

תכנות לוגי בפרולוג

מסוכם משיעורי מרן הרב
פרופ' יעקב הכהן-קרנר שליט"א

עם פירוש

"רי"ח טוב"

מאתי הצב"י יוחנן חאיק

שנת 0101100111111



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

להערות, הארות ותיקונים:
yohananha@gmail.com
yohanan@ - בטלגרם
ניתן להשתמש בסיכום באופן חופשי לכולם!!

תוכן עניינים

3	מושגים רלוונטים מלוגיקה
3	לוגיקה מסדר 0
3	לוגיקה מסדר ראשון Predicate logic
3	לוגיקה מסדר שני
3	חוקי גרירה לוגיים כלליים
4	פסוקיות הורן
5	תהליך ההאחדה
11	תוכניות למציאת אופטימום
12	ריקורסיות מתמטיות
16	שרשור
16	ראש וזנב
17	טעויות נפוצות בשרשור
17	רשימות עם איברים אנונימיים
17	פונקציית append
19	פונקציות ופרדיקטים מובנים על רשימות
21	Back-Tracking
22	בעיות עם Backtracking
23	כפיית נסיגה - Fail
23	CUT - !
25	סוגי CUT
25	בעיות עם CUT
27	הערות נוספות על קאט
28	רקורסיה לא-זנבית
29	רקורסיה זנבית
30	יתרונות של ר"ז על פני רל"ז
30	יתרונות של רל"ז על פני ר"ז
30	רר"מ - ריקורסיית רשימה מקוננת
32	פרדיקטים להחזרת קבוצת תוצאות
32	bagof/3
33	setof/3
33	findall/3
34	שימוש בתנאי מורכב
35	השמטת ערכים

36	פרדיקטים לבדיקת טיפוס
36	פרדיקטים ללוגיקת על
37	פרדיקטים אקסטרה-לוגיים
37	פרדיקטי קלט/פלט.....
38	פרדיקטי קלט/פלט מקובץ.....
38	ניהול זיכרון דינאמי.....
39	יצירת אינטרפטר

פרולוג – הכרת השפה

שפת פרולוג פותחה בשנת 1972 במרסיי במטרה לבוא לידי שימוש בבינה מלאכותית. השפה מכילה מנוע היסקים אוטומטי הבודק את כל האפשרויות הנתונות לבעיה שמוצגת, ונותן את התשובות הרצויות. בשונה משפות אחרות שאנחנו מכירים, אין בשפה תנאים של IF או SWITCH ודומיהם, אלא יש הגדרה של חוקים ועובדות, עליהם אנחנו יכולים לבדוק תשובות לשאלות נתונות. הרעיון של השפה היה חיקוי של תהליך חשיבה אנושי (בינה מלאכותית וזה), שעבור המידע הקיים אצל האדם, הוא יכול לנתח ולהכריז על דברים שהם אמת או שקר.

מושגים רלוונטים מלוגיקה

לוגיקה מסדר 0

לוגיקה מסדר 0 מוגדרת "תחשיב טעויות". זה החלק הפשוט ביותר שלמדנו בקורס לוגיקה – מגדירים מספר פסוקים אטומיים ונותנים לכל אחד שם, כאשר התחשיב בעצמו – החישוב האם ניתן לקיים כמה פסוקים במקביל או בנפרד, נעשה על יד האופרטורים השונים והגרירות.

לוגיקה מסדר ראשון Predicate logic

הרחבה של הלוגיקה מסדר 0, בה אנחנו מוסיפים גם תנאים מסויימים והרחבות נוספות. אפשר להגדיר משתנים (נהוג לסמן משתנה על יד אות גדולה), כאשר כאן אנחנו עדיין לא מדברים על כמתים של "קיים" או "לכל", ואנחנו מתייחסים לכל משתנה בתור "קיים". בנוסף, אנחנו יכולים להגדיר קבועים כלשהם (נהוג על ידי אות קטנה), כאשר אם אנחנו מגדירים קבוע, אנחנו מתייחסים אליו בתור לכל (כי הוא קבוע, אז הוא תמיד מתקיים, ולכן הוא "לכל"). שפת פרולוג מקבילה ללוגיקה מסדר ראשון.

לוגיקה מסדר שני

כאן אנחנו כבר מוסיפים את הכמתים של "קיים" (\exists) ו"לכל" (\forall), ואנחנו יכולים להשתמש בהם בשביל לכמת פרדיקטים שונים. "פרדיקט" זה יחס בין שתי ישויות. (אין שפות תכנות המתייחסות ללוגיקה מסדר שני).

חוקי גרירה לוגיים כלליים

בדרך כלל בלוגיקה, ובשפות התכנות הלוגי השונות, אנחנו מגדירים כל שורת קוד באופן הבא:

$$B_1, \dots, B_m \leftarrow A_1, \dots, A_n$$

כאשר הקריאה מתבצעת מימין לשמאל, כך שאם כל התנאים מימין מתקיימים, אזי יכול להיות שיתקיימו אחד מה-B-ים משמאל. כלומר –

$$\text{If } (A_1 \wedge \dots \wedge A_n) \text{ then } (B_1 \vee \dots \vee B_m)$$

כאשר כל הכללים הלוגיים חלים על שני הקצוות השונים של החוקים. מה הכוונה? מספר החוקים הנמצא בצד ימין/שמאל יכול ללמד אותנו קצת יותר ממה שאנחנו חושבים –

אם נתייחס לכמות הכללים מכל צד במספור של n, m כמו במשפט למעלה, נוכל לראות מספר קומבינציות שונות –

$n=0$

מקרה כזה מתאר לנו מצב שאין לנו מימין תנאי מקדים, כלומר מה שכתוב בצד שמאל יתקיים בכל אופן, או בניסוח אחר, P הוא עובדה. במידה ויש יותר מעובדה אחת שרשומה אז מדובר על קשר OR שאחד מהם צריך להיות אמת.

$n=1, m=0$

חוק כזה עבור P (שבעצם בכלל לא נמצא מצד שמאל), הוא שווה ערך ל- $n=0, m=1$ אך עבור שלילה של P . מצד שני, אפשר להתייחס לזה גם כשאלתא P ?

$m, n > 0$

זה חוק מהצורה שציינו קודם של אם ("חלק ימין") אז ("חלק שמאל").

$m=n=0$

סתירה או שקר.

פסוקיות הורן

את הפרולוג שכללו עם הזמן להיות בצורה המכילה "פסוקיות הורן", שעל פי החוק של פסוקיות אלו, באגף השמאלי מותר שיהיה אך ורק פרדיקט בודד, ומימין יכול להיות כמה שירצו. כדאי לשים לב שאנחנו מדברים על חץ גרירה ← כאשר בכתיבה ברוב האינטרפרטרים אנחנו בכלל מסמנים :- אבל זה רק עניין סמנטי.

אם נתייחס לחלקי החוקים בעצמם, אנחנו מחלקים אותו למספר חלקים קטנים-

$$P1(X_1 \dots X_n) :- q_1(Y_1, \dots Y_n), q_n(Z_1, \dots Z_k)$$

החלק משמאל לחץ הגרירה הוא הראש של החוק (Head), או היעד (Goal).

והחלק הימני בכללותו נקרא בהתאמה הזנב (Tail), או תתי-היעדים (Sub-Goal).

כל חוק אנחנו מחלקים לעוד תת-חלוקה נוספת - Functor - שהוא ראש היעד, הוא השם שלו (כאשר למעשה אלו אותם פאנקטורים גם בצד השמאלי), ומספר הארגומנטים שאנחנו מכניסים לתוך כל חוק שכזה נקרא Arity.

כך שעל מנת לכתוב עובדה פשוטה, פשוט נכתוב אותה באופן הבא -

head().

וזה שקול באופן מלא פשוט לביטוי

head():-true.

עכשיו אנחנו צריכים לדון על שני דברים נוספים - AND, OR. האם אנחנו יכולים לכתוב אותם בצורה פשוטה? בוודאי. AND עצמו מובנה במערכת, כך שכל התנאים מצד ימין מחוברים ביניהם עם AND, ואת זה כבר ראינו. מה עם OR?

למשל עבור $C \leftarrow A \vee B$ נוכל לכתוב את התוכנית הבאה-

C:-A

C:-B

וכך אם רק אחד מהתנאים יחזיר לנו true, זה יספיק לנו.

עכשיו נשאר לנו לשאול את אותה השאלה, אבל בצד השמאלי של התנאי. אמנם אמרנו שאנחנו מתירים מצד שמאל רק חוק אחד בכל פעם, אבל אם בכל אופן נרצה לכתוב $B \wedge C \leftarrow A$, איך נעשה את זה? נוכל לכתוב את התכנית הבאה:

B:-A

C:-A

ואז אם יש לנו מקרה ש-A מתקיים, אז גם B וגם C יתקיימו בהתאם.

דבר אחרון שנותר לנו לאמת או אפשרות של OR בחלק השמאלי. על פי חוקי דה-מורגן וחוקי לוגיקה שלמדנו, $\neg q \rightarrow q \equiv p$. ולכן, אם נרצה לכתוב $B \vee C \leftarrow A$, נוכל לעשות את ההמרה הבאה:

$$B \vee C \leftarrow A \equiv \neg A \leftarrow \neg(B \vee C) \equiv \neg A \leftarrow \neg B \wedge \neg C$$

ואת השקילות האחרונה אנחנו כבר יודעים לכתוב בצורה של תוכנית בפרולוג באופן הבא:

$\neg A :- \neg B, \neg C.$

עכשיו אחרי שעברנו על הבסיס הלוגי, נוכל לדבר קצת על תהליך העבודה של השפה, וכיצד אנחנו מזהים פרדיקטים.

תהליך ההאחדה

ניתן לבדוק עבור כל תכנית מספר עובדות. למשל, אנחנו יכולים לכתוב את העובדה הבאה-

male(avi).

ואז לשאול בחלון ההרצה האם זה מתקיים-

?- male(avi).

true.

תהליך ההאחדה, עובר ומחפש בין העובדות הקיימו את ה-Functor המתאים לשאילתא. ברגע שנמצא male, אנחנו בודקים עם ה-Arity מתאים, ושיש לנו מספר ארגומנטים מתאים לאותו Functor שמצאנו – כי יכול להיות שעשינו מעין "העמסת פונקציות", ובעבור אותו Functor, הכנסנו מספר Arity משתנה. לאחר שנמצא חוק עבורו יש גם מספר מתאים של ארגומנטים, אנחנו מתחילים לרוץ על העובדות ולבדוק האם ידוע לנו חוק כמו שהתבקשנו לבדוק בשאילתא. ברגע שיימצא לנו אחד מתאים, יוחזר לנו true.

בנוסף, אנחנו יכולים לשאול את השאילתא בצורה של השמת משתנה. אם אנחנו לא יודעים מה הם הערכים הקיימים, אנחנו יכולים לשלוח את השאילתא האה:

?- male(X).

כך שברגע שיימצא לנו חוק מתאים מבחינת ה-Functor וה-Arity, הוא יחזיר לנו ערך שאפשר לשים ב-X על מנת לספק את התשובה הרצויה, במקרה שלנו –

X = avi

מה קורה במידה ויש מספר ארגומנטים שיכולים לספק את השאילתא? האינטרפטר ישאיר לנו את האפשרות לעבור על כל העובדות המספקות את השאילתא, כך שאם יש לנו מספר גבוה יותר, נוכל לקבל את כולם אחד אחרי השני.

תהליך ההאחדה יכול לבוא בצורות שונות. ברגע שנמצא איזה Head שיקיים את מה שנשאל אותו, אנחנו יכולים לבדוק עוד דברים, ונראה מספר דוגמאות לדברים שאפשר לבצע-

דוגמא 1

worker(dan, cohen, Age).

?- worker(First, cohen, 40).

First = dan.

Age = 40.

כאן אנחנו יכולים לראות משהו מיוחד, את הגיל (Age) בעובדה הגדרנו בתור משתנה (אפשר לשים לב, שבניגוד לשני הארגומנטים הראשונים, הארגומנט של הגיל מתחיל באות גדולה, מה שמראה שהוא בכל משתנה ולא קבוע).

הדבר המיוחד שההאחדה קיימה פה, היא האחדה דו-כיוונית. ברגע שמצאנו משהו שייקיים worker/3, כלומר Functor של worker עם Arity=3, הוא חיפש את האחד שייקיים את אחד מהתנאים שלו, ואכן נמצא לבסוף שיש לנו התאמה בין cohen=cohen. ברגע זה אנחנו עושים גם הצבה על First שנשלח כמשתנה לפונקציה והוא מקבל כי First = dan, וגם המשתנה של העובדה, מקבל השמה מאחר ויש לנו פה תנאי שמתקיים, ו-Age = 40.

דוגמא 2

worker(dan, cohen, Age).

worker(moshe, levi, 40).

?- worker (First, cohen, 40).

First = dan

Age = 40;

False

?

מה קרה פה? מבחינת הגדרת העובדות, עשינו כמעט אותו דבר, רק שאת משה לוי, הגדרנו גם אם הגיל שלו מראש. לאחר מכן, השאילתא עוברת ומוצאת את דן כהן, בדיוק כמו בתכנית הקודמת, אבל ממשיכה לבדוק אחר כך אם יש עובדות נוספות שעלולות לקיים את השאילתא (כמו שראינו קודם, שאפשר לעשות male(X) ולקבל את כל התשובות האפשריות). אבל מאחר ולא נמצאה שום אופציה נוספת, הוא חזר false, ופשוט איפס את עצמו לקראת קבלה של שאילתא נוספת.

