4

Controlul backtrackingului

4.1. Predicatul *tăierea*

Predicatul tăierea nu are argumente şi de obicei e notat *!/0*. El este strâns legat de mecanismul backtracking şi poate schimba parcurgerea arborelui de căutare a soluţiilor, modificându-l prin tăierea ramurilor. El nu are o semnificaţie logică, ci numai una operaţional-determinstă legată de parcurgerea standard a arborelui de căutare.

Predicatul *!/0* reuşeşte imediat, dar el nu mai poate fi resatisfăcut. Mai mult ca atât, regula în care el e prezent nu mai poate fi resatisfăcută. Acest predicat predefinit indică sistemului că nu mai poate reveni înapoi după punctul unde el se găseşte. Deci, astfel poate fi redus spaţiul de căutare a soluţiilor.

Predicatul *!/0* se utilizează ca orice subscop în corpul clauzelor. Atenţionăm, că predicatul *!/0* trebuie utilizat cu prudenţă. Cu toate că el are o semnificaţie declarativă simplă (el întotdeauna e adevărat), folosirea lui înseamnă, de fapt, sacrificarea unei părţi a lizibilităţii unui program. În afară de aceasta, tăierea poate întrerupe executarea unui program într-un mod neprevăzut.

În general, tăierea poate fi utilizată, când este găsită o soluţie pentru un scop şi nu suntem interesaţi în căutarea altor soluţii, pentru excluderea din cercetare ale acelor clauze, care nu pot duce spre soluţii, sau pentru schimbarea semanticii procedurale ale predicatelor.

Să considerăm acum un program ce conţine predicate cu variabile şi predicatul *!/0* în diverse poziţii în corpul regulilor.

```
q(a).
q(b).

r(b,b1).
r(a,a1).
r(a,a2).

p(A,B):-q(A), r(A,B).
p(c,c1).

p1(A,B):-q(A), r(A,B),!.
p1(c,c1).

p2(A,B):-q(A),!, r(A,B).
p2(c,c1).

p3(A,B):-!,q(A), r(A,B).
p3(c,c1).
```

Şi, fiind puse următoarele întrebări, se obţin răspunsurile respective:

```
?-p(X,Y)
X=a, Y=a1;
X=a, Y=a2;
X=b, Y=b1;
X=c, Y=c1;
no

?-p1(X,Y)
X=a, Y=a1;
no

?-p2(X,Y)
X=a, Y=a1;
X=a, Y=a2;
no

?-p3(X,Y)
X=a, Y=a1;
X=a, Y=a2;
X=b, Y=b1;
no
```

Deseori, unii programatori utilizează tăierea în cazul când programele furnizează la întrebarea dată acelaşi răspuns de mai multe ori. Nu întotdeauna, însă, aceasta este ieşirea corectă din situaţie. E necesară o atenţie deosebită pentru observarea domeniului de extindere a acţiunii predicatului *!/0*. De exemplu, mulţimea de clauze

```
p:-a,b.
p:-c.
```

e echivalentă expresiei *((a şi b) sau c) implică p*, în timp ce pachetul

```
p:-a,!,b.
p:-c.
```

e echivalent expresiei *((a şi b) sau (non a şi c)) implică p*, pe când programul

```
p:-c.
p:-a,!,b.
```

are semnificaţia expresiei *(c sau (a şi b)) implică p*.

În concluzie la cele spuse mai sus, putem aduce următoarele reguli de utilizare a predicatului tăierea:

- Predicatul *!/0* stopează considerarea tuturor clauzelor din pachet ce urmează după el. Adică, dacă un scop s-a unificat cu capul clauzei ce conţine *!/0* şi predicatul *!/0* este atins, atunci acest scop nu poate fi utilizat pentru construirea soluţiilor cu clauze cu acelaşi cap ce urmează mai jos.
- Predicatul *!/0* înlătură toate soluţiile de alternativă pentru subscopurile plasate în stânga sa. Conjuncţia de subscopuri, ce stau înaintea semnului de tăiere, produce nu mai mult de o soluţie.
- Predicatul *!/0* nu influenţează asupra subscopurilor ce sunt în dreapta sa. Însă, dacă careva subscop din dreapta eşuează, atunci căutarea trece la ultima alegere ce a fost făcută până a trece la clauza cu predicatul *!/0*.

