- **Ce se întâmplă când scriem în Prolog L = [H|T]?**

Efectul lui **L = [H|T]** depinde de care dintre cele 3 variabile (L, H și T) este liberă și care este legată.

Nu uitați, Prolog tot timpul încearcă să răspundă la întrebarea: *Este acest lucru posibil?* În cazul nostru, unde vorbim de L = [H|T], Prolog se întreabă dacă: *Este posibil ca L să fie egal cu lista formată prin adăugarea lui H la începutul lui T?* Și în principiu Prolog vrea să răspundă cu YES, de aceea, dacă e necesar o să facă atribuiri (legări de variabile) pentru a face expresia adevărată.

L	H	T	Exemplu	Explicație
Legată	Legată	Legată	L = [1,2,3], H = 1, T = [2,3], L = [H|T]. *sau* L = [1,2,3], H = 1, T = [2,3,4], L = [H|T].	Prolog verifică dacă este posibilă împărțirea listei L în așa fel încât primul element să fie H și restul elementelor să fie cele din T. Prolog returnează *true* (pentru primul exemplu) sau *false* (pentru al doilea).
Legată	Liberă	Liberă	L = [1,2,3], L = [H|T].	Întrebarea la care Prolog răspunde este "*Este posibil ca L să fie egal cu elementul H adăugat la începutul listei T*"? Răspunsul va fi YES, dacă H va fi legat de valoarea 1 și T de valoarea [2,3]. Deci putem spune că Prolog împarte lista în head (1) și tail (lista [2,3]).
Liberă	Legată	Legată	H = 1, T = [2,3], L = [H|T].	Întrebarea la care Prolog răspunde este "*Este posibil ca L să fie egal cu elementul H adăugat la începutul listei T*"? Răspunsul va fi YES, dacă L va fi legat de lista [1,2,3]. Deci putem spune că Prolog construiește lista L adăugând elementul 1 la începutul listei [2,3].
Legată	Liberă	Legată	L = [1,2,3], T = [2,3], L = [H|T] *sau* L = [1,2,3], T = [2,3,4], L = [H|T].	Întrebarea la care Prolog răspunde este "*Este posibil ca L să fie egal cu elementul H adăugat la începutul listei T*"? Din moment ce T are deja o valoare aici răspunsul poate fi NO (adică *false*) (exemplul 2, nu există cum să fie lista [1,2,3] egală cu lista [2,3,4] la care mai adăugăm un element). Dacă elementele din T se potrivesc cu elementele din L, atunci Prolog va lega H de primul element din L, pentru a putea răspunde cu YES (în primul exemplu)
Legată	Legată	Liberă	L = [1,2,3], H = 1, L = [H|T]. *sau* L = [1,2,3], H = 3, L = [H|T].	Foarte similar cu cazul anterior. Dacă H este egal cu primul element din L, atunci Prolog va lega T de sublista din L (primul exemplu) pentru a putea răspunde cu YES la întrebarea "*Este posibil ca L să fie egal cu elementul H adăugat la începutul listei T*?". Dacă H nu este egal cu primul element din L, Prolog va răspunde cu *false*.
Liberă	Legată	Liberă	H = 1, L = [H|T].	Întrebarea la care Prolog răspunde este "*Este posibil ca L să fie egal cu elementul H adăugat la începutul listei T?*". Din moment ce nici L nici T

				nu sunt legate, răspunsul este YES (adică nu avem nimic ce să contrazică acest lucru). Ce este interesant este ce se întâmplă dacă mai târziu pe parcursul execuției L sau T va fi legată la o valoare. Dacă T va fi legată de o valoare, atunci ajungem la cazul 3 din tabel, și L va fi legată la lista formată din H urmat de valorile din T. De exemplu, dacă scriem H = 1, L = [H|T], T = [2,3]. L va fi legată la lista [1,2,3]. Dacă L va fi legată la o valoare, atunci ajungem la cazul 5 din tabel, Prolog va verifica daca primul element din L este egal cu H, dacă da, T va fi legată de restul listei, altfel, Prolog va returna *false*.
Liberă	Liberă	Legată	T = [2,3], L = [H|T].	Similar cu cazul anterior.