דוגמא 3

worker (dan, cohen, Age).

worker (moshe, cohen, 40).

?- worker (First,cohen,40).

First=dan

Age=40;

First= moshe

?

כאן, בהחלט יש לנו שתי עובדות שמשתייכות לכהן. מה שמיוחד פה הוא שאחד מהם כבר מכיל את הגיל המתאים, ולכן לאחר שהוא מגלה את משה כהן, אין לו צורך לעשות השמה, והוא מחזיר רק את השם הפרטי שלו.

אפשר לשים לב, שבדוגמה הקודמת הייתה לנו את השורה false, שכבר לא מופיעה פה. למה כאן זה שונה? מאחר והפרולוג עבר על כל החוקים האפשריים ובדק את הנכונות שלהם, אז אין לו כבר צורך לחזור ולשאול האם אנחנו מעוניינים בכך שהוא ימשיך את הבדיקות.

דוגמא 4

worker (dan, cohen, Age).

worker (moshe, cohen, 40).

worker(Last, cohen, Kuku).

?- worker (First, cohen, 40).

First=dan

Age=40;

First= moshe;

First=Last

Kuku=40

?

פה אין הרבה מיוחד, רק יש לשים לב, שההשמה האפשרית אינה רק למשתנה אחד. את ה-40 שהכנסנו לשאילתא, אנחנו יכולים לבדוק הן עם Age, והן עם Kuku. ולהשתמש בכל אחד מהמשתנים באופן אחר.

דוגמא 5

parent(avr, yit).

parent(yit, yaak).

parent(yaak, yosef).

parent(yaak, binyamin).

?- parent(yaak, X).

X=yosef;

X=binyamin

?

?- parent (yaak, _).

yes

?

?- parent(X, Y).

X=avr

Y=yit;

X=yit

Y=yaak;

X=yaak

Y=yosef;

X=yaak

Y= binyamin

?

סיכום פרולוג – יוחנן חאיִק

כאן אנחנו מגדירים רשימה של עובדות ביחס אבות ובנים. ועל זה אנחנו מריצים סדרת שאילתות – קודם כל אנחנו בודקים מי הם הבנים של yak. תהליך ההאחדה מזהה את האפשרות להציב ב-X שני ערכים שונים.

השאילתא השניה, נראית כמעט אותו דבר, רק שבמקום לדרוש השמה עם משתנה אנחנו משתמשים בקו-תחתון, הסימן הזה, מסמל בפרולוג את ה"Don't Care" שמשמעותו ברמת השאילתא הזאת היא "האם קיים מישהו שהוא הבן של יעקב?". אנחנו לא מתעניינים בלדעת מי הילד המדובר, אלא שאלים בצורה בוליאנית האם יש לו או אין לו ילדים. מהסתכלות פשוטה על העובדות קל לראות שיש לו ילדים. הלאה.

השאילתא הבאה, מבקשת את כל הזוגות המתאימים (X,Y) המקיימים את השאילתא ויש ביניהם קשר parent, ולכן מחזירה הלוך ושובת את כל האפשרויות להציב בזוג הזה את העובדות השונות.

לאחר שעשינו את כל זה, אנחנו נדבר גם על עובדות וחוקים – נוסיף את החוק הבא לשרשרת החוקים הקודמת:

$\text{grandparent}(X, Y) :- \text{parent}(X, Z), \text{parent}(Z, Y).$

כעת אנחנו אומרים שאם יש לנו הורה "טרנזיבי", כלומר אבא של אבא, אנחנו מגדירים את הקשר הזה בשם המפתיע "סבא". ואנחנו יכולים לחפש זוגות של סבא ונכד באופן הבא-

$?\text{-grandparent}(T,D).$

T=avr, D=yaak;

T=yit, D=yosef;

T=yit, D=Binyamin

?

תהליך ההאחדה עובד פה באופן כזה, שהוא מחפש קודם כל זוגות (X,Z), ברגע שהוא מוצא זוג כזה, הוא עובר הלאה לחלק הבא, תוך כדי שהוא "זוכר" שהוא כבר הציב את הזוג הקודם, ועכשיו הוא מתחיל לבדוק שוב עבור הזוג (Z,Y). ברגע שהוא יזהה גם חלק כזה, הוא יוכל להגיד שיש לנו את שני הקשרים האלה שיתקיימו, והוא יכול להכריז על כך שיש לנו סבא. איך הוא יראה את זה? יבטא לנו צמידים אפשריים של (סבא, נכד).

פעולות אריתמטיות בפרולוג

הפרולוג מביא לנו מספר פונקציות ואופרטורים עליהם אנחנו יכולים לעבוד בצורה אריתמטית פשוטה, כמו בכל שפה. נעבור בזריזות על האופרטורים השונים, ונראה את משמעותם, כאשר נעצור להסביר את החלקים החשובים בהם.

$X+Y$

$X-Y$

$X*Y$

$X**Y \% (X^i)$

ארבעת הפונקציות הקודמות הן די בסיסיות, אבל כאשר אנחנו מתקרבי לתחום החילוק, אנחנו מסתעפים לכמה כיוונים –

X/Y

$X//Y$

$X \text{ mod } Y$

קודם כל יש לנו את החילוק כ- float . אם נציב ל- x את השאילתא X/Y אנחנו נקבל את התוצאה כ- float . אם נרצה לעומת זאת את הפתרון ב- int נשתמש בשני סימני חילוק ($//$), ואם נרצה דווקא את השארית, נשתמש בפונקציית המודולו. רק לצורך הבנת ההבדל נראה את שלושת הווריאציות על אותם מספרים –

$7/3 = 2.33333333$

$7//3 = 2$

$7\%3 = 1$

מעבר לזה יש את כל ההשוואות בין משתנים -

$X<Y$

$X>Y$

$X==Y$

$X!=Y$

$X>= Y$

$X<= Y$

כאן יש לנו שוני קטן בעניין ההשוואה שאנחנו רגילים לעשות " $==$ ", פה אנחנו צריכים להוסיף את הנקודתיים באמצע להשוואה ואת הקו-הנטוי במידה ואנחנו בודקים אי-שוויון.

נסתכל כעת על ההשמה, שיש בה שוני מאוד גדול ממה שראינו עד עכשיו, ונסביר אותו -

$X \text{ is } 6 + 2$

$X = 6 + 2$

$kuku(X, Y):- X \text{ is } 2+6, Y = 2+6.$

? $kuku(X, Y).$

$X = 8$

$Y = 2 + 6$

עלינו לזכור שבפרולוג אין לנו טיפוסים כמו שאנחנו רגילים אליהם משפות אימפרטיביות ומונחות עצמים כמו $c++$ ודומיהם, ודבר זה מוביל להבדל גדול שעלול ליצור בעיות. השמת מספרים או תוצאה לתוך משתנה אינה נעשית עם

סיכום פרולוג – יוחנן חאיך

התו "=" , אלא עם האופרטור is. אם אנחנו ננסה לעשות השמה רגילה כמו $X=6+2$ מה שיוצב לנו זאת המחרוזת "6+2" ולא התוצאה.

רק לצורך ההרחבה, אוסיף את מה שנכתב על האופרטור בחוברת של פרופ' אנדריס¹:

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

אם לא נאמר לפרולוג בצורה מפורשת שאנחנו מעוניינים בהערכה אריתמטית של ביטוי, הוא יסתכל על הביטוי 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 לבדוק 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 בSWI-Prolog). מה שאומר, לדוגמא, שאם נשאל לגבי $0.5+0.5$ is 1.0 נקבל תשובה שלילית, מאחר והתוצאה של $0.5+0.5$ תחזיר את הערך 1 ולא את הערך 1.0 . בגדול, עדיך להשתמש באופרטור $=$ במקרה שבו הביטוי השמאלי מוגדר כבר כמספר.

¹ שתרגמתי בשנה קודמת, אך אינה מותאמת ממש למה שאנחנו לומדים.

בנוסף יש לנו גם ערכים קבועים במערכת, שיהיה לנו יותר נוח לעבוד איתם – כדאי לשים לב שהשם שלהם הוא באותיות קטנות ולא גדולות, מאחר ומדובר בקבועים, ולא במשתנים, מלבד הראנדום שהוא פונקציה ולא קבוע –

π ; Evaluates to the mathematical constant π (3.141593).

e ; Evaluates to the mathematical constant e (2.718282).

Random

אם נרצה נוכל להגדיר פרידקט משלנו העושה השוואה בין שני משתנים (שאינן מספרים פשוטים, שהם כבר מובנים במערכת). לצורך זה, נגדיר את הפונקציה $\text{diff}/2^2$. כמובן שחשוב לוודא שאנחנו עושים את זה בצורה נכונה. נראה דוגמה לא נכונה, ונוכל להבין ממנה איך לכתוב בצורה מוצלחת יותר –

$\text{diff}(X, Y) :- \text{diff}(Y, X)$.

לוגית אולי הנתון הזה אולי נכון, אבל הוא עלול לגרום לנו רקורסיה שתיכנס לעצמה בלולאה אינסופית. בכל פעם אנחנו ניכנס שוב ושוב בשביל לבדוק את שני האיברים שבכל פעם יחליפו צד לבדיקה, ואף פעם לא נעצור. איך נתקן את זה?

$\text{diff}(X, Y) :- \text{not}(X=Y)$.

כמובן, שגם היינו יכולים פשוט לכתוב רק את החלק הימני וזה היה מספיק לנו, אבל לפעמים נידרש לכתוב את הפרדיקטים שלנו בעצמנו, ונשתמש בכלים העומדים לרשותנו. עכשיו ננסה להריץ כמה שאילתות על הפרדיקט שכתבנו, ונראה שאכן הפרולוג עושה את השוואה המתבקשת.

?- $\text{diff}("a", "a")$.

no

?- $\text{diff}("a", "b")$.

yes

?- $\text{diff}(ab, ab)$.

no

?- $\text{diff}(ab, ba)$.

yes

חוק אריתמטי נוסף שחשוב לעמוד עליו – בפרולוג אנחנו לא משנים משתנה על ידי עצמו. כלומר – אנחנו לא יכולים לכתוב משהו בסגנון $X = X+1$, וכמובן שגם לא $X++$ (שזה רק קיצור של הביטוי הקודם), אלא אם נרצה לשנות ערך במשתנה ולהעביר אותו הלאה, אנחנו צריכים ליצור משתנה חדש, להכניס לתוכו את החישוב הדרוש, ואותו להעביר הלאה, באופן הבא:

$X1$ is $X+1$.

לאחר השורה הזאת אנחנו יכולים להעביר את $X1$ ללא חשש הלאה.

תוכניות למציאת אופטימום

ככלל, יש לפרולוג פונקציות של מינימום ומקסימום, אך הן לא עובדות תמיד באופן שאנחנו רוצים (מבחינת ההגדרה שלהן), ולכן ננסה להגדיר את $\text{min}/3$ בתור פונקציה שמקבלת שני ערכים ומחזירה את המינימלי מביניהם לערך השלישי.

$\text{min}(X, Y, Z) :- X \leq Y, Z \text{ is } X$.

² בתיעוד הרשמי של התוכנית (או אם סתם תחפשו חומר על הפרולוג), מקובל לרשום כל שם פונקציה עם קו נטוי ולאחרי מספר, המבטא את אריות – Arity – כמות הארגומנטים הנכנסת לפונקציה. הרבה פעמים ניתן לראות שיש הבדלים בין פונקציה עם אותו שם, כאשר השוני יהיה רק מספר הארגומנטים.

$\min(X, Y, Z)$:- $Y < X, Z \text{ is } Y$.

ברגע שנכניס שני ערכים, אנחנו נבדוק קודם כל $X \leq Y$, כי גם אם הם שווים זה בסדר מבחינתנו, ואם זה לא נכון, אז אנחנו נלך בתהליך ההאחדה ונחפש תנאי נוסף שעלול לקיים לנו את התנאי. כאשר נראה את $Y < X$, ונראה שהוא נכון, נוכל להציב בתוך Z את Y , ולהחזיר את התשובה.

כדאי לשים לב, אנחנו עלולים לחשוב ולומר, שהתנאי השני הוא מיותר, כי אם התנאי הראשון לא מתקיים, בוודאי שהשני מתקיים, אז אולי אנחנו יכולים לכתוב את התוכנית באופן הבא –

$\min(X, Y, Z)$:- $X \leq Y, Z \text{ is } X$.

$\min(X, Y, Z)$:- $Z \text{ is } Y$.

דבר כזה עלול לגרום לטעות. למה? דיברנו קודם בתהליך ההאחדה, על כך שיכול להיות יותר מתשובה אחת לשאלות מסוימות. מה יקרה אם נכתוב את השאלתא הבאה? –

?- $\min(2, 5, Z)$.

דבר ראשון, אנחנו נבדוק את השורה הראשונה, מה שיחזיר לנו כבר את הערך $Z = 2$. אבל אז יש לנו אפשרות לעשות next, מה שיגרום לפרולוג להמשיך ולבדוק, ואם הגדרנו את השורה השניה ללא שום תנאי, הוא פשוט יכניס את T לתוך Z , ויחזיר לנו $Z = 5$, שזה בוודאי כבר לא נכון. אנחנו נראה בהמשך, איך אנחנו יכולים לייעל קצת את התכנית שלנו, אבל קודם כל עלינו להבין את הבעיה הנוכחית.

