Programowanie deklaratywne

Artur Michalski
Informatyka II rok

Plan wykładu

- Wprowadzenie do języka Prolog
- Budowa składniowa i interpretacja programów prologowych
- Listy, operatory i operacje arytmetyczne
- Złożone/abstrakcyjne struktury danych
- Sterowanie mechanizmem nawrotów
- Operacje wejścia/wyjścia w Prologu
- Predefiniowane procedury prologowe
- *Styl i techniki programowania w Prologu*

Styl i techniki programowania w Prologu

- Ogólne zasady poprawnego programowania w Prologu
- Jak interpretować programy prologowe?
- Styl programowania
- Efektywność programów prologowych

Ogólne zasady poprawnego programowania w Prologu

Kryteria oceny programowania:

- Poprawność - program realizuje przyjęte na początku założenia i generuje oczekiwane wyniki
- Efektywność - program nie zużywa niepotrzebnie zasobów systemu komputerowego

Ogólne zasady poprawnego programowania w Prologu

Kryteria oceny programowania (c.d.):

- Czytelność - program jest łatwy w interpretacji i zrozumieniu; nie jest bardziej skomplikowany niż to konieczne; struktura i budowa jest przejrzysta
- Modyfikowalność - program łatwo poddaje się zmianom, rozszerzeniom i ulepszeniom

Ogólne zasady poprawnego programowania w Prologu

Kryteria oceny programowania (c.d.):

- Odporność - program powinien być przygotowany na pewne błędy i niepoprawne dane; zachowywać się racjonalnie w obliczu drobnych pomyłek
- Dokumentacja - program powinien być właściwie opisany; minimalny wymóg to komentarz reguł programu prologowego

Ogólne zasady poprawnego programowania w Prologu

- Znaczenie poszczególnych kryteriów zależy od zadania, które realizuje program, od okoliczności w jakich program jest tworzony oraz od środowiska w jakim będzie wykorzystywany
- **Najważniejszym kryterium jest poprawność i żadne inne kryterium nie może być traktowane jako bardziej istotne**
- Punktem wyjścia do procesu pisania programu powinna być zawsze dogłębna analiza i zrozumienie problemu
- Ogólne zasady programowania w Prologu są tymi samymi zasadami, które stosuje się w innych paradygmatach programowania

Jak myśleć o programie prologowym?

Podstawowe techniki wykorzystywane podczas programowania w Prologu:

- Rekurencja
- Generalizacja
- Reprezentacja graficzna

Jak myśleć o programie prologowym?

Rekurencja

Problem zawsze można zredukować do przypadków należących do dwóch grup:

- przypadki trywialne, podstawowe lub „brzegowe”
- przypadki regularne, typowe, w których rozwiązanie jest konstruowane na podstawie rozwiązania zredukowanego przypadku problemu pierwotnego

Technika ta jest podstawową metodą programowania w Prologu.

Jak myśleć o programie prologowym?

Rekurencja - przykład

Przekształcanie wszystkich obiektów z listy zgodnie z pewną zadaną transformacją. Predykat **`map(List,F,NewList)`**, gdzie: **`List`** - lista pierwotna, **`F`** - funkcja transformacji (relacja binarna), **`NewList`** - lista wynikowa.

Rozwiązanie - dwa przypadki:

- przypadek „graniczny”: **`List=[]`**
  jeśli **`List=[]`**, to **`NewList=[]`** niezależnie od **`F`**
- przypadek typowy: **`List=[X|Tail]`**
  jeśli **`List=[X|Tail]`**, to:
  przekształcamy **`X`** zgodnie z **`F`** i otrzymujemy **`NewX`** oraz
  przekształcamy listę **`Tail`**, w listę **`NewTail`**;
  kompletna przekształcona lista to **`[NewX|NewTail]`**.

Jak myśleć o programie prologowym?

Rekurencja - przykładu ciąg dalszy

Rozwiązanie w Prologu:

```
map([],_,[]).
map([X|Tail],F,[NewX|NewTail]):-
 G =.. [F,X,NewX],
 call(G),
 map(Tail,F,NewTail).
```

Najważniejszą z przyczyn, dla których stosujemy rekursję jest rekurencyjny charakter struktur danych wykorzystywanych w języku Prolog.

Jak myśleć o programie prologowym?

Generalizacja

- Problem, który próbujemy rozwiązać często jest przypadkiem szczególnym innego, ogólniejszego problemu
- Znalezienie rekurencyjnego rozwiązania zadania ogólnego umożliwia rozwiązanie zadania pierwotnego, będącego jego przypadkiem szczególnym
- Generalizacja wymaga w reguły wprowadzenia dodatkowych argumentów w opisie zadania
- Zasadnicza trudność polega na konieczności głębokiej analizy problemu i znalezieniu odpowiedniego uogólnienia

Jak myśleć o programie prologowym?

Generalizacja - przykład „Problem 8 hetmanów”

Załóżmy, że próbujemy znaleźć rozwiązanie zadania *8 hetmanów,* w którym żaden z ośmiu hetmanów umieszczonych na szachownicy nie atakuje innego hetmana.

Rozwiązanie zadania
Predykat **`eightqueens(Pos)`**,
gdzie: **`Pos`** - zawiera informacje o położeniu wszystkich hetmanów.

Zadanie uogólnione
Predykat **`nqueens(N,Pos)`**,
gdzie: **`N`** - oznacza aktualną liczbę hetmanów.

Jak myśleć o programie prologowym?