Dacă vreți să încercați exemplele, puteți să le introduceți direct în consola de la SWI-Prolog.

- **Ce se întâmplă când avem în Prolog o clauză de genul: p([], []). De unde știe Prolog când vrem verificare și când vrem atribuire?**

Să presupunem că avem o singură clauză în predicatul nostru:

p([], []).

Dacă vrem, putem să o rescriem folosind parametri:

p(List1, List2):- List1 = [], List2 = [].

Se vede că avem aceeași sintaxă pentru ambii parametri. De unde știe Prolog că (de cele mai multe ori) la List1 vrem verificare și la List2 atribuire?

Pe scurt, nu știe. Prolog nu funcționează în termeni de atribuire și de verificări, Prolog vrea potriviri (matches). Ce se va întâmpla depinde de cum apelăm predicatul p. Dacă apelăm cu:

p([], R).

Prolog încearcă să răspundă la întrebarea: *Este o potrivire aici?* Primul parametru se potrivește (are valoarea [] și Prolog vrea valoarea respectivă acolo) iar parametrul 2 este liberă. Răspunsul la întrebare este YES, dacă R va fi legată de valoarea []. Deci se întâmplă o atribuire (legare), R va fi legată de lista vidă, dar ca un fel de efect secundar al faptului că Prolog vrea potrivire.

Dacă noi apelăm predicatul cu:

p(R, []).

Se întâmplă ceva foarte similar. Prolog încearcă să răspundă la întrebarea: *Este o potrivire aici?* Parametrul 2 se potrivește (are valoarea [] și Prolog vrea valoarea respectivă acolo) iar parametrul 1 este liberă.

Răspunsul la întrebare este YES, dacă R va fi legată de valoarea []. Deci se întâmplă o atribuire (legare), R va fi legată de lista vidă, dar ca un fel de efect secundar al faptului că Prolog vrea potrivire.

Dacă apelăm predicatul cu:

p([], []).

Prolog încearcă în continuare să răspundă la întrebarea: *Este o potrivire aici?* E cel mai simplu caz, avem o potrivire, deci Prolog poate răspunde cu YES, nu se vor întâmpla legări de valoare.

Și dacă apelăm predicatul cu:

p(R1, R2).

Prolog încearcă în continuare să răspundă la întrebarea: *Este o potrivire aici?* Răspunsul este YES dacă R1 și R2 (ambele variabile libere) vor fi legate la valoarea [].

Și dacă apelăm predicatul cu:

p([1,2], R).

Prolog încearcă în continuare să răspundă la întrebarea: *Este o potrivire aici?* Răspunsul este NO, primul parametru nu se potrivește cu ce vrea Prolog acolo, nu e lista vidă. În acest caz, nu există nicio posibilitate de a avea o potrivire, deci Prolog returnează *false.*

Concluzie: Ce face o expresie de genul R = [] depinde de variabila R: dacă este legată, atunci se întâmplă o verificare, iar dacă R este liberă, se întâmplă o legare, dar legarea nu e scopul în sine pentru Prolog, scopul e ca Prolog să poată să spună că R se potrivește cu []. Și de unde știm noi dacă se va întâmpla atribuire sau verificare? Trebuie să știm care parametru are valoare și care e liberă. Și acest lucru aflăm din **modelul de flux.**

- **Cum construim lista rezultat?**

Să presupunem că vrem să implementăm un predicat care adaugă un element la sfârșitul unei liste (ați discutat acest predicat și la curs).

Model matematic:

adauga_sf(l_1...l_n, e) =

- (e), dacă n = 0
- l_1 (+) adauga_sf(l_2...l_n, e), altfel

Implementare Prolog:

```
%adauga_sf(L:lista, E: element, R: lista)
%model de flux (i, i, o)
adauga_sf([], E, [E]).
```

```
adauga_sf([H|T], E, [H|Rez]):-
	adauga_sf(T, E, Rez).
```

O greșeală frecventă este ca studenții să implementeze ultima clauză astfel:

```
adauga_sf([H|T], E, Rez):-
	adauga_sf(T, E, [H|Rez]).
```

De ce prima variantă este corectă și această variantă nu este?

