Λογικός Προγραμματισμός με Περιορισμούς 1

1. List Processing

```
1  exclude_range(_,_,[],[]).
2  exclude_range(Low,High,[X|Tail],Answer):-
3      X >= Low, X =< High,!,
4      exclude_range(Low,High,Tail,Answer).
5
6  exclude_range(Low,High,[X|Tail],[X|Answer]):-
7      exclude_range(Low,High,Tail,Answer).
```

- *Βασική περίπτωση:* Όποιο και να είναι το διάστημα τιμών, όταν η λίστα είναι κενή, η νέα λίστα είναι επίσης κενή.
- *Γενική περίπτωση:* Υπάρχουν δύο εναλλακτικές,
    1) Το πρώτο στοιχείο ανήκει στο διάστημα τιμών οπότε δεν προστίθεται στη νέα λίστα
    2) Το πρώτο στοιχείο δεν ανήκει στο διάστημα τιμών οπότε προστίθεται στη νέα λίστα.

Στη γενική περίπτωση υπάρχει αποκλειστική διάζευξη, δηλαδή η μία εναλλακτική αποκλείει την άλλη. Για αυτό το λόγο χρησιμοποιείται το cut μετά τον έλεγχο, ώστε να μην υπάρχει οπισθοδρόμηση στην άλλη εναλλακτική περίπτωση. Αν το X ανήκει στο διάστημα, να μην εκτελέσει και τον κώδικα στον οποίο δεν ανήκει. Ένας άλλος τρόπος με τον οποίο θα μπορούσε να είχε γραφτεί αυτό θα ήταν με not στην άλλη περίπτωση, αλλά το cut φαίνεται πιο κομψό. Ελέγχεται αν το X ανήκει στο διάστημα και όχι το αντίθετο, γιατί τότε θα χρειαζόταν δύο περιπτώσεις, μία όπου το X είναι μεγαλύτερο από το ανώτατο όριο και μία όπου το X είναι μικρότερο από το κατώτατο, δηλαδή περισσότερες γραμμές κώδικα.

<u>Παραδείγματα Εκτέλεσης:</u>

```
?- exclude_range(2, 5, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], List).
List = [1, 6, 7, 8, 9, 10]
Yes (0.00s cpu)

?- exclude_range(3, 3, [1, 2, 3, 5, 10], List).
List = [1, 2, 5, 10]
Yes (0.00s cpu)

?- exclude_range(3, 3, [], List).
List = []
Yes (0.00s cpu)
```

2. Matching Number Series

```
1  math_match([_],_,[]).
2  math_match([X1,X2|Rest], C, [(X1,X2)|Solution]):-
3     Predicate =.. [C,X1,X2],
4     call(Predicate),!,
5     math_match([X2|Rest],C,Solution).
6
7  math_match([_,X2|Rest], C, Solution):-
8     math_match([X2|Rest],C,Solution).
```

- *Βασική περίπτωση:* Αν η λίστα έχει μόνο ένα στοιχείο, όποιο και αν είναι αυτό, τότε δε μπορούν να δημιουργηθούν ζευγάρια και η Λύση είναι κενή.
- *Γενική περίπτωση:* Υπάρχουν δύο εναλλακτικές,
    1) Τα δύο πρώτα στοιχεία ικανοποιούν τη συνθήκη C, άρα προστίθενται στη Λύση και διαγράφεται το πρώτο στοιχείο, ώστε να ελεγχθεί στη συνέχεια το δεύτερο με το τρίτο.
    2) Η συνθήκη δεν ικανοποιείται, οπότε το ζεύγος των τιμών δεν προστίθεται στη Λύση και συνεχίζει με τον ίδιο τρόπο.

Χρησιμοποιείται ακριβώς η ίδια λογική με την πρώτη άσκηση, μόνο που εδώ ο κανόνας που ελέγχεται είναι η επιτυχία ή μη του κατηγορήματος Predicate.

