הכפלה של כל המספרים הטבעיים החל מ-1 ועד n. להלן הגדרה יותר פורמלית, המוגדרת בצורה רקורסיבית:

1! = 1	(מקרה הבסיס)
n! = (n-1)!*n for n>1	(הכלל הרקורסיבי)

על מנת לחשב את הערך של !5 אנחנו צריכים לעבור על החלק השני של ההגדרה 4 פעמים עד שנגיע למקרה הבסיס ובעזרתו נוכל לעשות חישוב כללי. זוהי רקורסיה!

רקורסיה בג'אווה

להלן פונקציה הכתובה בג'אווה המחשבת עצרת של מספרים טבעיים. כמובן, היא מוגדרת באופן רקורסיבית:

```
public int factorial(int n) {
  if (n==1) {
    return 1;  //base case
} else {
    return (factorial(n-1)*n; //recursion step
  }
}
```

רקורסיה בפרולוג

כעת נגדיר את אותה פונקציה רק בפרולוג:

```
factorial(1,1).            % base case

factorial ( N, Result) :-  %recursion step
  N > 1,
  N1 is N -1,
  factorial( N1, Result1),
  Result is Result1*N.
```

ניקח כדוגמה, את השאילתא factorial (5, X) ונעקוב אחרי ביצוע המרה צעד אחרי צעד, בדיוק כמו שהפרולוג יעשה – ולמעשה גם אנחנו אם נרצה לבצע את ה!5 בעצמנו.

דוגמא נוספת

הפרדיקט הבא משמש לבדיקת אורך של רשימה (למעשה באופן דומה לצורה שמגיב הפרדיקט המובנה length/2):

```
len([], 0).                    % base case

len( [_ | Tail], N ) :-       % recursion step
  len (Tail, N1),
  N is N1+1.
```

אילו בעיות ניתן לפתור

אתה יכול לפתור בעיות רקורסיביות רק במידה וטווח הבעיות שלך הוא כזה שניתן לפרמטריזציה. בדרך כלל, פרמטרים כאלה מגדירים מורכבות של בעיה שפותרת בעיקר מספרים (טבעיים) ובפרולוג גם רשימות (או אורכים שלהם).

יש לוודא שכל צעד רקורסיבי באמת ישנה את הבעיה לעבר צעד פשוט יותר עד שנוכל להגיע למקרה הבסיס.

למשל, אם מורכבות הבעיה שלך תלויה במספר טבעי, צריך לשים לב שאכן משתלשלים מטה לעבר מקרה הבסיס. בפרדיקט factorial/2 שהגדרנו קודם ירדנו עד 1, ובפונקציה len/2 הארגומנט יורד מטה עד לרשמה הריקה.

הבנה

יש לעשות מאמץ על מנת להבין באמת לפחות אחד מהגדרות הפרדיקטים, למשל concat_lists/3 או len/2 לגמרי. יש לצייר את עץ ההחלטה של פרולוג ולראות לאן זה לוקח אותנו.

העיקרון הרקורסיבי הוא מאוד פשוט וניתן להשמה על מגוון בעיות. למרות פשטות העיקרון, ההשמה של הרעיון עבור בעיות שונות עלול להיות קשה יותר.

לאחר שמגיעים לשלב בו קל להבין בשלמות את הרעיון והמימוש, זה מספיק להסתכל בעיה ולהבין אותה בצורה יותר מופשטת: "אני יודע שהגדרתי מימן את כלל הבסיס, ואני יודע שהגדרתי כלל רקורסיבי נאות, שקורא לאותו פרדיקט שוב ושוב עם ארגומנטים פשוטים יוצר ויותר. ולבן, זה יעבוד. דבר זה הוא משום שהבנתי את העיקרון הרקורסיבי, אני מאמין בו, ואני מסוגל ליישם אותו. מעתה ועד עולם."

דיבאג

ב-SWI-Prolog יש אפשרות לדבג את תוכניות הפרולוג. זה *יכול* לעזור לך ולהבין כיצד שאילתות נפתרות (מצד שני, זה יכול גם מאוד לבלבל).

ניתן להשתמש ב-spy/1 על מנת לשים נקודות חקירה בפרדיקטים (מוקלדים בתוך האינטרפרטר בתור שאילתא, לאחר קימפול). לדוגמה:

```
?- spy(len).
Spy point on len/2
Yes
[debug] ?-
```

על מנת להבין טוב יותר כיצד לדבג בפרולוג, כדאי לבדוק את המדריך. להלן דוגמה לפרדיקט len/2 כמו שהוגדר מקודם.

```
[debug] ?- len( [dog, fish, tiger], X).
* Call: ( 8) len( [dog, fish, tiger], _G397) ? leap
* Call: ( 9) len( [dog, fish], _L170) ? leap
* Call: ( 10) len( [dog,], _L183) ? leap
* Call: ( 11) len( [], 0) ? leap
* Call: ( 10) len( [tiger], 1) ? leap
* Call: ( 9) len( [fish, tiger], 2) ? leap
* Call: ( 8) len( [dog, fish, tiger],2) ? leap
X = 3
Yes
```