Neînţelegerea mecanismului de tăiere este o sursă de greşeli chiar şi pentru programatorii experimentaţi. Greşelile pot fi de două feluri:

- se taie ramuri ale arborelui de căutare ce nu trebuie tăiate, deoarece conţin soluţii

- nu se taie acele ramuri ce nu conţin soluţii

Adică există două contexte diferite în care se poate utiliza predicatul *!/0*. Într-un context, predicatul se introduce numai pentru creşterea eficienţei programului - caz în care el se numeşte tăierea verde. În alt context, utilizarea lui *!/0* modifică semnificaţia procedurală a programului - caz în care el se numeşte tăierea roşie.

4.2. Tăierea verde

Tăierea poate servi pentru optimizarea căutării, evitându-se căutările inutile. Cu alte cuvinte, se poate recurge la stoparea căutărilor, dacă răspunsul deja este găsit. Gruparea unei mulţimi de obiecte în categorii ce se exclud mutual este o utilizare clasică a tăierii verzi. Să examinăm, de exemplu, următorul predicat destinat determinării tipului listei.

Fiind dată o relaţie definită de mai multei clauze, chemarea predicatului selectează (prin unificare) o singură clauză, dacă formele capetelor lor se exclud mutual. Să considerăm, de exemplu, următorul program:

```
tip_lista([],"lista vida").
tip_lista([_|_],"lista nevida").
```

Dacă în întrebarea

```
?-tip_lista(L,M).
```

L este instanţiată cu vre-o listă, *M* va fi instanţiat de unul din atomii *"lista vida"* sau *"lista nevida"*. Dacă *L* este instanţiată cu o listă vidă, o eventuală tentativă de a resatisface scopul cu clauza a doua eşuează, nefiind posibilă unificarea. În acest caz, programatorul poate comunica sistemului că este inutilă încercarea de aplicare a clauzei a doua. Dacă în prima clauză se introduce predicatul *!/0*, se confirmă acest lucru.

```
tip_lista([],"lista vida"):-!.
tip_lista([_|_],"lista nevida").
```

Tăierea în clauza a doua este inutilă, deoarece după ea nu urmează altă clauză. Este evident, că în acest caz semnificaţia declarativă a predicatului este aceeaşi cu sau fără tăiere.

Introducerea predicatului de tăiere nu afectează comportamentul acestui program. Predicatul *tip_lista/2* va reuşi sau va eşua pentru aceleaşi valori ale primului argument în ambele versiuni ale programului. Versiunea a doua, însă, este mai eficientă şi dacă acest predicat va fi parte a altui program, atunci backtrackingul va sări peste predicatul *tip_lista/2*. Câştigul în acest exemplu nu este considerabil, deoarece subscopurile sărite nu sunt calcule mari, dar în general utilizarea predicatului *!/0* pentru indicarea excluderii mutuale între clase poate aduce un câştig important de timp.

Se poate generaliza programul pentru mai multe cazuri:

```
tip_lista([],"lista vida"):-!.
tip_lista([_],"lista cu un element"):-!.
tip_lista([_,_],"lista cu doua elemente"):-!.
tip_lista([_,_,_|_],"lista cu mai mult de 2
              elem.").
```

Dacă în întrebarea

```
?-tip_lista(L,M).
```

L este un termen diferit de listă, predicatul va eşua, furnizându-ne (în mod implicit) acest mesaj. Dacă dorim să se explice acest caz, se poate completa, de exemplu, primul program:

```
tip_lista([],"lista vida"):-!.
tip_lista([_|_],"lista nevida"):-!.
tip_lista(_,"nu e lista").
```

Situaţia acum, aparent analogică cu precedenta, în realitate este diferită. De fapt, în acest caz cele trei forme ale primului argument nu se exclud mutual, deoarece primele două sunt cazuri particulare ale celei de a treia. Adică, atunci când o instanţă se unifică cu prima sau a doua, ea se va unifica şi cu a treia. Fără tăiere predicatul, după ce furnizează prima soluţie *"lista vida"* sau *"lista nevida"*, în caz de resatisfacere, va da *"nu e lista"* în calitate de a doua soluţie.