<u>Παραδείγματα Εκτέλεσης:</u>

```
?- math_match([2, 4, 16, 3, 5, 25, 2, 100], square, L).
L = [(2, 4), (4, 16), (5, 25)]
Yes (0.00s cpu, solution 1, maybe more)
No (0.03s cpu)

?- math_match([], square, L).
No (0.00s cpu)

?- math_match([1, 2, 3], square, L).
L = []
Yes (0.00s cpu, solution 1, maybe more)
No (0.00s cpu)
```

```
1  math_match_alt(List, C, Solution):-
2       Predicate =.. [C,X1,X2],
3       setof((X1,X2),(pair(X1,X2,List),call(Predicate)),Solution).
4
5  pair(X,Y,List):-
6      append(L2,[Y|_], List),
7      append(_,[X],L2).
```

Η λογική που χρησιμοποιείται για την παραλλαγή είναι η εξής
“Βρες μου όλες τις λύσεις των X,Y όπου τα X,Y ικανοποιούν τη συνθήκη C και βρίσκονται το ένα δίπλα στο άλλο στη λίστα”

Για τον έλεγχο της δεύτερης προυπόθεσης δημιουργήθηκε το κατηγόρημα pair(X,Y,List), το οποίο πετυχαίνει όταν τα X,Y εμφανίζονται διαδοχικά σε μία λίστα List. Για να βρίσκονται το ένα δίπλα στο άλλο, πρέπει η λίστα να μπορεί να χωριστεί σε δύο υπο-λίστες όπου στην πρώτη το X είναι το τελευταίο στοιχείο (ελέγχεται στη σειρά 7) και στη δεύτερη το Y να είναι το πρώτο.

Η σειρά των append παίζει ρόλο σε αυτή την περίπτωση. Αν ελέγχεται πρώτα αν το X είναι το τελευταίο στοιχείο τότε το πρόγραμμα πέφτει σε Infinite loop, καθώς δημιουργεί όλες τις πιθανές (άπειρες) λίστες που θα μπορούσαν να έχουν το X στο τέλος. Το L2 πρέπει πρώτα να πάρει τιμή και μετά να ελεγχθεί.

<u>Παραδείγματα Εκτέλεσης:</u>

```
?- math_match_alt([2, 4, 16, 3, 5, 25, 2, 100], square, L).
L = [(2, 4), (4, 16), (5, 25)]
Yes (0.00s cpu)

?- math_match_alt([], square, L).
No (0.00s cpu)

?- math_match_alt([1, 2, 3], square, L).
No (0.00s cpu)
```

Χρησιμοποιήθηκαν τα ίδια παραδείγματα για να ελεγχθεί αν θα υπάρξει κάποια διαφορά.

Παρατηρήσεις: Το μόνο πρόβλημα με αυτή τη λύση είναι πως δεν έχει ακριβώς την ίδια συμπεριφορά με την προηγούμενη, Συγκεκριμένα, όταν δεν υπάρχουν ζευγάρια, στην πρώτη λύση επιστρέφεται η κενή λίστα, ενώ στη δεύτερη γίνεται Fail. Αυτό συμβαίνει λόγω της χρήσης του setof, το οποίο κάνει fail όταν δε βρίσκει λύση, όμως επιστρέφει μόνο τις μοναδικές, που είναι πιο επιθυμητό.

3. Stacks of Blocks

```
1   stack_blocks([X,Y,Z],[X2,Y2,Z2],H):-
2       stack(X,Y,Z,H),
3       stack(X2,Y2,Z2,H),
4       different([X,Y,Z],[X2,Y2,Z2]).
5
6   top(B1,B2):-
7       a_block(B1,_,W1),
8       a_block(B2,_,W2),
9       W2 =< W1.
10
11  stack(X,Y,Z, H):-
12      top(X,Y),
13      top(Y,Z),
14      a_block(X,H1,_),
15      a_block(Y,H2,_),
16      Y \= X,
17      a_block(Z,H3,_),
18      Z \= X,
19      Z \= Y,
20      H is H1+H2+H3.
21
22  different(List1,List2):-
23      findall(X, (member(X, List1),member(X,List2)), []).
```

Σε αυτή τη λύση δημιουργήθηκαν πολλά κατηγορήματα με στόχο να σπάσει ο κώδικας και να μη χρειάζεται να γραφούν αναλυτικά όλοι οι έλεγχοι που πρέπει να γίνουν.