התכנית למציאת מקסימום יכולה גם כן להיות מאוד דומה –

$\max(X, Y, Z)$:- $X > Y, Z \text{ is } X$.

$\max(X, Y, Z)$:- $Y > X, Z \text{ is } Y$.

או לחילופין, אנחנו יכולים לייעל את הפונקציה, אבל מכיוון אחר לגמרי מזה שעבדנו עליו קודם, ולבטל את המשתנה Z . זה לא משמעותי מלבד הקצאת המקום והקצת-זמן שלוקח לעשות את הפעולה, אבל אם אפשר אנחנו תמיד מעדיפים לייעל קצת תוכניות –

$\max(X, Y, X)$:- $X > Y$.

$\max(X, Y, Y)$:- $Y > X$.

ה"טריק" הזה הוא דבר שמשתמשים בו הרבה, ופשוט במקום לקחת את Z אנחנו מגדירים את הערך המוחזר כבר בכותרת. אמרנו הרי שבדקים את התנאי מימין, ורק אם הוא מתקיים, בודקים את התנאי משמאל, ולכן לאחר שאנחנו מגיעים לתנאי שמתקיים לנו, הראש של החוק פשוט מחזיר את הערכים כמו שהם ועושה את ההצבה ברקע.

ריקורסיות מתמטיות

כמו בכל שפה, גם פה ניתן לעשות ריקורסיות. אנחנו יכולים בתוך אחד החוקים לתת אפשרות לבדיקה של אותו חוק בעצמו, אך עם שינויים קלים בארגומנטים וכך לחשב דברים אחד בתוך השני. כמובן, שעלינו לשים לב שאנחנו עובדים לפי כל חוקי הרקורסיה שאנחנו כבר מכירים.

נראה מספר דוגמאות פשוטות של ריקורסיות, ובהמשך ניכנס קצת יותר להגדרות של סוגי ריקורסיה, והשימושים השונים בה.

כהמשך לדוגמאות של הפונקציות המתמטיות, נראה עוד כמה אופציות לכתיבה, רק בצורה ריקורסיבית.

עצרת

$\text{fact}(0, 1)$.

$\text{fact}(X, Y)$:- $X > 0, X_1 \text{ is } X - 1, \text{fact}(X_1, Y_1), Y \text{ is } X * Y_1$.

לא ממש צריך להסביר מה זה עצרת, אז רק נראה את הצורה של הכתיבה. קודם כל, בשורה הראשונה יש לנו תנאי עצירה, מאחר ובכל כניסה מחדש לרקורסיה, אנחנו מתחילים מהחוקים הראשונים ומתחילים לרדת כלפי מטה.

כדאי לשים לב – מבחינת זמן ריצה, לא תמיד נכון לשים את כלל העצירה ראשון, מאחר וזה מכריח אותנו לבדוק את השורה הזאת בכל איטרציה, ואם אנחנו נכנסים לעומק של 10 רקורסיות, אנחנו בודקים את התנאי הזה 10 פעמים, אבל למעשה נצרכים אליו רק פעם אחת. אבל כרגע נעשה את זה ככה, בהמשך נראה איך אנחנו יכולים לשלוט יותר בזרימת התכנית ונפעל בהתאם.

בנוסף, תנאי העצירה לא בודק כלום מצד ימין, אלא בודק רק האם מה שנשלח אליו הוא בפורמט של (0,1). במידה וכן, הוא פשוט מחזיר true, ואנחנו מתחילים לחזור במעלה הקריאות.

בשורה השניה, אנחנו עובדים על תוכנית העצרת עצמה – התכנית מקבלת X שמהווה את החסם העליון של העצרת, ומחזיר את התוצאה ב-Y. קודם כל, הוא בודק ש- $X > 0$, שזו בדיקה שכמובן רלוונטית בעיקר לפעם הראשונה, כי במהלך הריצה הרגילה $X = 0$ זה תנאי עצירה. אחרי זה הוא מוריד את הערך של X לטובת ההכפלה העצרתית, וקורא שוב לפונקציה עם $X1$ שעבר שינוי, ועם Y1 שבכלל עדיין לא הוגדר. איך אנחנו יכולים לעשות דבר כזה? Y1 שנשלח מטה ברקורסיה, הוא בעצם דומה ל-Y שנשלח בקריאה המקורית. התוכנית פשוט מקצה מקום בזיכרון ל-Y1, וממשיכה הלאה.

בסוף הריצה, לאחר שנחזור מתנאי העצירה, אנחנו חוזרים עם $X1 = 0, Y1 = 1$, שזו ההצבה של תנאי העצירה, ואז ברמה אחת מעל, אנחנו מכניסים לאותו Y את X (של הרמה העליונה יותר) מוכפל ב-Y1 שהוא בעצם תוצאת ההכפלה ברמה מתחת. ואז אנחנו משחררים את הרמה הנוכחית ומחזירים את Y עם הערך החדש רמה אחת כלפי מעלה (שם הוא היה Y1), וכן על זה הדרך.

הערה נוספת – אמנם אמרנו שהתנאי הראשון $X > 0$ מוכרח בעיקר בתנאי הראשון, אבל לאחר שנסיים את הריצה, תופיע לנו אופציה להמשיך עם next. זאת מאחר, שתנאי העצירה בדק רק את החוק הראשון, ומבחינת הפרולוג הוא צריך להציע לנו לעבור על עוד אופציות, אם ניתן לו להמשיך לנסות הוא יעבור הלאה, כך שאם לא היה לנו התנאי שבדק שאנחנו מעל 0, היינו נכנסים לעוד סבב שלעולם לא היה נגמר מאחר ולא נגיע לתנאי העצירה. ברגע ששמנו את התנאי $X > 0$, גם אם ניתן לפרולוג להמשיך, הוא יחזיר לנו false, ונצא בבטחה מהתוכנית.

כפל

$\text{times}(X, o, o)$.

$\text{times}(X, S, Z)$: - $S > 0, S1 \text{ is } S-1, \text{times}(X, S1, Z1), Z \text{ is } Z1+X$

למעשה, את כל הפונקציות המתמטיות, ניתן לבצע באופן דומה למה שעשינו את העצרת. מבחינת הדרוש לנו לביצוע נכון של פונקציה רקורסיבית, אפשר למנות מספר שלבים עיקריים, ולראות איך אנחנו מבצעים אותם על מספר פונקציות שונות:

1. תנאי עצירה - $(X, 0, 0)$, לוודא שלא נרוץ לנצח. בדרך כלל נדאג עם תנאי העצירה לבצא את פעולת ההשמה המתאימה, במקרה שלנו ברגע ש- $S=0$, אז אנחנו עוצרים ומציבים גם ב-Z 0.
2. בדיקת חוקיות - $S > 0$, או במקרה הקודם שבו זה היה X. אם אנחנו עובדים על משהו שהוא בכלל לא רלוונטי, חבל להתחיל.
3. שינוי רקורסיבי - $S1 \text{ is } S-1$, אם אנחנו לוקחים משתנה ומשנים אותו באופן קבוע תוך כדי הירידה, עלינו לבצע את השינוי מייד לפני שנרצה להשתמש בו. כדאי גם לוודא שלא עושים את זה מוקדם מידי וסתם עובדים לשווא.
4. קריאה רקורסיבית - $\text{times}(X, S1, Z1)$, הכנסת הארגומנטים הרלוונטיים לקריאה מחדשת.
5. ביצוע פעולת החישוב - $Z \text{ is } Z1+X$, לאחר שחוזר לנו הערך הנכון מהרקורסיה, אנחנו מחשבים אותו הלאה, ומסיימים את הפונקציה.

חזקה

$\text{exp}(X, 0, 1)$.

$\text{exp}(X, S, Z)$:- $S > 0$, S_1 is $S-1$, $\text{exp}(X, S_1, Z_1)$, **Z is $Z_1 * X$** .

לאחר שהגדרנו את פונקציית הכפל הקודמת, אנחנו יכולים למעשה להחליף את החלק המודגש בקריאה לפונקציה באופן הבא – $\text{times}(Z_1, X, Z)$, ולקבל את אותו דבר.

מודולו

$\text{mod}(X, Y, X)$:- $X < Y$.

$\text{mod}(X, Y, Z)$:- $X \geq Y$, X_1 is $X - Y$, $\text{mod}(X_1, Y, Z)$.

גם פה אין איזה שינוי גדול ממה שהיה עד עכשיו, מלבד ההצבה בסוף. המודולו שלנו בעצם כל פעם מחסיר מה-X הנוכחי את Y עד שהוא יגיע לשארית הרצויה, ואז יחזיר אותה בתור ערך Z. לצורך החוויה, נעשה איזה ריצה קטנה על המודולו, ונראה איך זה עובד:

$\text{mod}(17,5,Z)$.	// $17 \geq 5$, X_1 is 12
$\text{mod}(12,5,Z)$	// $12 \geq 5$, X_1 is 7
$\text{mod}(7,5,Z)$.	// $7 \geq 5$, X_1 is 2
$\text{mod}(2,5,2)$	// $2 < 5$ ($Z=X$)
$\text{mod}(7,5,2)$.	
$\text{mod}(12,5,2)$.	
$\text{mod}(17,5,2)$.	

Z=2.

רשימות

הרשימות בפרולוג הן מבני נתונים נוחים מאוד לעבודה. הרשימות תחומות בתוך סוגריים מרובעות [], ויכולות לבוא במגוון צורות וסגנונות. ונראה מספר דוגמאות כדי להבין את האופציות השונות –

רשימה ריקה – [] – רשימה לא חייבת להכיל איברים, ויכולה להיות ריקה. אנחנו משתמשים ברשימות ריקות, במהלך רקורסיות וכדו', ואחת כך אנחנו מוסיפים משתנים, ומבצעים מניפולציות שונות.

תכולת הרשימה – נראה מספר רשימות בעלות איבר אחד בכל אחד מהן, ונראה מה אנו לומדים שם –

[7], [a]	// אין הבדל בין אותיות למספרים, הכל נשמר באותו אופן//
[[]]	// איבר ברשימה יכול להיות רשימה אחרת, אפילו אם היא ריקה //
[[7,5]]	// הרשימה הפנימית יכולה להכיל גם איברים //
[[[],[]]]	// הרשימה הפנימית יכולה להכיל רשימות נוספות, אך זה עדיין נחשב שיש רק איבר אחד ברשימה המקורית //
[_]	// רשימה יכולה גם להכיל (או להתייחס ל) איבר אנונימי //

באופן דומה, אנחנו יכולים להסתכל על רשימות עם שני איברים (או יותר) –

[1,2]
[a,b]
[a,[]]
[-,-]

הרעיון בכל אלו דומה מאוד ואין צורך להרחיב.

שרשור

לצורך תחילת הדין על שרשורים ברשימות, נביא את דברי פרופ' אנדריס –

ראש זנב

האיבר הראשון ברשימה נקרא ראש וכל שאר הרשימה מוגדרת כזנב. 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 על ידי שימוש בקו המפריד |. בכל מקום בו הוא מונח ברשימה, הוא מגדיר את כל החלק שאחריו להיות רשימה חדשה. הרשימה החדשה נבנית באופן שתת-הרשימה לפני הקו יהיה הראש. ובמידה ויש רק איבר אחד לפני הקו הוא יהיה הראש והשאר יהיה הזנב. בדוגמה הבאה, המספר 1 יהיה הראש, והרשימה [2,3,4,5] יהיה הזנב, שיחושב 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 Head ו-Tail הם שמות של משתנים. אנחנו יכולים להשתמש גם ב X או Y או כל דבר אחר שנרצה להגדיר בעצמנו. חשוב גם להדגיש שזנב הרשימה (או ליתר דיוק: החלק המופיע לאחר הסימון |), תמיד יהווה רשימה בפני עצמו. 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

באופן פשוט, כאשר אנחנו רוצים להרכיב רשימה חדשה, מאחת שכבר קיימת לנו, אנחנו פשוט תוחמים את הכל כרשימה חדשה, כאשר בזנב הרשימה תעמוד הרשימה הישנה, ואליה נשרשר את האיבר החדש שאנחנו רוצים להוסיף. נראה מספר דוגמאות לעניין –

[a | [b]] = [a,b]

[[a][b]] = [[a],b] // מאחר והכנסנו רשימה ל"ראש", הוא יישאר באיבר של רשימה

[a, b, c] = [a | [b, c]] = [a | [b | [c]]] = [a | [b | [c | []]]] = [a, b | [c]] = [a, b, c | []]

// כל השרשורים למעלה הם חוקיים ומבטאים את אותו דבר בדיוק³

[a | [b] | [c]] = [a, b],c]

[] ≠ [[]] ≠ [[] , []] ≠ [[[]]]

// צריך לזכור – רשימה היא איבר גם אם היא ריקה, ולכן
// רשימה ריקה אינה שווה לרשימה המכילה רשימה ריקה

³ למרות שיכול להיות שבחלק מהפרולוגים זה לא חוקי לעשות יותר משרשור אחד בו-זמנית, אבל מבחינה רעיונית מדובר על אותו דבר.