Să observăm că în ultimul exemplu tăierea nu este verde. Problema constă în faptul, că se doreşte ca clauza a treia să se aplice numai în cazul când nu se aplică nici una din primele clauze şi soluţia să nu depindă de forma pe care o are primul argument. Utilizarea tăierii are acest efect, dar semnificaţia declarativă a predicatului cu tăiere nu mai este aceeaşi ca în cazul când acest predicat lipseşte.

În multe cazuri nu este posibilă deosebirea intrării numai în baza formei sale (singurul caz în care acţionează unificarea), dar sunt necesare şi condiţii adăugătoare. Considerăm, de exemplu, relaţia *mai_mare(X,Y,Z)*: "*Z* este numărul mai mare dintre cele două, *X* şi *Y*". În definiţia

```
mai_mare(X,X,X).
mai_mare(X,Y,Y):-X<Y.
mai_mare(X,Y,X):-X>Y.
```

după formă se poate deosebi dacă două numere sunt egale sau diferite. În clauza doi apare o condiţie care ajută să se stabilească care este mai mic. Definiţia are o semnificaţie declarativă deosebită. Procedural, însă, conţine o redundanţă. Clauzele sunt mutual exclusive, dar a doua şi a treia conţin câte o condiţie de comparare în calitate de subscopuri. De acea, dacă a doua clauză reuşeşte, o eventuală tentativă de resatisfacere va apela a treia clauză, dar subscopul respectiv va eşua. Astfel, tentativa este inutilă. Această redundanţă nu este relevantă, dar devine semnificativă când se consideră în contextul altui predicat, cum ar fi predicatul *maximum(X,L)*: "*X* este elementul maximum al listei de numere *L*".

```
maximum(M,[M]).
maximum(Min,[T1,T2|C]):-
       maximum(M,[T2|C]),
       mai_mare(T1,M,Min).
```

În predicatul *mai_mare/3* se poate evita redundanţa, utilizând tăierea ca confirmare a clauzei alese.

```
mai_mare(X,X,X):-!.
mai_mare(X,Y,Y):-X<Y,!.
mai_mare(X,Y,X):-X>Y.
```

Considerările desfăşurate până acum şi ilustrate de exemplele precedente pot fi generalizate. Utilizarea tăierii pentru confirmarea alegerii unei reguli oferă o realizare în mod eficient a predicatului ce defineşte o relaţie articulată în cazuri. Astfel, relaţia este constituită dintr-o mulţime de clauze, fiecare din ele tratând un tip particular de intrări. În aceste circumstanţe se poate utiliza tăierea verde pentru a semnaliza sistemului despre inaplicabilitatea alternativelor clauzei selectate. Astfel se evită risipirea timpului şi a memoriei în tentative inutile.

Acest tip de utilizare a predicatului *!/0* consideră subscopurile, care îl preced în clauză, ca condiţii care, după cea a formei din capul clauzei, specifică clasa de intrări. Dacă aceste condiţii sunt satisfăcute, clauza care le conţine va fi considerată ultima din pachet, chiar dacă sunt şi altele, şi chiar dacă subscopurile, ce urmează după tăiere, eşuează. Dacă *!/0* este

primul şi, eventual, unicul subscop prezent în clauză, singurile condiţii de care depinde reuşita clauzei sunt cele reprezentate de forma argumentelor din capul clauzei.