- Το κατηγόρημα top(B1,B2) πετυχαίνει όταν τα B1,B2 είναι δύο blocks και το B2 είναι πάνω από το B1.
- Το κατηγόρημα stack(X,Y,Z,H) πετυχαίνει όταν τα X,Y, Z σχηματίζουν μια στοίβα, δηλαδή το Y βρίσκεται πάνω από το X, το Z πάνω από το Y και είναι όλα διαφορετικά μεταξύ τους. Επίσης επιστρέφεται και το ύψος H καθώς χρειάζεται αργότερα.
- Το κατηγόρημα different(List1,List2) πετυχαίνει όταν τα List1 και List2 δεν έχουν κοινά στοιχεία, δηλαδή η findall δε βρίσκει λύση για X που ανήκουν και στις δύο λίστες.

Η επίλυση του προβλήματος γίνεται με Generate and Test, αφού δημιουργούνται στοίβες και μετά τεστάρεται η εγκυρότητά τους.

Παράδειγμα εκτέλεσης με τα παρακάτω γεγονότα:

```
a_block(b1,2,4).
a_block(b2,1,3).
a_block(b3,3,3).
a_block(b4,1,2).
a_block(b5,4,1).
a_block(b6,2,1).
a_block(b7,5,3).
a_block(b8,5,2).
a_block(b9,4,4).
a_block(b10,2,3).
a_block(b11,2,2).
a_block(b12,1,1).
a_block(b13,3,3).
a_block(b14,1,1).
a_block(b15,4,1).
a_block(b16,2,1).
a_block(b17,5,5).
a_block(b18,5,2).
a_block(b19,4,6).
a_block(b20,2,3).
```

```
?- stack_blocks(A, B, H).
A = [b1, b2, b3]
B = [b4, b5, b12]
H = 6
Yes (0.00s cpu, solution 1, maybe more)

A = [b1, b2, b3]
B = [b4, b5, b14]
H = 6
Yes (0.05s cpu, solution 2, maybe more)

A = [b1, b2, b3]
B = [b10, b11, b6]
H = 6
Yes (0.13s cpu, solution 16, maybe more)
```

Παρατηρήσεις: Αυτός ο τρόπος επίλυσης της άσκησης ΔΕΝ είναι ο πιο αποδοτικός. Το πρόβλημα έγκειται στο γεγονός ότι πρώτα δημιουργούνται οι δύο λίστες (σειρές 2-3) και μετά ελέγχεται αν είναι διαφορετικές μεταξύ τους. Αυτό σημαίνει ότι η Prolog μπαίνει στη διαδικασία να βρει μια έγκυρη στοίβα (που χρειάζεται και τους δικούς της ελέγχους), μόνο και μόνο για να την πετάξει στο επόμενο βήμα.
Αποδοτικότερο θα ήταν να γίνονται οι κατάλληλοι έλεγχοι τη στιγμή που χρειάζεται, όχι δηλαδή σε επίπεδο στοίβας, αλλά κάθε φορά που επιλέγεται ένα block, αρχικά να ελέγχεται αν είναι διαφορετικό από τα υπόλοιπα και μετά αν ικανοποιεί τον περιορισμό πλάτους. Για το ύψος να ελέγχεται επίσης αν τα 2 block της δεύτερης στοίβας έχουν μικρότερο ύψος από όλη την πρώτη.