טעויות נפוצות בשרשור

- [2 | 3] // אי אפשר לשרשר שני אטומים, כאשר הזנב המשורשר אינו רשימה
- [2 | [3], [4]] // ניתן לשרשר רק לרשימה בודדת
- [[2]] // אין אפשרות לשרשר "כלום". ניתן לשרשר רשימה ריקה, אבל היא איבר

רשימות עם איברים אנונימיים

כמו שראינו קודם, אנחנו יכולים להשתמש באיברים אנונימיים לצורך בדידה של חוקים וכדו'. אנחנו יכולים לעשות זאת גם עם רשימות. למה זה חשוב? כי איבר אנונימי יותר נוח לטיפול מבחינת הפרולוג וחוסך בזמן ריצה ומקום בזיכרון. כל הדוגמאות שנביא כרגע, רלוונטיים בעיקר לבדיקות חוקים, ונשתמש בהם באופנים שונים –

- [_ | _] // אנחנו יכולים לבדוק ששני הצדדים יהיו אנונימיים
- [_ | T] // בדיקה שהרשימה לא ריקה לגמרי – בראש יש משהו וגם בזנב
- [H | _]
- [2 | _] // כאן אנחנו בודקים שהראש שווה לערך מסוים
- [5, 7 | _] // הראש, מבחינתנו, יכול להיות גם מספר איברים, ולכן אפשר לבדוק ריבוי איברים
- [H1, H2 | _] // בדיקה שברשימה יש שני איברים לפחות
- [_, _, 17, _, _] // הרשימה מכילה לפחות 5 איברים כשהשלישי הוא 17

בגדול, אפשר להמשיך ולשחק עם זה, אבל נשים לב שבעצם יש לנו כאן 3 סוגי בדיקות:

- קיים (3) – אם אנחנו שמים רק איבר אנונימי, אנחנו רוצים לראות שיש **משהו** ברשימה, ואין לנו שימוש בו בהמשך.
- שימוש בקיים – כאשר אנחנו שמים משתנה בודד, אנחנו מבצעים שתי פעולות – שקיים משהו, כמו בבדיקה הקודמת. והדבר השני – את מה שקיים אנחנו נשים במשתנה ונוכל לעבוד איתו אחר כך. שוב, אם אנחנו לא הולכים להשתמש בראש, אין לנו סיבה להציב אותו, זה סתם תופס לנו זיכרון מיותר.
- קיים מסוים – אנחנו לא רוצים להשתמש במה שנמצא, אבל חשוב לנו לדעת שיש בו ערך מסוים. דבר זה יעיל מאוד לתנאי עצירה, ברגע שהראש הוא ריק, או שיש בו איבר מסוים שאנחנו מעוניינים בו, נעשה את הפעולות הדרושות.

פונקציית append

append([], Z, Z).

append([H|X], Y, [H|Z]):- append(X, Y, Z).

הפונקציה append מוגדרת לקבל שתי רשימות, ולהוציא אותן משורשרות אחת לשניה בתוך המשתנה השלישי.

אופן הפעולה (הרגיל) שלה הוא כזה – הרשימה הימנית נשארת ללא שינוי עד הסוף. ברשימה השמאלית, אנחנו כל פעם לוקחים את הראש, שומרים אותו לשימוש עתידי, ומכניסים לרקורסיה את הזנב ואת שתי הרשימות הנותרות. דבר זה נמשך, עד שמגיעים לתנאי העצירה, שמעתיק את הרשימה הימנית בשלמות לתוך Z, ובכל רמה שאנחנו עולים ברקורסיה, אנחנו מדביקים עוד "ראש" שחתכנו מהרשימה השמאלית, כך שהרשימה הולכת ונבנית עד שאנחנו יוצאים מהפונקציה.

נראה עכשיו הרצה פשוטה על יבש של הפונקציה –

append([1,2],[3,4],Z)

Call: append([1, 2], [3, 4], _3452)

Call: append([2], [3, 4], _3644)

// Z = [1|Z']

Call: `append([], [3, 4], _3650)` // `Z' = [2|Z']`
Exit: `append([], [3, 4], [3, 4])` // `Z'' = [3,4]`
Exit: `append([2], [3, 4], [2, 3, 4])` // `Z' = [2|3,4]`
Exit: `append([1, 2], [3, 4], [1, 2, 3, 4])` // `Z = [1|2,3,4]`

`Z = [1, 2, 3, 4]`

(העתקתי את זה מתוך החלון ריצה של הפרולוג אונליין. ניתן לבצע דיבאג על שאילתא, ולראות את כל הקריאות השונות, יש בטרמינל כפתור של Solutions ושם למטה לוחצים על ה-Debug(trace) עד כאן החלק הפשוט.

הייחודיות שבפונקציה הזאת, היא היכולת לעשות עוד הרבה דברים שהם הרבה מעבר לשרשור של שתי רשימות, שכדאי להכיר. נביא את רשימת הדוגמאות ונראה מה זה נותן לנו-

?- `append([], [4,5,6], Z).`

`Z = [4,5,6] ;`

no

אם נשרשר רשימה ריקה לאחרת, פשוט נעתיק את הרשימה לתוך Z.

?- `append([1,2,3], [4,5,6], Z).`

`Z = [1,2,3,4,5,6] ;`

No

זה ההרצה הרגילה שמשרשרת שתי רשימות.

?- `append(2, [4,5,6], Z).`

no

דבר שאמרנו כבר קודם – לא ניתן לשרשר מספר לתוך רשימה. לכן, כשהפרולוג בודק היתכנות של השאילתא, הוא בודק את השורה הראשונה, רואה שזה לא מתאים (כי 2 הוא לא רשימה ריקה), וממשיך הלאה – בודק בשורה השניה, אבל מאחר ו-2 אינו רשימה הוא לא יכול להפריד לראש וזנב, ולכן הוא פשוט מחזיר שכלום לא מתקיים.

?- `append(X, [4,5,6], [1,2,3,4,5,6]).`

`X = [1,2,3] ;`

no

?- `append([1,2,3], Y, [1,2,3,4,5,6]).`

`Y = [4,5,6] ;`

no

שתי השאילתות האלה, מבצעות בדיוק את אותו הדבר – חותכות מ-Z את החלק שלחנו ב-Y/X. איך זה מתבצע? נסתכל על ה-trace של השאילתא השניה:

Call: `append([1, 2, 3], _3488, [1, 2, 3, 4, 5, 6])`

Call: `append([2, 3], _3488, [2, 3, 4, 5, 6])`

Call: `append([3], -3488, [3, 4, 5, 6])`

Call: `append([], -3488, [4, 5, 6])`

Exit: `append([], [4, 5, 6], [4, 5, 6])`

Exit: `append([3], [4, 5, 6], [3, 4, 5, 6])`

Exit: `append([2, 3], [4, 5, 6], [2, 3, 4, 5, 6])`

Exit: `append([1, 2, 3], [4, 5, 6], [1, 2, 3, 4, 5, 6])`

Y = [4, 5, 6]

מה שמתבצע הוא בעצם חיתוך של ה-X וה-Z במקביל עד שמגיעים לתנאי העצירה (מאחר ומבחינת חוקי התוכנית, הוא רואה בכל פעם שה-H של X וה-H של Z שווים, אז הוא מוריד את שניהם, בעצם הפוך לביצוע המקורי), שם אנחנו מעתיקים את כל מה שנותר לתוך Y וחוזרים בחזרה למעלה. מה שמוחזר בסוף, הוא המשתנה שקיבל את ההשמה.

?- `append(X,Y,[1,2,3,4,5,6])`.

X = '[]'

Y = [1,2,3,4,5,6] ;

X = [1]

Y = [2,3,4,5,6] ;

X = [1,2]

Y = [3,4,5,6] ;

X = [1,2,3]

Y = [4,5,6] ;

X = [1,2,3,4]

Y = [5,6] ;

X = [1,2,3,4,5]

Y = [6] ;

X = [1,2,3,4,5,6]

Y = '[]' ;

no

אם נשרשר רק רשימות ריקות, הפונקציה תגרום לנו להביא את כל הפרמוטציות האפשריות לחלוקת Z לתוך X,Y.

פונקציות ופרדיקטים מובנים על רשימות

רשימת הפונקציות הבאה, מופיע בחוברת של פרופ' אנדריס:

append/3

שרשור שתי רשימות.

last/2

פרדיקט זה מחזיר ערך אמת (true), 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

שליטה בזרימת התוכנית

דיברנו קודם על סדר הוכחת הטענות בפרולוג, בקצרה – אנחנו עוברים ומחפשים בצד השמאלי (Head) תבנית שמתאימה לצורה שאנחנו שולחים (שם, מספר משתנים, סוג טיפוסים (אטום/רשימה) וכו'). וברגע שמוצאים תבנית מתאימה, אנחנו בודקים את התנאים מימין, ואם הכל מתקיים אנחנו מחזירים אמת (ומציבים מה שצריך משמאל). בחלק הזה, אנחנו נלמד על הדרכים השונות שנוכל ליצור מניפולציות על הזרימות השונות, ועל הדרכים להשתמש בזרימה לטובתנו.

Back-Tracking

כחלק מהמנגנון האוטומטי של פרולוג, אנחנו עלולים להכנס לתתי-שאלות. ניתן להסתכל על זה בצורה של עצים – כל כניסה לתת שאלה פותחת תת-עץ חדש, כך שברגע שנצטרך לחזור אחורה, נקפוץ לתת עץ אחר. למשל, נסתכל על התכנית הבאה:

buy (X, Y) :- car(Y), price(Y, Z), Z < 1000.

car(alpha).

car(subaru).

price(alpha, 1100).

price(subaru, 990).

התכנית הזאת בעצם בודקת לנו האם במסד הנתונים יש רכב שעולה פחות מ-1000 ש"ח (אנחנו לא מתייחסים למשתנה X, ואין באמת סיבה שהוא יהיה שם).

?- buy (ronen, Car).

Car = Subaru.

כאשר הפרולוג מבצע את השאילתא הנתונה, הוא רואה שהחלק השמאלי מתאים מבחינת התוכנית, ואכן הוא כך. אחרי זה הוא בודק את car(Y) להתאמה, ראשית הוא נכנס לאפשרות הראשונה שהוא רואה, "alpha", ולפני שהוא בודק אם שאר התנאים מתקיימים, הוא מסמן לעצמו את נקודת ההסתעפות, ובודק את התנאים הבאים, אבל כשהוא ממשיך לתנאי price(Y,Z), הוא נכשל שם ויוצא מהתוכנית, מאחר שהמחיר יותר גבוה מ-1000. אבל הוא לא יוצא לגמרי מהתוכנית אלא עף החוצה עד שהוא נתקל בנקודת ההסתעפות הקרובה אותה הוא סימן לעצמו כ-back-Tracking, וממשיך משם לבדוק אם יש רכב אחר שמסוגל לקיים את התנאי.

אנחנו יכולים לראות גם לפי התשובה, שהפרולוג עבר על כל האפשרויות הקיימות להסתעפות – כיצד? אם כבר באפשרות הראשונה, התוכנית היתה מצליחה למצוא נתון שיקיים את השאילתא, היתה לנו אופציה להמשיך ולבדוק את שאר האפשרויות, והוא היה ממשיך עד שהוא מגיע לסוף, שם אם לא היה מתקבל הוא היה מחזיר false.

אך מאחר וכאן יש לנו רק את התשובה בלי הצעה להמשיך לחפש, המשמעות היא שהגענו לאפשרות האחרונה, והיא זו שהתקבלה.

אבל יש לנו בעיות עם ה-Back-Tracking, אנחנו לא תמיד רוצים שהוא ימשיך הלאה ויחפש. איך אנחנו מתמודדים עם זה? קודם כל, נבין לגמרי את הבעיה –

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(Tail, Result).`

המשמעות ההצהרתית של הפרדיקט הזה היא כדלקמן – הסרת האיברים הכפולים מרשימה ריקה היא כמובן החזרה של רשימה ריקה. 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 - שני 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 רוצים 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 ל `remove_duplicates/2`) יבקש משמעויות נוספות, לא נקבל שום אלטרנטיבות שיביאו לנו תוצאות שייחשבו כטעויות.

הקטע הקודם, כמובן, מהחברת של פרופ' אנדרים.

נראה כעת מספר פתרונות לשליטה בתוכנית –

כפיית נסיגה - Fail

קודם כל, נתחיל מהצד ההפוך – נניח ואנחנו רוצים לקבל את כל התשובות האפשריות, אבל אנחנו רוצים לעשות את זה בצורה שוטפת, כלומר לא ללחוץ כל פעם על next, אלא שהתכנית תוציא לנו במכה אחת את כל התשובות הנכונות. בניגוד להיגיון האינטואיטיבי, הדרך לעשות את זה, הוא דווקא להכשיל את השאלתא – נראה דוגמא:

job: - father(X, Y), write(X), write(' father of '), write(Y), nl, fail.

father(aa, bb).

father(bb, cc).

father(cc, dd).

לתכנית כאן, יש רק תנאי אחד בודד, למצוא את הזוגות של אב ובן. מה שהתכנית עושה אחר כך, זה מדפיסה כל צמד של אב ובן אליו היא מגיעה, ואז יורדת שורה (nl = New Line). לאחר מכן, מופיע לנו הפרדיקט fail, שפשוט מכשיל באופן אוטומטי את התכנית. מה שזה יוצר לנו הוא קפיצה אוטומטית לנקודת ההסתעפות – Back-Tracking – האחרונה, כלומר לבדיקה של זוגות אב ובן. וכל התהליך חוזר על עצמו עד שריצת התכנית מסתיימת.