Să luăm prima variantă:

```
adauga_sf([H|T], E, [H|Rez]):-
	adauga_sf(T, E, Rez).
```

Ce se întâmplă aici? Presupunem că apelul din consola va fi sub forma adauga_sf([1,2,3], 5, R).

Dacă ne uităm din perspectiva modelului de flux, parametrul 3 (R, cel de output, parametru liber) vrem să fie potrivit (matched) cu expresia [H|Rez]. Dacă ne gândim la tabelul de la începutul documentului, suntem într-o situație R = [H|Rez], unde doar H este legată. Ce se va întâmpla (așa cum v-am descris în tabel) depinde de cine primește valoare prima dată: R sau Rez? Din apel recursiv Rez va primi valoare, deci efectul este ca R să fie legată de lista formată prin adăugarea lui H la începutul listei Rez.

Acest lucru se vede și mai bine, dacă rescriem clauza să avem varianta lungă.

```
adauga_sf([H|T], E, R):-
	adauga_sf(T, E, Rez),
	R = [H|Rez].
```

Să vedem acum ce se întâmplă pe varianta greșită. Din nou, presupunem că apelul din consola va fi sub forma adauga_sf([1,2,3], 5, R).

```
adauga_sf([H|T], E, Rez):-
	adauga_sf(T, E, [H|Rez]).
```

Sunt 2 probleme aici. În primul rând am declarat că rezultatul va fi Rez, și mai târziu am folosit [H|Rez]. Acest lucru înseamnă că H nu va fi parte din Rez, e un element extra pe care îl punem la apel recursiv, dar care nu face parte din ce vom returna. Acest lucru e greșit. Problema a 2-a este că în apelul recursiv avem expresia [H|Rez], unde H este legată și Rez este liberă. După ce Prolog calculează rezultatul apelului

recursiv va încerca să o potrivească (match) cu [H|Rez], unde H este legată și Rez este liberă. Deci suntem din nou într-o situație de genul Result = [H|Rez], unde Result este rezultatul apelului recursiv (nu îl avem reținută într-o variabilă aici), deci este legată. Ce se va întâmpla, conform tabelului de mai sus este că Prolog va verifica dacă Result începe cu valoarea H și dacă da, Rez va fi legată de restul listei. Și noi nu asta vrem.

Dacă facem de exemplu apelul adauga_sf([3], 3, R), având implementarea greșită pentru clauza a 2-a, rezultatul va fi R = []. (Pentru alte liste va returna false). De ce?

Se potrivește clauza a 2-a, și facem acel apel adauga_sf(T, E, [H|Rez]). T este lista vidă, E și H au valoarea 3. Apelul recursiv fiind pentru lista vidă va intra pe prima clauză, și va returna lista formată din elementul E, adică lista [3]. Când ne întoarcem din recursivitate, Prolog va avea: [3] (adică rezultatul apelului recursiv) = [3 | Rez] (unde 3 este valoarea lui H). Din moment ce avem o potrivire, Rez va fi legată la valoarea [] și această valoare va fi returnată. Dacă E nu se potrivea cu singurul element din listă, sau lista era mai lungă, rezultatul era false.

Problema cu această variantă de clauză se vede și mai bine, dacă rescriem clauza folosind forma mai lungă. În loc să avem [H|Rez] pentru parametrul 3 în apel recursiv, vom pune acolo un singur parametru, Result, pe care-l facem egal cu [H|Rez] după apel.

```
adauga_sf([H|T], E, Rez):-
      adauga_sf(T, E, Result),
      Result = [H|Rez].
```

Pe această variantă se vede și mai bine în ce caz suntem: Result și H au valoare, Rez nu are. Și ce se va întâmpla nu este ca la Rez să mai adăugăm elementul H.