4.3. Tăierea roşie

Precum s-a observat, adăugarea în programele de mai sus a predicatului de tăiere nu afectează mulţimea de soluţii. Aceasta, însă, nu întotdeauna are loc. În literatură se mai cunoaşte şi tăierea roşie. Tăierea verde înlătură ramurile arborelui care nu duc spre găsirea noilor soluţii. Deci, tăierea verde este o expresie a determinismului. Iar tăierile ce nu sunt verzi, se numesc roşii.

Un programator intenţionat să scrie programe eficiente, privind la predicatul *mai_mare/3* din secţiunea precedentă, va observa că poate fi satisfăcută doar o condiţie din $X=Y$, $X>Y$ şi $X<Y$ pentru orice valori ale lui X şi Y. Reieşind din aceasta, el poate modifica definiţia predicatului *mai_mare/3*, reprezentându-l

```
mai_mare(X,X,X):-!.
mai_mare(X,Y,Y):-X<Y,!.
mai_mare(X,Y,X).
```

sau mai succint

```
mai_mare(X,Y,Y):-X<=Y,!.
mai_mare(X,_,X).
```

Ideea ce a stat la baza acestei modificări este următoarea. Dacă X este mai mic sau egal cu Y, atunci Y e mai mare; în caz contrar mai mare e X. Adică nu e nevoie de compararea lui X şi Y în a doua clauză. Dar astfel de raţionament are un neajuns foarte serios. Ultima versiune a predicatului *mai_mare/3* are o comportare diferită de prima. În particular, întrebarea

```
?-mai_mare(1,4,1)
yes
```

reuşeşte, demonstrând, de fapt, că ultima definiţie are altă semnificaţie.

Astfel, distrugând corespondenţa dintre interpretarea declarativă şi cea procedurală, este obţinut un câştig în eficienţă. Dacă se citesc aceste două clauze, ignorând tăierea, nu este definită relaţia precedentă. Utilizarea tăierii, de fapt, a substituit prezenţa unei condiţii, din punct de vedere declarativ, esenţiale. Pe lângă aceasta, efectul dorit există, numai dacă se respectă aceeaşi ordine a clauzelor. Tăierea prezentă este roşie.

Problema constă în unificarea implicită a argumentelor doi şi trei în prima clauză. Soluţia de înlăturare a acestui neajuns poate fi găsită în reprezentarea explicită a unificării în prima clauză:

```
mai_mare(X,Y,Z):-X<=Y,!,Z=Y.
mai_mare(X,Y,X).
```

O astfel de utilizare a tăierii după o unificare parţială este o tehnică uzuală. Dar în cazul programului de calculare a numărului mai mare, această tehnică e prea complicată. Tăierea în prima clauză continuă să fie roşie.

Predicatul *!/0* poate fi utilizat în interiorul unei reguli recursive pentru îmbunătăţirea eficienţei. Considerăm, de exemplu, predicatul *eliminare(E,L1,L2)* cu semnificaţia declarativă: "Lista *L2* este lista *L1* fără orice apariţie a elementului *E*".

```
eliminare(Elem,[Elem|Coada],Rez):-
        eliminare(Elem,Coada,Rez).
eliminare(Elem,[Cap|Coada1],[Cap|Coada2]):-
        Elem<>Cap,
        eliminare(Elem,Coada1,Coada2).
eliminare(_,[],[]).
```

Prima regulă recursivă este utilizată când primul element din listă coincide cu elementul de eliminare. A doua clauză recursivă e utilizată – când nu coincide. La orice pas ce nu duce la obţinerea listei vide, poate fi aleasă numai una din clauzele recursive. Astfel, toate trei clauze se exclud mutual şi se poate obţine o definiţie mai eficientă, dacă se utilizează predicatul *!/0*.

```
eliminare(Elem,[Elem|Coada],Rez):-
        !,eliminare(Elem,Coada,Rez).
eliminare(Elem,[Cap|Coada1],[Cap|Coada2]):-
        Elem<>Cap,!,
        eliminare(Elem,Coada1,Coada2).
eliminare(_,[],[]).
```