כדאי לשים לב – שיש משהו מיוחד בתכנית הזאת – אם היינו עושים בדיוק את אותו דבר, אבל בלי פקודות הדפסה (write), כלום לא היה קורה. כלומר – אם היינו מוצאים זוג וישר מפילים אותו, היינו ממשיכים עד סוף התכנית ולא היינו מחזירים כלום. כך שאולי זה יעיל לפעמים, אבל בחלק גדול מהמקרים זה לא מספיק טוב.

CUT - !

הפרדיקט הקודם שראינו (fail) עושה נסיגה כפויה, לעומת זאת, ה-CUT שנלמד עכשיו, מבטל נסיגות.

ברגע שאנחנו כותבים ! באחד מהרצפים השונים של התוכנית, אנחנו בונים "חומה". אם כל התנאים שלפני הקאט יתקיימו, אזי ברגע שנעבור אותו, אנחנו גורמים לתוכנית להיות חסומה לנסיגה החל מאותו רגע. גם אם יש אפשרויות נוספות של תתי-המטרות (Sub-Goals) שעדיין לא נוסו, התכנית לא תנסה לבצע אותם. יותר מזה – כל head שיהיה דומה לחלק שעברנו כבר בעזרת הקאט לא יבוצע יותר בהמשך (מבחינת התכנות הרגיל שאנחנו מכירים, יש לנו בעצם מעין אפשרות לעשות כאן תנאים של if, else). נסביר את דרך העבודה בצורה קצת מופשטת, ואחרי זה בצורה פשוטה יותר –

a: - b, c, d.

b: -

b: -

c: - c1, c2, !, c3, c4 .

c: - c5.

c1: -

c1: -

c2: -

c2: -

c3: -

c3: -

c4: -

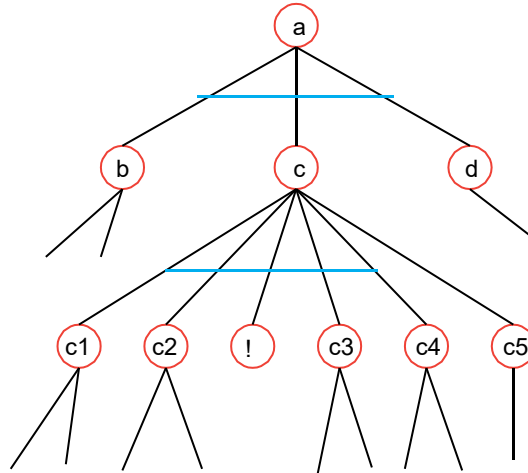
c4: -

c5: -

d: -

?a.

לצורך הדוגמה, נחשוב על כך שכל אות מבטאת סוג של חוק שעלינו לבדוק. רוב החוקים והעובדות פה בנויים בצורה פשוטה, מלבד c שמכיל בתוכו קאט. בשביל להבין את ריצה התכנית בדרך הפשוטה ביותר, נצטרך לדמיין את עץ הקריאות הבא –



כאשר אנחנו שואלים "האם a מתקיים?" אנחנו צריכים לוודא ששלושת החוקים המרכיבים את a מתקיימים, ורק אז נוכל לענות על זה בחיוב. בהתחלה, נבדוק את b שם יש לנו רמה אחת, ומספר עובדות לבדוק, כך שזה די פשוט. גם d, אם נגיע אליו, בודק את התנאי הבודד שלו. אבל עבור c הסיפור מתחיל להסתבך –

ברמה של c יש לנו חמישה תתי-תנאים, כאשר לאחר שני תנאים מופיע לנו הקאט. מה הדבר הזה יוצר – נניח שהעובדה הראשונה של c1 מתקיימת, אנחנו בודקים עבודה את c2 ומצליחים לאמת את שני ה-sub-goals הנוכחיים, כך שעברנו את הקאט בהצלחה. ברגע הזה, אם נמצא שמתקיימים לנו כל ה-sub-goals האחרים שיקיימו לנו את c, ונוכל להחזיר ש-a אכן מתקיים (בהנחה ש-d עובר ללא פגע), הפרולוג לא יביא לנו לבדוק אפשרויות נוספות של c1-c2. אם יש עוד אפשרויות באחרים, נוכל לבדוק אותם, אבל לא מעבר לקאט.

לעומת זאת, אם ניכשל עבור אותה רמה, האם נחזיר fail? תלוי. צריך לשים לב שהחוק שאנחנו לא בודקים שוב את ה-head תקף רק לאותה רמה של שאילתות. כלומר – אם ל-b היו אפשרויות נוספות שלא ניסינו, אז אמנם c יחזיר false, אבל הקפיצה תהיה לנקודת ה-back-tracking ברמה מעל – b.

כמובן, שזה לא הכל הרמוניה, פרפרים וחדי קרן. מסביר זאת פרופ' אנדריס (שמעכשיו נקרא לו "הפרופסור", במלרע)

בעיות עם CUT

קאטים הינם יעילים במיוחד על מנת "להדריך" את האינטרפרטר לכיוון הפתרון, 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[Head | 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/3 יעבוד 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!

סוגי CUT

זה שאנחנו דוחפים עכשיו קאטים לכל מקום, זה נחמד וכיף, אבל אנחנו צריכים להבין באמת מה אנחנו עושים. בשביל להבין כמו שצריך, אנחנו כמובן "ניתן בהם סימנים". יש שתי הגדרות בין-לאומיות לקאטים לפי צבעים, ועוד שתי הגדרות אותם הגדיר פרופ' קרנר, ועלינו לדעת את המייחד של כל קאט, ולזהות בעצמנו את הסוגים השונים. נתחיל בשניים המוסכמים על כולם – ירוק ואדום.

1. ! ירוק – קאט ירוק, הוא קאט שמשפר את התוכנית מבחינת זמן הריצה. ניתן להוריד אותו מהתוכנית ולהמשיך להשתמש בה, והפונקציונליות שלה תהיה בדיוק אותו דבר. מלבד שהיא תיתן לנו להמשיך ולרוץ לשווא.

2. ! אדום – קאט אדום, הוא קאט **נכון** שמשפר את התוכנית, ברמה כזאת שאסור לנו להוריד אותו. אם נוריד את הקאט הזה מהתוכנית, אנחנו נהרוס את התוכנית! חשוב לשים לב, כי זה מאוד מבלבל – זה שהקאט אדום זה **לא** כי הוא לא טוב, אלא כי הוא כל כך טוב, **שאסור לנו להוריד אותו**.

נראה בהמשך דוגמאות לכלל הקאטים השונים. ונעבור כעת לקאטים שהגדיר פרופ' קרנר. קאטים אלה הם ברמה יותר הבנתית של זרימת התכנית –

3. ! מיותר – צעד אחר מעבר לקאט הירוק. לפעמים מרוב התלהבות, אנחנו עלולים לדחוף קאטים בכל מקום, אבל לפעמים זה קצת מיותר. אם נשים קאט בסוף החוק האחרון האפשרי, אנחנו לא נועיל לעצמנו, כי גם ככה התכנית תעצור שם. אם נשים שם משהו הדבר היחיד שנעשה הוא להוריד לנו 2 נק' במבחן על קאט מיותר.

4. ! מזיק – דילוג מיותר מעבר למיותרים. לא רק שאנחנו דוחפים קאט במקום שהוא לא נצרך, אלא גם במקומות שאסור לנו. הדוגמה שהביא "הפרופסור", מדגימה לנו כיצד קאט לא נכון עלול לפגוע בתכנית ובתשובות שהיא מביאה. דבר זה כמובן יוריד לנו יותר מ-2 נקודות במבחן...

נראה עכשיו דוגמאות לכלל הקאטים באותה פונקציה, ונבין כיצד לעזור, וכיצד להמנע משגיאות מיותרות –

$\min(X, Y, X) :- X \leq Y.$

$\min(X, Y, Y) :- Y < X.$

הדוגמה הנ"ל היא התכנית הפשוטה ביותר וללא קאטים של מציאת המינימום. אם נכניס לכאן שאילתא בצורה הבאה-

?- $\min(2,5,Z).$

אז כמובן שנמצא לזה אימות כבר בשורה הראשונה. אבל מה שיחזור לנו יהיה כזה –

$Z = 2$

false.

שזה בגדול סבבה, אבל הבדיקה השניה היתה מיותרת. כי כבר מצאנו מה המינימום, ואין לנו צורך לבדוק שוב בשורה השניה שתחזיר לנו false. על מנת לשפר את זה נוסיף את הקאט הבא –

$\min(X, Y, X) :- X \leq Y, !.$

$\min(X, Y, Y) :- Y < X.$

הקאט הזה הוא כמובן, קאט ירוק. עבור אותה שאילתא שנכניס עכשיו אנחנו נקבל רק את ההצבה, ומאחר שעברנו בהצלחה, הפרולוג לא ירגיש צורך להמשיך ולחפש קיומים נוספים.

האם ניתן לשפר זאת יותר? כן. נראה את התכנית הבאה –

$\min(X, Y, X) :- X \leq Y, !.$

$\min(X, Y, Y).$

לקחנו את אותה תכנית ששיפרנו קצת, ומחקנו ממנה את התנאי של השורה השניה. אם השורה הראשונה עובדת, אז אין לנו צורך להמשיך ולבדוק. אבל אם השורה הראשונה לא מתקיימת, אז ברור שהשורה השנייה נכונה, ואפילו אין לנו מה לבדוק.

אבל, הקאט הזה הוא אדום. אנחנו שיפרנו את התכנית ברמה כזאת, שאם נוריד עכשיו את הקאט, אנחנו תמיד נקיים את השורה השניה, גם אם השורה הראשונה היא נכונה, אנחנו נחזיר כבר טעות.

עד כאן, ייעלנו את התכנית, עכשיו ננסה להרוס אותה –

$\min(X, Y, X) :- !, X \leq Y.$

$\min(X, Y, Y) :- Y < X$

נניח ושמנו את הקאט ישר בשורה הראשונה לפני כל תנאי. זה אומר שברגע שאנחנו עוברים אותו אנחנו לא נמשיך לבדוק תנאים דומים. כך שאם נכניס את השאילתה הבאה –

$\min(5, 2, Z).$

אנחנו ננסה את השורה הראשונה, ונראה שהיא לא מתקיימת. וכשננסה לחזור אחורה ולבדוק תנאים נוספים נראה שאנחנו לא מסוגלים לעשות את זה ונחזיר `false`, מה שכמובן לא נכון, והכנסנו פה קאט מזיק.

לעומת זאת, מה יקרה אם נכתוב את התכנית הבאה –

$\min(X, Y, X) :- X \leq Y, !.$

$\min(X, Y, Y) :- !.$

הקאט הראשון, הוא אדום, כפי שכבר אמרנו קודם. אך השני לא באמת עוזר לנו 0 אין לנו לאן להמשיך מכאן, ואם הגענו עד לכאן, התכנית בכל אופן תסתיים. אז חבל על 2 נקודות.

הערות נוספות על קאט

מאחר והכנסה של קאט שוברת ריצה לא רק עבור התנאי הנוכחי, אלא גם לכל ה-`heads` השונים, כדאי לשים לב שסדר השורות עלול להשפיע על התשובות המוחזרות. ניקח את הדוגמה האבסטרקטית הבאה:

$p :- a, b.$

$p :- c.$

כל עוד התכנית כתובה בצורה כזו, אין הבדל בין אם השורות יהיו כתובות כך או אחרת. כך או כך, אם אחת השורות לא תתקיים, נבדוק את השורה השנייה. אך אם נכניס קאט בין `a` ל-`b`. המצב עלול להיות שונה. נסתכל על שתי האפשרויות-

$p1 :- a, !, b.$

$p1 :- c.$

$p2 :- c.$

$p2 :- a, !, b.$

האם התוכניות שוות? לכאורה כן. רק שלא. אנחנו יכולים למצוא הצבה מסויימת בה נקבל תוצאות שונות. נניח ש $a=T, b=F, c=T$

עבור $p1$ מה שיקרה שניכנס בשורה הראשונה, ומאחר ו-`a` אמת, אנחנו נעבור מעבר לקאט, אך שם `b` שווה `F`. מה שיחזיר לנו `F` עבור כל השורה, מה שיכשיל את כל התכנית.

עבור $p2$, אם נשאל $p2$? (כלומר האם החוק הזה מתקיים), בשורה הראשונה, אנחנו נוכל להחזיר `c=T`, והתכנית תחזיר אמת. אבל אנחנו נוכל להמשיך, כי אין לנו קאט או סיבה אחרת שנעצור, ובשורה השניה, נקבל שלילה בגלל `b`. מה שיחזיר לנו אחר כך `F` לכל התכנית.

מכאן אנחנו צריכים להבין את החשיבות של סידור השורות על מנת לשלוט בתכנית בצורה נכונה.

רקורסיות

את השימוש ברקורסיות ראינו כבר בפרקים הקודמים בכל מיני ווריאציות, עכשיו נלמד כיצד לסווג את הרקורסיות השונות. אנחנו נדבר על רקורסיה זנבית, לא זנבית, ורקורסיות על רשימות.