Τελικά επιλέχθηκε ο μη αποδοτικός τρόπος λόγο της ευελιξίας του. Αν όλοι οι περιορισμοί γράφονταν αναλυτικά, θα ήταν δύσκολη για παράδειγμα η προσθήκη μιας τρίτης στοίβας Ίσως η καλύτερη λύση να ήταν μία στην οποία οι δουλειές χωρίζονται με διαφορετικό τρόπο, αλλά οι έλεγχοι πάλι γίνονται την κατάλληλη στιγμή.

```
1   find_min([X],X,0).
2   find_min([X,Y|Tail],Min,Counter):-
3       X=<Y,!,
4       find_min([X|Tail],Min,CounterSoFar),
5       Counter is CounterSoFar+1.
6
7   find_min([X,Y|Tail],Min,Counter):-
8       Y=<X,
9       find_min([Y|Tail],Min,CounterSoFar),
10      Counter is CounterSoFar+1.
11
12  find_lowest_stack(StaA, StaB, Min, Sols):-
13      findall(H,stack_blocks(_A,_B,H),Heights),
14      find_min(Heights,Min,Sols),
15      stack_blocks(StaA,StaB,Min).
```

Η λογική που ακολουθήθηκε είναι η εξής:
“Βρες όλα τα πιθανά ύψη για τις έγκυρες στοίβες και βρες το μικρότερο στοιχείο στη λίστα τους”

Η εύρεση του min έγινε με τον τρίτο τρόπο που αναφέρθηκε στο μάθημα, δηλαδή ελέγχονται τα πρώτα δύο στοιχεία της λίστας, και σκουντιέται το μικρότερο στη λίστα με τα υπόλοιπα στοιχεία μέχρι να μείνει ένα στοιχείο. Έγινε μόνο μια τροποποίηση με το CounterSoFar, το οποίο μετράει πόσοι έλεγχοι γίνονται.

Παράδειγμα εκτέλεσης:

?- find_lowest_stack(StaA, StaB, Min, Sols). (με 10 στοιχεία)
StaA = [b1, b10, b6]
StaB = [b2, b4, b5]
Min = 6
Sols = 1171
Yes (0.03s cpu, solution 1, maybe more)
StaA = [b1, b10, b6]
StaB = [b9, b2, b4]
Min = 6
Sols = 1171
Yes (0.03s cpu, solution 2, maybe more)
StaA = [b2, b4, b5]
StaB = [b1, b10, b6]
Min = 6
Sols = 1171
Yes (0.03s cpu, solution 3, maybe more)
StaA = [b9, b2, b4]
StaB = [b1, b10, b6]
Min = 6
Sols = 1171
Yes (0.03s cpu, solution 4, maybe more)
No (0.03s cpu)

?- find_lowest_stack(StaA, StaB, Min, Sols). (με 20 στοιχεία)
StaA = [b1, b2, b4]
StaB = [b6, b12, b14]
Min = 4
Sols = 229835
Yes (1.95s cpu, solution 1, maybe more)

Παρατηρήσεις: Σχετικά με το CounterSoFar, η τιμή του είναι γνωστή. Εύρεση του min σε λίστα σημαίνει εξαντλητική αναζήτηση, οπότε θα κοιτάξει όλες τις τιμές - 1. Για να υπάρξει βελτίωση στην αποδοτικότητα, πρέπει να αλλάξει όλος ο τρόπος αναζήτησης. Δηλαδή, θα πρέπει να ελέγχεται αν στα δύο blocks έχει ήδη ξεπεραστεί το τρέχων min ύψος.
Ένα δεύτερο σημείο στο οποίο χάνεται η αποδοτικότητα είναι η γραμμή 15, όπου αφού βρεθεί το μικρότερο ύψος, πρέπει να υπολογιστούν ξανά οι στοίβες που έχουν αυτό το ύψος, ενώ υπολογίστηκαν ήδη μια φορά στη findall.

Έγινε προσπάθεια για λύση με τη χρήση του lemma, ώστε να αυξηθεί η αποδοτικότητα, αλλά ήταν ανεπιτυχής. Για κάποιο λόγο, ο κώδικας με το lemma έβγαζε “Out of swap space in heap allocation” Error.