Aici toate tăierile sunt verzi. Dar prezenţa tăierii în prima clauză exclude alternativa exprimată de clauza a doua. În regulile recursive sunt reprezentate cele două cazuri diferite: coincide sau nu capul listei cu elementul destinat eliminării. Caracterul mutual eliminatoriu al acestor două condiţii este exprimat de cele două predicate *!/0*. Deoarece eşuarea primei clauze presupune că elementul *Elem* nu e egal cu capul listei, atunci verificarea explicită a inegalităţii din corpul clauzei doi este de prisos:

```
eliminare(Elem,[Elem|Coada],Rez):-
        !,eliminare(Elem,Coada,Rez).
eliminare(Elem,[Cap|Coada1],[Cap|Coada2]):-
        !,eliminare(Elem,Coada1,Coada2).
eliminare(_,[],[]).
```

În această variantă tăierea din clauza întâi este roşie, iar din clauza a doua este verde. Să menţionăm că, în caz general, nu e raţional de exclus din clauze unele verificări simple. În cazul eliminării verificărilor, câştigul în eficienţă este mizer în comparaţie cu pierderea lizibilităţii programului.

Predicatele considerate au ieşiri multiple, dar un singur prototip, adică un mod de utilizare. În anumite cazuri efectul introducerii predicatului *!/0* poate fi mai subtil: tăierea inclusă în program drept verde, adică pentru ridicarea eficienţei, este de fapt roşie. Să examinăm un exemplu de predicat cu mai multe moduri de utilizare, cum ar fi predicatul *apartine/2*.

```
apartine(Elem,[Elem|_]):-!.
apartine(Elem,[_|Coada]):-
        apartine(Elem,Coada).
```

Odată ce s-a descoperit că un element este membru al unei liste este inutilă încercarea de resatisfacere a scopului. Cu toate aceasta, predicatul *!/0* a schimbat comportamentul programului. De exemplu, la întrebarea

```
?-apartine(Elem,[1,2,3])
```

răspunsul este

```
Elem=1
```

Lectura declarativă a predicatului *apartine(E,L)* este "Elementul *E* aparţine listei *L*", independent cum sunt instanţiate argumentele. Semnificaţia procedurală, invers, depinde de instanţierea argumentelor şi este diferită în cele două versiuni. Versiunea fără tăiere are semnificaţia procedurală: "Să se verifice apartenenţa unui element unei liste, dacă ambii sunt date sau să se furnizeze elementele unei liste date". Versiunea cu tăiere, dimpotrivă, are semnificaţia procedurală: "Să se verifice

apartenența unui element unei liste, dacă ambii sunt date sau să se furnizeze primul element al listei date". Deci, tăierea în această definiție este roșie.

4.4. Predicatul *fail/0* şi conjucția *!,fail*

În Prolog backtrackingul este provocat de un scop care eşuează. Dar în unele situații pentru a găsi soluțiile de alternativă este necesară forțarea backtrackingului. Cu acest țel în Prolog există un predicat special, *fail/0.* El impune eşecul, "încurajând" astfel backtrackingul. Efectul predicatului *fail/0* corespunde efectului comparației *2=3* sau oricărui alt subscop imposibil. Datorită acestui lucru, introducerea unui predicat *fail/0* într-o conjuncție de subscopuri (de obicei la sfârşit, căci după *fail/0* tot nu se mai poate satisface nici un subscop) determină intrarea în procesul de backtracking.

Să ilustrăm aplicarea predicatului *fail/0* pentru afişarea listei femeilor din familia considerată în secțiunile precedente.

```
femeie(maria).
femeie(eliza).
femeie(emilia).

afisare:-
       femeie(FEMEIE),
       write(FEMEIE),nl,
       fail.
afisare.
```

Predicatul *afisare/0* utilizează *fail/0* pentru a forța backtrackingul şi, în rezultat, vor fi afişate toate soluțiile. Când s-a afişat ultimul nume, subscopul *femeie(FEMEIE)* eşuează şi controlul trece la clauza a doua a predicatului *afişare/0* care, fiind un fapt fără argumente, întotdeauna reuşeşte.