רקורסיה לא-זנבית

נתחיל דווקא עם הלא-זנבית, מאחר והיא מה שהשתמשנו בה עד היום, גם אם לא ידענו שהיא מוגדרת כך. רקורסיות לא-זנביות, הן אלו שאנחנו מצמצמים את הערכים שקיבלנו, עד שמגיעים לרמה הנמוכה ביותר, ובדרך חזור לחוקים המקוריים אנחנו כל פעם מבצעים פעולה על הערך המוחזר. נראה דוגמה פשוטה לרקורסיה שכזאת –

`count ([H| T], RES):- count (T, RES1), RES is RES1+1.`

`count ([], 0).`

נראה דוגמת הרצה –

Call: `count([1, 4, 7], _3838)`

Call: `count([4, 7], _4110)`

Call: `count([7], _4110)`

Call: `count([], _4110)`

Exit: `count([], 0)`

Call: `_4114 is 0+1`

Exit: `1 is 0+1`

Exit: `count([7], 1)`

Call: `_4120 is 1+1`

Exit: `2 is 1+1`

Exit: `count([4, 7], 2)`

Call: `_3838 is 2+1`

Exit: `3 is 2+1`

Exit: `count([1, 4, 7], 3)`

Res = 3

התכנית הנ"ל אמורה להחזיר לנו את מספר הערכים ברשימה נתונה. בכל כניסה לרקורסיה, אנחנו חותכים את ראש הרשימה ומחזירים את השאר עד שאנחנו מגיעים לתנאי העצירה – רשימה ריקה. בשלב זה אנחנו מתחילים לחזור למעלה, כאשר כל יציאה מהרקורסיה מוסיפה לנו 1 לסכימת האיברים.

כאשר אנחנו כותבים רקורסיה שאינה זנבית, אנחנו מתכוונים לזה שהפיתרון אותו אנחנו מבקשים יהיה בראש התכנית, ולא בזנבה. בדוגמה שהבאנו RES יקבל את ההשמה הנכונה שלו, רק כשנחזור מספירת כל הרקורסיות השונות.

טיפ שכדאי לשים אליו לב – את תנאי העצירה שמנו בשורה השניה, מאחר ואנחנו פועלים פה על ספירה של רשימה לא-ריקה. כך שאם נקבל רשימה של 100 איברים, אין צורך לבדוק בכל פעם אם הרשימה כבר ריקה, כי ב-99 מתוך 100 פעמים התשובה תהיה "לא". במקרה כזה, עדיף לנו לחסוך את השאלות המיותרות האלה, ופשוט להכניס את האפשרות הזאת בסוף התכנית.

רקורסיה זנבית

בניגוד לרקורסיה לא-זנבית, שם אנחנו מקבלים את תוצאת הרקורסיה בראש המחסנית של הקריאות, ברקורסיה זנבית אנחנו נעשה את החישוב לעומק הקריאות, וכאשר נגיע לרמה הנמוכה ביותר – שם נקבל את התוצאה.

בדרך כלל, אנחנו פשוט נגדיר ארגומנט צובר חדש שיקבל ערך ראשוני, ובהמשך ההגדרות אנחנו נטפל בו באופן המתאים לדרישות.

על מנת להראות דוגמה, נסתכל על פונקציה של מניית איברים ברשימה, אותה משימה כמו שעשינו גם עבור הרקורסיה הלא-זנבית, ונעמוד על ההבדלים בין שניהם –

`count(List,Res):- count(List,o,Res).`

`count([],Res,Res).`

`count([H|T], Acc, Res):- Acc1 is Acc+1, count(T,Acc1,Res).`

בעוד הפונקציה הרל"ז (רקורסיה לא-זנבית) השתמשה באותו מספר ארגומנטים לכל אורך התכנית, בפונקציית ר"ז (רקורסיה זנבית), אנחנו בדרך כלל נגדיר ישר עבור מקרה הבסיס, פונקצייה דומה עם הצובר (כמו שכבר אמרנו מקודם). את הפונקציה אנחנו נמשיך ונשלח הלאה, עד שנקבל בתנאי העצירה את התוצאה הרצויה. על מנת לראות את ההבדל בקריאות, נראה את דוגמת ההרצה עבור אותה רשימה מהחלק הקודם –

Call: `count([1, 4, 7], -3950)`

Call: `count([1, 4, 7], 0, -3950)`

Call: `-4226 is 0+1`

Exit: `1 is 0+1`

Call: `count([4, 7], 1, -3950)`

Call: `-4232 is 1+1`

Exit: `2 is 1+1`

Call: `count([7], 2, -3950)`

Call: `-4238 is 2+1`

Exit: `3 is 2+1`

Call: `count([], 3, -3950)`

Exit: `count([], 3, 3)`

Exit: `count([7], 2, 3)`

Exit: `count([4, 7], 1, 3)`

Exit: `count([1, 4, 7], 0, 3)`

Exit: `count([1, 4, 7], 3)`

Res = 3

הדבר הראשון שאנחנו יכולים לראות בהרצה זאת, הוא שבניגוד להרצת הרל"ז, אנחנו לא עושים פה הקצאות חוזרות של ארגומנטים. אם קודם על כל כניסה לרקורסיה, היינו יכולים לראות הקצאת מקום (שנראית בתור קריאה של `_1234` או משהו בסגנון), כאן אנחנו משתמשים רק בארגומנט בודד איתו נכנסנו בפעם הראשונה. כל פעם אנחנו עושים חישוב-צבירה, ואז קוראים לפונקציה `count` מחדש. כאשר אנחנו מגיעים לתנאי העצירה, ומציבים שם את התוצאה הסופית, כל הכניסות פשוט מתפקקות להן החוצה בלי שום פעולה עד שיוצאים מהתוכנית.

בעוד שפונקציה רל"ז היא הרבה יותר חופשית באופן השימוש בה, ובאפשרות להטות אותה לשתי רקורסיות שונות, לפונקציות ר"ז יש קצת יותר הגבלות. בחלק מהמקומות מקפידים עליהם יותר ובחלק פחות, אך אנחנו ננסה לעמוד בהן כמה שאפשר עד הסוף. הדרישות עבור פונקציית ר"ז הן:

1. קריאה רקורסיבית יחידה – אין מצב בו אנחנו שולחים לשתי קריאות רקורסיביות, שיכולות לסבך את הקריאות השונות.
 2. הקריאה הרקורסיבית נמצאת ב-Sub-Goal האחרון של ה-Body, בהגדרה האחרונה בפרדיקט – כלומר, אין לנו פעולות נוספות לאחר היציאה מהרקורסיה.
 3. אין נקודת Back-Tracking משמאל לקריאה הרקורסיבית בחוק בו היא נמצאת – ברגע שאנחנו נצא מהרקורסיה עקב שלילה, אנחנו לא נמשיך ונבדוק אופציות נוספות.
- אם כתבנו את הרקורסיה נכון, בהתאם לשלושת התנאים הנ"ל, אז התוצאה תהיה בדיוק כמו שראינו קודם – חישוב, כניסה לרקורסיה, הגעה לסוף, החזרת תוצאה.
- השימוש הזה בפרולוג חוסך לנו קריאות למחסנית, ומוודא שהפעולות בצורה החלקה ביותר האפשרית מבחינת ריצת התוכנית. כדאי לשים לב, שאמנם אמרנו קודם שאת תנאי העצירה כדאי לשים בסוף, מאחר וזה חוסך לנו בדיקות מיותרות, אבל אם נעשה את זה כאן, אנחנו כבר נעבור על חוק הר"ז, ולכן אנחנו כבר שמים אותו במקום הרגיל.

יתרונות של ר"ז על פני רל"ז

ריצה מהירה יותר – כאשר אנחנו מסתכלים על העברת ערכי משתנים משכבה אחת לאחרת, אין צורך בהחזרת תוצאות ושחרור של זיכרון. אמנם יש לנו האטה מסוימת, בגלל הבדיקה של תנאי העצירה כל פעם, שבוודאי אנחנו נזקקים לזה רק עם אחת, אבל בסך הכול (בריצה של יותר משלושה מספרים...) הר"ז מהירה הרבה יותר.

ניצול זיכרון טוב יותר – כאשר אנחנו עובדים עם פונקציה שהיא ר"ז יש לנו בסך הכל שתי רמות זיכרון – המקורית, והרקורסיבית שממחזרת את עצמה, לעומת זאת, ברל"ז יש לנו $n+1$ רמות של מחסנית ביחס לכמות הקריאות הרקורסיביות. אמנם זה לא תמיד אסון, אבל יש הסתברות גבוהה יותר שרל"ז יעוף בגלל stack overflow מרוב קריאות לזיכרון והקצאת מקום מיותרת.

יתרונות של רל"ז על פני ר"ז

ניתן לשלוח יותר מרקורסיה אחת – הרבה פעמים כאשר הרקורסיה היא מותנית, אנחנו לא יכולים להסתפק בקריאה רקורסיבית אחת ויחידה, ועלינו להשתמש בתנאים שונים.

פחות מגבלות על הכתיבה – אמנם ר"ז קריאה יותר ונוחה, אבל כל ההגבלות שחלות על הפונקציות האלה מגבילות עלינו ומקשות את הכתיבה עצמה.

זמן ריצה – אנחנו יכולים ברל"ז לחסוך את כל הקריאות של תנאי העצירה על ידי שנכניס את השורה הזאת בסוף. כמה זה משמעותי? אם יש לנו n איטרציות, אז פונקציה רל"ז תעשה $n+2$ בדיקות (n בדיקות רגילות, תנאי עצירה ורשימה ריקה). ולעומת זאת, פונקצייה ר"ז תעשה $2n+1$ בדיקות (כי לפני כל בדיקה רגילה אנחנו נצטרך לבדוק גם את תנאי העצירה לשווא).

ככלל – אם אנחנו מדברים על ריצה שניתן לעשותה בעזרת לולאה אחת (while, for) אז כדאי להשתמש בר"ז, אחרת – עדיף להשתמש ברל"ז.

ר"מ – ריקורסיית רשימה מקוננת

מה יקרה כאשר נרצה לעבוד על רשימות מקוננות? אם יש איבר ב-Top-Level שהוא רשימה עם מספר איברים, אי אפשר להתייחס לזה בצורה פשוטה כמו רשימה "שטוחה". בדרך כלל אנחנו מתייחסים לכל האיברים בתור ראש וזנב ומשחקים עליהם, תחת ההנחה שכך או כך, מדובר באיברים אטומיים ולא רשימות.

עכשיו ננסה להבין איך לעבוד על איבר שהוא בעצמו רשימה, או מכיל מספר רשימות מקוננות.

קודם כל, אנחנו בודקים על האיבר הראשון האם הוא רשימה או איבר בודד, אם אנחנו מצליחים (H הוא אכן רשימה) אנחנו ממשיכים הלאה, ובודקים כל תת רשימה בנפרד. אם לא, אנחנו מכניסים את שאר הרשימה ברקורסיה וממשיכים למנות.

נראה פונקציה של מניית איברים ברשימה מקוננת, ונראה כיצד היא פועלת:

$\text{cnt}([H | T], Z) :- \text{listp}(H) , !, \text{cnt}(H, Z_1) , \text{cnt}(T, Z_2) , Z \text{ is } Z_1+Z_2.$

$\text{cnt}([_ | T], Z) :- !, \text{cnt}(T, Z_1) , Z \text{ is } Z_1 + 1.$

$\text{cnt}([], 0).$

$\text{listp}([_ | _]):- !.$

$\text{listp}([]).$

בשורה הראשונה אנחנו מקבלים את האיבר הראשון ברשימה, ושולחים אותו לפרדיקט שבודק האם האיבר H הוא רשימה. איך זה מתבצע? הפרדיקט `listp` בודק אחת משתי אופציות –

1. יש רשימה עם יותר מאיבר אחד (שורה רביעית), כאשר אין זה חשוב לנו מה הערכים בפנים, אלא עצם העובדה אם יש שם ערכים או לא.

2. בשורה החמישית, נבדקת האפשרות שמדובר ברשימה ריקה, ואז אמנם אין שם משהו, אך אנחנו מתייחסים לזה כרשימה.

לאחר שהצלחנו לאמת שאכן H הוא רשימה, אנחנו שולחים אותו בצורה רקורסיבית לתוך הפונקציה שוב, על מנת לבדוק כמה איברים יש בו, ואז שולחים את זנב הרשימה בעצמה כדי לבדוק כמה איברים יש גם בה, ולבסוף מחברים ומחזירים את התוצאה.

מה קורה אם H אינו רשימה, אלא איבר אטומי? אנחנו שולחים את זנב הרשימה למניה, ולתוצאה המוחזרת אנחנו מוסיפים 1 כנגד ה-H.

הערות לגבי הקאטים: ה-! הראשון הוא אדום (מועיל והכרחי) כי אם יש לנו רשימה ב-H, אין לנו סיבה להמשיך ולבדוק תנאים נוספים, כי אז זה עלול פגום לנו בספירה (אם נגיע לתוצאה הנכונה, ואז ננסה לבדוק ב-Back-Tracking איזה תנאי לא נכון).

ה-! השני, הוא ירוק, כי אם נכנסנו לשורה הזאת, אז בשורה השלישית בטח ניכשל, והקאט הזה רק מסייע לנו לחסוך ריצה מיותרת.

גם ה-! השלישי בשורה 4 הוא ירוק, כי הוא רק חוסך חנו כניסה לשורה החמישית, בה לא יקרה שום דבר מיוחד (כי אם הרשימה מכילה יותר מאיבר אחד, היא בוודאי לא ריקה, והתנאי כניסה לא יתקיים שם).

פרדיקטי מערכת ותכנות על

בחלק זה נתמקד יותר בפרדיקטים ופונקציות המוגדרות במערכת הפרולוג, לטובת שימושים שונים אותם נצטרך לפונקציות שאנחנו נבנה בעצמנו. לאחר מכן, נדבר על האפשרות ליצור אינטרפטר משלנו שיגיב אחרת ויענה על השאלות שאנחנו שואלים בצורות חישוב שונות, ונדבר על ההבדלים, היתרונות והחסרונות.

פרדיקטים להחזרת קבוצת תוצאות

עד עכשיו, דאגנו להחזיר תשובה בודדת לכל שאלה. אם חיפשנו ערך מסוים, או ארגומנט מתוך רשימה או חישוב כלשהו, החזרנו בסוף את תשובה הרצויה. אם רצינו, וזה היה אפשרי, היינו לוחצים על "next" ומקבלים תשובות נוספות. ראינו גם בפרק של "זרימה בשליטת התוכנית", את האפשרות להדפיס תוצאות ואז "להכשיל" את התוכנית שבביל להוציא את כל התשובות האפשריות. עכשיו, אנחנו נראה שלושה פרדיקטי מערכת המסייעים לנו במשימה הזאת, באופן מוגדר וברור ללא צורך להכשיל את השאילתא.

bagof/3

הפרדיקט bagof מקבל שלושה ארומגנטים – bagof(Type, Cond, List).

כאשר החישוב מתבצע על סוגים מסויימים (Type), העונים לתנאי מסויים (Cond), ומחזיר את כל התשובות האפשריות תוך הרשימה (List).

נראה דוגמת מסד נתונים, וכיצד הפונקציה תענה:

```
class(a,vow).
class(b,con).
class(d,con).
class(c,con).
class(e,vow).
class(f,con).
class(b,con).
```

?- bagof(Letter,class(Letter,con),Letters).

```
Letters=[b,d,c,f,b].
```

ברשימת העובדות הגדרנו סוגים של אותיות – עיצורים (con) ואותיות ניקוד (vow), ובשאילתא ביקשנו שמתוך הזוגות של class, שהארגומנט הימני שלו הוא con, יחזיר את כל ה Letter - המשתנים של האותיות, לתוך הרשימה Letters, ואכן, הוחזרה לנו הרשימה המתאימה.

תכנות bagof

- אם נשלח ל-bagof פרדיקט שלא יצלח, אנחנו נקבל בחזרה fail.
- אם בנתונים יהיה לנו מידע כפול, הוא יוחזר בכל פעם מחדש.
- סדר ההופעה בתוך הרשימה המתקבלת, הוא סדר ההופעה.

למעשה, bagof מביא לנו את רשימת התוצאות באופן הכי גולמי וראשוני האפשרי, יש מקרים מסוימים בהם זה יספיק לנו, אך אם לא, נוכל לעשות אחר כך מניפולציות על הרשימה המתקבלת, עד שנקבל את הדרוש.

setof/3

הפרדיקט setof מקבל בדיוק את אותם ארגומנטים כמו bagof, אך מביא שיפור ביחס לתוצאות המוחזרות- אם נסתכל על מסד הנתונים הקודם, עם אותה שאילתא, אך מופנית לכיוון ה-setof, נקבל את הדבר הבא:

?- setof(Letter,class(Letter,con),Letters).

Letters=[b,c,d,f].

תכונות setof

- הרשימה החוזרת היא ללא כפילויות (כמו קבוצה מתמטית).
- הערכים המוחזרים בתוך הרשימה, מופיעים ממוינים בסדר עולה.

אם הגדרנו שני סוגי Type, אז הערך המוחזר יהיה רק של הערכים המקיימים את שני התנאים השונים. לעומת זאת, אם הגדרנו שהערך המוחזר יורכב ממספר ארגומנטים, אז יחזרו לנו צמדים של שני הארגומנטים. כאשר, המיון יהיה על פי הארגומנט הראשון, ואם יש כפילות, אז מכניסים אותם אחד אחרי השני, בצורה ממוינת פנימית. נראה וריאציה של מסד הנתונים הקודם כדי לראות את ההבדל:

class(a,vow,one).

class(b,con,one).

class(d,con,two).

class(c,con,one).

class(e,vow,one).

class(f,con,one).

class(b,con,two).

?- setof((Letter,Num),class(Letter,con,Num),Letters).

Letters = [(b,one), (b,two), (c,one), (d,two), (f,one)]

אם נסתכל למשל על ההחזרה של הרשימה הזאת, נוכל לשים לב שקודם כל המיון מתבצע על האותיות לפי הסדר, ואיפה שיש כפילויות (למשל b), מופיעות שתי האפשרויות ממוינות בצורה פנימית. כמובן, שיש חשיבות לסדר ההופעה של הערכים בתוך השאילתא עצמה – אם היינו משנים את הסדר בשאילתא, היינו מקבלים מיון אחר לגמרי –

setof((Num,Letter),class(Letter,con,Num),Letters).

Letters = [(one,b), (one,c), (one,f), (two,b), (two,d)]

מאחר והגדרנו ראשון את ה-Num אז קודם כל זה מיון לפי הערכים האלה, ואז בתוך כל סט, עשינו מיון נוסף.

כמובן, שאת הדבר הזה, היינו יכולים לעשות גם ב-setof, אבל כך או כך, היינו מקבלים את הערכים באותו סדר הופעה, אך עם סדר פנימי שונה –

setof((Letter,Num),class(Letter,con,Num),Letters).

Letters = [(b,one), (b,two), (c,one), (d,two), (f,one)]

setof((Num,Letter),class(Letter,con,Num),Letters).

Letters = [(one,b), (one,c), (one,f), (two,b), (two,d)]

findall/3

גם הפרדיקט הזה מקבל את שלושת הארגומנטים הידועים, ומחזיר אותם בצורה דומה לזו של bagof, אך עם שוני אחד גדול – אמנם הערכים חוזרים בסדר ההופעה, אך ברשימה יופיעו **כל** הערכים המתאימים לסוג ברשימה אחת. על מנת להבין את ההבדל, נראה את אותה שאילתא בדיוק, אך עבור שני הפרדיקטים האלה, ונראה את התוצאה –

?- bagof(Letter,class(Letter,Class),Letters).

Class = con,
Letters = [b, d, c, f, b]

Class = vow,
Letters = [a, e]

אם נבקש ב-bagof את כל הצמדים של אותיות וסוג, נקבל בהתחלה את הרשימה של המחלקה con, כאשר קודם כל נקבל את ההשמה של class, ואחר כך את הרשימה של האותיות המתאימות המוחזרות ב-Letters. לאחר שנבקש תוצאות נוספות, נקבל את ההשמה השניה האפשרית, ואת הערכים המתאימים אליה.

לעומת זאת, עבור findall, מה שנקבל הוא כזה -

findall(Letter,class(Letter,Class),Letters).

Letters = [a, b, d, c, e, f, b]

כל הרשימה של הערכים המקיימים השמה מסויימת של Class ייכנסו כאחד בתוך Letters ויחזרו אלינו. כמובן, שאך אחד משתי האפשרויות האלה לא "מיותרות", ונשתמש בהם באופן שונה עבור שאילתות שונות.

תכונות findall

- מחזיר רשימה מלאה של כל האפשרויות הניתנות להשמה בתנאי.
- אם לא נמצא שום השמה, אנחנו לא מקבלים false, אלא רשימה ריקה [].

שימוש בתנאי מורכב

עד עכשיו, ראינו בכל הפרדיקטים, תנאי בודד של משהו שרק יקיים השמה מסויימת. נראה עכשיו את האפשרות לשימוש בתנאי מורכב. למעשה, אנחנו בונים לנו מעין "פרדיקט עזר" שיכול להיות מורכב מתנאי מורכב או כמה תנאים כאלה, ואז כשנקרא לאחת מהפונקציות שראינו, אנחנו יכולים פשוט להשתמש בשם התנאי שיצרנו. נראה דוגמא שתסביר את העניין. ניקח מסד נתונים חדש –

student(ann,44711, pass).

student(bob,50815, pass).

student(pat,41018, fail).

student(sue,41704, pass).

לצורך העניין – רשימת תלמידים ות"ז שלהם, שעברו מבחן מסוים. אנחנו עכשיו רוצים לבקש תנאי מאוד מסוים – קודם כל, אנחנו מחפשים את השלשה הזאת בסדר הנכון, ושמספר התלמיד יהיה נמוך מ-50,000. כמו כן, אנחנו כמובן רוצים לקבוע את פורמט ההחזרה באופן מסוים של שם תלמיד/(עובר/נכשל). בשביל זה, נשלח את השאילתא הבאה –

?- Goal = (student(Name,Num, Grade), Num<50000),

findall(Name/Num, Goal, List).

שימו לב – קודם כל הגדרנו **משתנה** בשם Goal (אפשר לראות שהוא משתנה כי הוא מתחיל באות גדולה), ובתוכו הכנסנו בדיוק את התנאים הרצויים לנו, ולאחר מכן כאשר קראנו לפרדיקט findall, התנאי הנשלח היה המשתנה Goal, בלי ציון נוסף של תנאים. מה שיוחזר לנו עבור השאילתא הזאת, יהיה הדבר הבא –

Goal = (student(Name,Num, Grade), Num<50000),
 List = [ann/pass, pat/fail, sue/pass]
 Yes

אפשר לראות ש-bob לא הופיע כאן, מאחר והמספר שלו הוא 50815, שזה בוודאי גדול מ-50000.

השמטת ערכים

ניתן להשתמש בפרדיקט bagof, באופן שלא מתחשב בכל הערכים שיש לכל איבר. כלומר – אם ניקח את רשימת הסטודנטים שעבדנו עליה עכשיו, ואנחנו רוצים להחזיר רשימות נפרדות של אנשים שעברו/נכשלו, ללא ציון של המספר שלהם. איך נעשה את זה?

מאחר ויש לנו מספר שונה לכל תלמיד, אנחנו חייבים להתייחס אליו כמשתנה. אך גם אם ננסה לבקש רק את שם התלמיד, מה שנקבל יהיה הבלאגן הבא –

?- bagof(Name,student(Name,Num,Grade),List).

Grade = fail,
 List = [pat],
 Num = 41018

Grade = pass,
 List = [sue],
 Num = 41704

Grade = pass,
 List = [ann],
 Num = 44711

Grade = pass,
 List = [bob],
 Num = 50815

ערימות של כל העובדות המקיימות את המשתנים ללא מיון וסדר. כמובן, שזה לא מספיק לנו. אם ננסה להתחכם, ולומר "סבבה. לא אכפת לנו מה יש ב-Num", ואז פשוט נשים שם " _", גם זה לא יפתור לנו את הבעיה –

?- bagof(Name,student(Name,_,Grade),List).

Grade = fail,
 List = [pat]

Grade = pass,
 List = [sue]

Grade = pass,
 List = [ann]

Grade = pass,
 List = [bob]

אמנם קיבלנו את הכל בלי מספר התלמיד, אבל כמובן שזה לא מספיק טוב. כל היתרון ב-bagof הוא להחזיר את תתי הרשימות השונות. לצורך זה, ייצרו לנו בפרולוג את האפשרות לבקש התעלמות ממשתנה מסוים. זה מתבצע על ידי שרשור של הערך המיותר לתנאי –

?- bagof(Name,Num^student(Name,Num,Grade),List).

Grade = fail,
 List = [pat]

```
Grade = pass,  
List = [ann, bob, sue]
```

עכשיו קיבלנו את המיון לפי הציון עובר/נכשל, ובכל אחד מהם יש לנו את הרשימה המתאימה שלו.

נעבור עכשיו לרשימה של פרדיקטים מוגדרים במערכת, שאנחנו יכולים להשתמש בהם באופן חופשי. המטרה הסופית שלנו, כאמור, תהיה האפשרות לבנות אינטרפטר משל עמנו. לצורך כך אנחנו נציג פרדיקטים שחוקרים לנו ביטויים שונים ויכולים לתת לנו את ה-metadata של כל דבר –

פרדיקטים לבדיקת טיפוס

`var(V)`

בדיקה האם המשתנה X הוא חופשי, כלומר ללא השמת ערכים.

`nonvar(NV)`

מחזיר "אמת" אם המשתנה אינו חופשי כרגע.

`atom(A)`

בדיקה האם המשתנה A מקושר לאטום. אטום אינו מספר, אלא ערך סופי כלשהו. למשל 'Taco', blue או אפילו רשימה ריקה.

`integer(I)`

בדיקה האם I הוא מספר שלם.

`number(N)`

בדיקה האם N הוא מספר כלשהו – שלם או שבר

`atomic(A)`

בדיקה האם המשתנה A הוא אטום (atom) או מספר (number).

בסגנון הפקודות האלה, ניתן גם למנות את הבדיקות האם מדובר במספר רציונלי (`float(F)`), האם משתנה מסוים הוא רשימה (`is_list(L)`) ועוד. אם אנחנו נתקלים בבדיקה שלסוג טיפוס מסויים, כדאי לבדוק, כי יכול להיות שזה כבר מיושם במערכת.