Dacă *fail/0* se întâlneşte după predicatul *!/0*, nu se mai face backtracking. Acest lucru se poate utiliza când e posibilă specificarea cazului în care un subscop nu poate reuşi, iar tratarea cazurilor de reuşită e destul de complexă. Atunci se izolează cazul când eşecul este inevitabil. Pentru aceasta se foloseşte asocierea *!, fail*.

Drept exemplu poate servi afirmația "Ioana iubeşte toate florile cu excepția garoafelor". Această afirmație se traduce în Prolog astfel:

```
iubeste(ioana,garoafa):-!,fail.
iubeste(ioana,X):-floare(X).
```

4.5. Bucle repetitive

Predicatele *!/0* şi *fail/0* se aplică în construirea buclelor repetitive. Un exemplu simplu de buclă repetitivă poate fi programul de detectare a unor erori. Considerăm următorul program care cere de la utilizator să insereze numărul calului (de la 1 la 9) ce va sosi primul la o cursă de alergări.

```
primul(N):-
       write( "Introduceți numărul calului: "),
       readchar(N).
```

Acesta e un program corect, dar el nu localizează greşelile. Adică el presupune, că utilizatorul introduce corect datele: date de tip întreg. Dacă utilizatorul inserează greşit, de exemplu, tastează din greşeală o tastă improprie sau tastează caracterul 'n', dorind să insereze cuvântul "nouă" în loc de 9, atunci e cert că programul nu-şi va atinge scopul. Utilizarea predicatului *primul/1* poate presupune aplicarea unei operații asupra *N*, ceea ce nu se va putea face în cazul când *N* nu va fi un termen numeric. Acest neajuns poate fi evitat, dacă în programul de mai sus se construieşte o buclă repetitivă.

```
primul(N):-
       repetare,
       write( "Introduceți numărul calului: "),
       readchar(N),
       N>'0',N<='9',!.
```

E cunoscut faptul, că backtrackingul este mecanismul cu care se obțin toate soluțiile unui scop. Dar backtrackingul se poate folosi pentru a introduce repetări chiar şi în cazul când scopul nu are multiple soluții. Acest lucru se poate produce cu ajutorul unui predicat definit prin două clauze:

```
repetare.
repetare:-repetare.
```

Acest predicat acționează asupra structurii de control a Prologului, de parcă el ar avea un număr infinit de soluții. Predicatul *repetare/0* întotdeauna reuşeşte când este apelat şi întotdeauna reuşeşte în backtracking. Adică, dacă Prologul trece înapoi la predicatul *repetere/0,* el apoi iarăşi va merge înainte. Condiția de testare, adică condiția de ieşire din buclă va fi plasată la sfârşitul buclei şi va reuşi, dacă se introduce un tip de date corect. Adică, va suferi eşec şi se va reveni înapoi, dacă utilizatorul inserează date inacceptabile. Astfel, Prologul se va întoarce înapoi la *repetare/0* din prima clauză şi va pleca din nou pentru a cere introducerea datelor. Prin urmare, Prologul nu va părăsi bucla până când utilizatorul nu va insera valoarea corespunzătoare.