פרדיקטים ללוגיקת על

הפרדיקטים הבאים מתייחסים ללוגיקה בשימוש של פרדיקטים כאלה ואחרים. כלומר – אם עד עכשיו העברנו ארגומנטים לתוך פרדיקט, עכשיו אנחנו מסוגלים להעביר פרדיקטים שלמים ולבדוק דברים מסויימים.

`call(X)`

בתוך X אנחנו שולחים קריאה לפרדיקט אחר, עם ערכים שונים, ואנחנו יכולים לבדוק אם הפרדיקט הנשלח X יסתיים בהצלחה או לא. אנחנו לא מתעסקים בערך המוחזר מתוך X אלא רק בשאלה עבר/נכשל, במידה שהצלחנו, call יחזיר לנו "אמת".

`clause(A,B)`

הפרדיקט הזה עובר על כל הנתונים שלנו, וברגע הראשון שהוא מזהה איזה ראש של חוק/עובדה שדומה ל-A הוא מכניס לתוך B את כל הגוף מימין. נראה דוגמא שתסביר את העניין –

parent(avi, ronem).

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

נגידר באופן פשוט מספר יחסים של אב ובן, ונראה מה יביא לנו השימוש בפונקציה הזאת –

?clause(parent(avi, ronem), Body).

Body = true

אם נבקש שיחפש לנו ראש שמתאים לפורמט שביקשנו, של "אבי אבא של רונן", יוחזר לנו ב-Body, החלק מימין. כזכור, הגדרנו בתחילת הקורס, שאם יש לנו עובדה, אז למעשה היא שקולה לכתיבת חוק אמת, ולכן ב-B ייכנס לנו true. עכשיו נראה משהו מורכב יותר –

?clause(father(X,Y), B).

B = (male(X), parent(X,Y))

אנחנו הגדרנו חוק שמגדיר את היחס "אבא של" על ידי בדיקה של זכר ושהוא מוגדר כ"הורה". ולכן ב-B כל זה יחזור לנו לשימוש עתידי.

functor(Term, Functor, Arity)

כאשר נכניס ביטוי כלשהו לתוך Term, אנחנו נקבל בחזרה את ה-Functor, כלומר השם/הכרזה של הביטוי, וב-Arity כמה ערכים צריך להכניס לתוך הביטוי, על מנת שיתקבל בצורה נכונה. למשל –

?- functor(parent(avi, ronem), F, A).

A = 2,

F = parent

אם נכניס את הביטוי לקבלת הורה, אנחנו נקבל את שם הביטוי "parent", ו-2, כמספר הערכים המגדירים אותו.

arg(N, Term, Argument)

אם נרצה לקבל בחזרה את אחד הערכים המרכיבים ביטוי, נוכל להכניס את הביטוי Term, ואת מספר הארגומנט הדרוש N, שיחזור לתוך Argument. למשל –

?- arg(2, parent(avi, ronem), Arg).

Arg = ronem

כאן אנחנו מבקשים להחזיר את הארגומנט השני מתוך הביטוי המוכנס, וחוזר לנו רונן. כדאי לשים לב, הפרדיקט arg איננו בודק במסד הנתונים עבור חוקים מתאימים, אלא מתייחס רק לביטוי שנכנס אליו, כך שאם נכניס (X,Y) הוא יחזיר לנו רק Y. ואם ניתן לו "_" הוא יחזיר לנו true.

פרדיקטים אקסטרנה-לוגיים

אוסף של פרדיקטי מערכת, הנותנים לנו שימוש בשימוש בתוכנית –

system(X)

מחזיר אמת אם X הוא פרדיקט מערכת, ושקר אם X הוא פרדיקט מוגדר על ידי משתמש.

פרדיקטי קלט/פלט

read(X)

קריאה מהטרמינל והכנסה ל-X.

write(X)

כתיבה לטרמינל, כאשר ה-X יכול להיות משתנה עם ערך כלשהו, ואפשר במקומו להכניס פשוט מחרוזת.

nl, tab

ירידה לשורה חדשה או הכנסה לטאב.

פרדיקטי קלט/פלט מקובץ

[+File], consult(File)

בארגומנט File אנחנו יכולים להכניס נתיב לקובץ, ואז כל הפסקויות הקיימות בו ייטענו לתוך מסד הנתונים. שני הפרדיקטים הרשומים למעלה הם מקבילים בפעולה שלהם.

[File], reconsult(File)

אם אנחנו רוצים להתייחס אך ורק לפסקויות בתוך הקובץ, ולדרוס את כל מסד הנתונים הקיים, אנחנו יכולים להוריד את ה+ או לעשות "התייעצות מחדש".

ניהול זיכרון דינאמי

ישנם מספר פרדיקטים שאנחנו כבר מכירים אותם כמו ! fail, שנכנסים כולם תחת הקטגוריה הזאת. מעבר להם, יש לנו מספר פקודות שיכולות להוסיף לנו מידע תוך כדי ריצת התוכנית

assert(X)

הוספה של פסקוית בסוף התוכנית.

asserta(X)

הוספה של פסקוית בראש כל הפסקויות. הפקודה הרגילה של assert, או assertz המקבילה לה, מוסיפה את הפסקוית בסוף רשימת הפסקויות הקיימות, מה שיותר נכון לעשות בזמן הריצה, אך אם משום מה אנחנו רוצה להוסיף אותו לראש העובדות (מה שישפיע על כך שזה ייבדק ראשון בכל פעם), נשתמש ב-asserta.

retract(X)

מחיקת הביטוי X מהנתונים בזמן ריצה.

retractall(X)

מחיקת כל הביטויים הדומים ל-X. נראה דוגמאות כדי להבין את השימוש במחיקה הזאת –

?- retractall(parent(avi, _)).

דבר זה ימחק לנו את כל מי שאבי הוא הורה שלו.

?- retractall(parent(_, _)).

המחיקה הזאת תשפיע על כל החוקים מסוג parent ותמחוק אותם.

Abolish(F, A)

מחיקת כל הפסקויות בעלות פאנקטור F ומספר ערכים A. כלומר אם נרצה עכשיו למחוק את Parent, נוכל לעשות זאת באופן הבא –

?- abolish(parent,2).

יצירת אינטרפטר⁴

בעזרת כל הפרדיקטים שראינו עכשיו, אנחנו יכולים למעשה לבנות אינטרפטר משלנו. אנחנו יוצרים שלושה גרסאות של אינטרפטרים שייקראו solve העובדים באופנים שונים. חלקם מהירים יותר וחלקם פחות.

כאשר אנחנו יוצרים אינטרפטר ורוצים להשתמש בו, אנחנו מקיפים את כל הפקודה שאנחנו שולחים תחת סוגריים. למעשה, מי שהשתמש עד עכשיו בדיבאגר, השתמש בסוג מסוים של אינטרפטר.

קודם כל נגדיר את הדבר הבסיסי ביותר –

`solve(true):- !.`

`solve((A,B):- !, solve(A), solve(B)).`

`solve(A):- clause(A,C), solve(C).`

מה שיצרנו פה, הוא האפשרות לשלוח ריבוי ארגומנטים לתוך האינטרפטר – אם יש שני ארגומנטים, אנחנו פשוט שולחים כל אחד מהם בנפרד, ומפרקים כמה שאפשר עד שאנחנו מגיעים ל-body הבסיסי ביותר, ואנחנו בודקים אם הוא מתקיים, וזה תנאי העצירה שלנו.

אם ניקח את מסד הנתונים הבא –

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

`parent(avr, yit).`

`male(avr).`

נראה מה יקרה כאשר נריץ אותו באינטרפטר שבנינו –

?- `solve(father(avr, yit)).`

Call: `solve(father(avr, yit))`

Call: `clause(father(avr, yit), -4344)`

Exit: `clause(father(avr, yit), (parent(avr, yit), male(avr)))`

Call: `solve((parent(avr, yit), male(avr)))`

Call: `solve(parent(avr, yit))`

Call: `clause(parent(avr, yit), -4372)`

Exit: `clause(parent(avr, yit), true)`

Call: `solve(true)`

Exit: `solve(true)`

Exit: `solve(parent(avr, yit))`

Call: `solve(male(avr))`

Call: `clause(male(avr), -4378)`

Exit: `clause(male(avr), true)`

Call: `solve(true)`

Exit: `solve(true)`

Exit: `solve(male(avr))`

Exit: `solve((parent(avr, yit), male(avr)))`

Exit: `solve(father(avr, yit))`

⁴ לא רלוונטי למבחן

true

אנחנו שולחים את הביטוי שהכנסנו לתוך השורה השלישית, שם על מנת לבדוק את האפשרות של ההורה, אנחנו שולפים את החוק המתאים שבדק אבהות, ומציבים את הערכים שהכנסנו במקומם, ועכשיו אנחנו בודקים כל אחד בנפרד – קודם כל האם יש לנו קשר של "הורה" בין שני האנשים, ואחריה, האם ההורה הוא גם זכר. ברגע שהכל מתברר כאמת, אנחנו מחזירים true.

עכשיו נראה איך אנחנו יכולים לשכלל את solve, על מנת שיהיה יותר יעיל בזמן ריצה –

$\text{solve}_1((A_1, A_2), B) :- !, \text{solve}_1((A_1, (A_2, B)))$.

$\text{solve}_1(\text{true}, B) :- !, \text{solve}_1(B)$.

$\text{solve}_1((A, B) :- !, \text{clause}(A, C), \text{solve}_1((C, B)))$.

$\text{solve}_1(\text{true}) :- !$.

$\text{solve}_1(A) :- \text{clause}(A, C), \text{solve}_1(C)$.

היעילות כאן מתבטאת בכך אנחנו עובדים בצורה של חיפוש לעומק DFS. ברגע שאנחנו מוצאים תנאי שמתריים אנחנו רצים עליו עד הסוף על מנת להחזיר תוצאה. נוכל לראות בדוגמת הריצה כמה זה יותר מהר:

?- $\text{solve}_1(\text{father}(\text{avr}, \text{yjit}))$.

Call: $\text{solve}_1(\text{father}(\text{avr}, \text{yjit}))$

Call: $\text{clause}(\text{father}(\text{avr}, \text{yjit}), _4816)$

Exit: $\text{clause}(\text{father}(\text{avr}, \text{yjit}), (\text{parent}(\text{avr}, \text{yjit}), \text{male}(\text{avr})))$

Call: $\text{solve}_1((\text{parent}(\text{avr}, \text{yjit}), \text{male}(\text{avr})))$

Call: $\text{clause}(\text{parent}(\text{avr}, \text{yjit}), _4844)$

Exit: $\text{clause}(\text{parent}(\text{avr}, \text{yjit}), \text{true})$

Call: $\text{solve}_1((\text{true}, \text{male}(\text{avr})))$

Call: $\text{solve}_1(\text{male}(\text{avr}))$

Call: $\text{clause}(\text{male}(\text{avr}), _4856)$

Exit: $\text{clause}(\text{male}(\text{avr}), \text{true})$

Call: $\text{solve}_1(\text{true})$

Exit: $\text{solve}_1(\text{true})$

Exit: $\text{solve}_1(\text{male}(\text{avr}))$

Exit: $\text{solve}_1((\text{true}, \text{male}(\text{avr})))$

Exit: $\text{solve}_1((\text{parent}(\text{avr}, \text{yjit}), \text{male}(\text{avr})))$

Exit: $\text{solve}_1(\text{father}(\text{avr}, \text{yjit}))$

true

לעומת solve שם היו לנו סך הכל 19 קריאות, כאן כבר ירדנו ל-16. שאמנם זה לא נראה מהותי, אבל הבדיקה פה כמובן היתה קטנה מאוד באופן יחסי.

עכשיו נראה איך אנחנו יכולים לייעל את כל התכנית עוד יותר, רק בעזרת שינוי התוכנית solve מ-DFS ל-BFS:

$\text{solve}_2((A_1, A_2), B) :- !, \text{solve}_2((A_1, (A_2, B)))$.

$\text{solve}_2(\text{true}, B) :- !, \text{solve}_2(B)$.

$\text{solve}_2((A, B) :- !, \text{clause}(A, C), \text{solve}_2((B, C)))$.

$\text{solve}_2(\text{true}) :- !$.

$\text{solve}_2(A) :- \text{clause}(A, C), \text{solve}_2(C)$.

מי שיחפש את השינוי לעומת הפונקציה הקודמת, יוכל למצוא את זה בסוף השורה השלישית, שם C ו-B החליפו מקומות, דבר זה יוצר לנו חיפוש רוחב.

?- solve2(father(avr, yit)).

Call:solve2(father(avr, yit))

Call:clause(father(avr, yit), _4952)

Exit:clause(father(avr, yit), (parent(avr, yit),male(avr)))

Call:solve2((parent(avr, yit),male(avr)))

Call:clause(parent(avr, yit), _4980)

Exit:clause(parent(avr, yit), true)

Call:solve2((male(avr),true))

Call:clause(male(avr), _4992)

Exit:clause(male(avr), true)

Call:solve2((true,true))

Call:solve2(true)

Exit:solve2(true)

Exit:solve2((true,true))

Exit:solve2((male(avr),true))

Exit:solve2((parent(avr, yit),male(avr)))

Exit:solve2(father(avr, yit))

True

מבחינת קריאות, גם כאן יש לנו 16, אבל החיפוש לרוחב נחשב טיפה יותר טוב.