Dar să menționăm, că există un moment destul de neprietenos pentru utilizator, dacă el continuă să introducă date improprii. Întoarcerea înapoi va continua fără să fie aduse explicații de ce; fără a se explica ce se cuvine să se introducă. Mai prietenoasă pentru utilizator ar fi bucla de prindere a erorilor, care ar afişa un mesaj de explicare că a fost o inserare greşită, înainte de a se întoarce înapoi spre predicatul *repetare/0*. Pentru aceasta se separă controlul într-un predicat distinct *verificare/1*. El reuşeşte, dacă, ca mai înainte, intrarea este corespunzătoare, în caz contrar, se trece la clauza a doua, unde se afişează mesajul explicativ, apoi se întâlneşte perechea *!,fail*. Asocierea *!,fail* este partea centrală a predicatului *not/1* în Prolog (vezi capitolul următor). Să se observe, că regula *verificare(N),* sfârşită cu *!,fail,* parcă ar fi regula *not(verificare(N))*, adică cazul când verificarea nu este adevărată pentru o valoare dată a lui *N*. Dar se ştie, că nu se poate scrie explicit o regulă *not(verificare(N)):-…*; negația nu poate apărea în capul regulii. Astfel, regula noastră reuşeşte în cazul când *N* nu corespunde cerințelor.

```
primul(N):-
       repetare,
       write( "Introduceți numărul calului: "),
       readchar(N),
       verificare(N),!.

verificare(N):-
       N>'0',N<='9',!.
verificare(N):-
       write("Introduceți, vă rog, un întreg",
       " mai mare ca 0 şi mai mic ca 10"),nl,
       !,fail.
```

În general vorbind, există două metode principale de formare a buclelor repetitive. În primul rând, este metoda *Eşuează până când reuşeşte,* reprezentată în Fig. 4.4. Aici

Prologul intră într-o buclă, unul din ultimele subscopuri ale căreia este o condiție care de obicei eşuează, dar eventual poate reuşi. Condiția de verificare poate fi, de exemplu, un număr, fie *Cnt=20*, unde *Cnt* este un contor ce numără de câte ori Prologul a trecut prin buclă. Astfel, când contorul atinge numărul *20*, în consecință *20=20*, Prologul reuşeşte şi abandonează bucla. Utilizarea buclei repetitive pentru localizarea erorilor urmează această metodă.

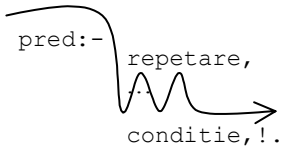

Fig. 4.4. Bucla *Eşuează până când reuşeşte*

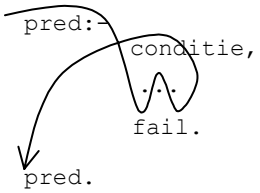

Fig. 4.5. Bucla *Reuşeşte până când eşuează*

A doua metodă este *Reuşeşte până când eşuează,* reprezentată în Fig. 4.5. Aici, în locul repetării, la începutul buclei este o condiție, care de obicei reuşeşte un număr finit de ori. O astfel de buclă a fost examinată în una din secțiunile precedente. Utilizarea acestei bucle este recomandată când condiția poate fi, de exemplu, de forma *rel(X,Y)*, unde relația *rel/2* e definită de un număr de clauze

```
rel(a,b).
rel(c,d).
…
```

în altă parte a programului. Această buclă va impune Prologul să examineze fiecare din clauzele predicatului *rel/2* exact odată. De fiecare dată când reuşeşte, el poate face careva operații asupra termenilor (*X* şi *Y*), obținuți din relația *rel(X,Y)*, apoi, la sfârşitul buclei întâlneşte *fail/0* şi revine înapoi, unde încearcă să examineze următoarea clauză a relației *rel/2*.

Astfel, se examinează toate clauzele şi *pred/0* eşuează, neavând alte alternative ale clauzei *rel/2*. Efectiv, sistemul a făcut tot ce trebuie, iar noi am dori că predicatul *pred/0* să reuşească. Pentru aceasta se introduce o clauză-fapt a predicatului *pred/0* după clauza principală. Astfel, se asigură că Prologul va trece prin ea şi va reuşi.

Atenționăm, că trebuie să se evite compoziția ambelor scheme într-o singură procedură, care, de fapt, va fi o buclă fără ieşire:

```
fara_sfarsit:-
       repetare,
       …
       fail.
```