Generalizacja - przykład „Problem 8 hetmanów”

Rozwiązanie zadania ogólnego - dwa przypadki:
- przypadek „graniczny”: **`N=0`**
  jeśli **`N=0`**, to rozwiązania jest trywialne
- przypadek typowy: **`N>0`**
  żeby umieścić **`N`** hetmanów **`(N>0)`** należy:
  znaleźć poprawną konfigurację dla **`N-1`** hetmanów
  oraz dodać nowego hetmana tak, aby nie atakował on pozostałych hetmanów

Rozwiązanie problemu ogólnego pozwala w prosty sposób otrzymać wynik dla zadania pierwotnego:

```
eightqueens(Pos):- nqueens(8,Pos).
```

Jak myśleć o programie prologowym?

Generalizacja - przykład „Arytmograf”

Napisać program rozwiązujący arytmograf sformułowany za pomocą równań postaci:

```
  D O N A L D
+ G E R A L D
-------------
  R O B E R T
```

przy czym każdej literze musi zostać przypisana inna cyfra.

Jak myśleć o programie prologowym?

Generalizacja - przykład „Arytmograf”

Zadanie

Zdefiniować predykat **`sum(N1,N2,N)`**, gdzie **`N1`**, **`N2`**, **`N`** reprezentują odpowiednio pierwszą, drugą i trzecią liczbę łamigłówki, zaś cel **`sum`** jest spełniony, jeżeli istnieje takie przypisanie cyfr do liczb, że **`N1+N2=:=N`**.

Reprezentacja danych: listy zmiennych, którym zostaną przypisane odpowiednie cyfry.

Cel główny

```
?- sum([D,O,N,A,L,D], [G,E,R,A,L,D],
[R,O,B,E,R,T]).
```

Jak myśleć o programie prologowym?

Generalizacja - przykład „Arytmograf”

Predykat **sum1(N1,N2,N,C1,C,Ds1,Ds)** realizuje dodawanie dziesiętne z cyfrą przeniesienia z prawej (**C1**) i cyfrą przeniesienia na lewo (**C**) oraz zbiorami cyfr dostępnych (**Ds1**) i nie wykorzystanych (**Ds**).

Przykładowe wywołanie predykatu **sum1**:

```
?- sum1([H,E],[6,E],[U,S],1,1,
        [1,3,4,7,8,9],Ds).
H=8
E=3
U=4
S=7
Ds=[1,9]
```

Jak myśleć o programie prologowym?

Generalizacja - przykład „Arytmograf”

Związek predykatu **sum(N1,N2,N)** z ogólniejszym predykatem **sum1(N1,N2,N,C1,C,Ds1,Ds)** wynika w warunków początkowych zadania:

```
sum(N1,N2,N):-
sum1(N1,N2,N,0,0,[0,1,2,3,4,5,6,7,8,9],_).
```

Jak myśleć o programie prologowym?

Reprezentacja graficzna problemu

- Reprezentacja graficzna problemu ułatwia zrozumienie zależności występujących w zadaniu
- Prolog jest szczególnie predestynowany do rozwiązywania problemów dotyczących obiektów i relacji między nimi
- Dane strukturalne wykorzystywane w Prologu są reprezentowane w sposób naturalny za pomocą struktur drzewiastych
- Interpretacja deklaratywna programu ułatwia zamianę reprezentacji graficznej w formę klauzul, gdyż kolejność opisu obrazów nie ma najczęściej znaczenia

Styl programowania

Przyczyny zaleceń stylistycznych w programowaniu:

- konieczność redukcji powtarzalnych błędów programistycznych
- poprawa czytelności programu, która decyduje o prostocie modyfikacji, poprawiania i ulepszania programu

Styl programowania

Podstawowe zasady dobrego stylu programowania w Prolog:

- definicje klauzul powinny być krótkie, powinny się składać nie więcej niż kilku podcelów (warunków)
- pojedynczy predykat powinien być reprezentowany za pomocą co najwyżej kilku klauzul; rozbudowane definicje są dopuszczalne, o ile mają dość jednolitą i powtarzalną strukturę
- nazwy predykatów, funktorów i zmiennych powinny być zrozumiałe - wyrażać przypisane im znaczenie
- struktura całego programu powinna być czytelna i spójna; użycie odstępów, tabulacji oraz pustych linii powinno służyć zwiększeniu czytelności; klauzule tego samego predykatu powinny być grupowane razem; zalecane jest również umieszczanie każdego podcelu (warunku) w innym wierszu

Styl programowania

Podstawowe zasady dobrego stylu programowania w Prolog:

- konwencja stylistyczna może być różna dla różnych osób i/lub różnych programów, lecz powinna być spójna w ramach jednego projektu
- operator odcięcia powinien być wykorzystywany z należytą ostrożnością i tylko tam, gdzie to niezbędne; o ile to możliwe należy stosować odcięcia „zielone” a nie „czerwone”; te ostatnie powinny być ograniczone do dwóch przypadków: konstrukcja logicznej negacji i selekcja alternatyw (if...else)
- należy zawsze pamiętać o postaci definicji predykatu **`not`**; stosować go tam, gdzie użycie odcięć może zmniejszyć czytelność programu

Styl programowania

Podstawowe zasady dobrego stylu programowania w Prolog:

- zmiany programu wywołane zastosowaniem predykatów **`assert`** i **`retract`** mogą w znaczący sposób ograniczyć zrozumienie jego działania - program może zachowywać się inaczej w innym czasie; jeśli zachowanie ma być powtarzalne musimy zadbać o odpowiednie odtwarzanie stanów sprzed użycia tych predykatów
- zastosowanie średnika (alternatywa celów) może czasami zmniejszyć czytelność klauzuli; można ją zwiększyć poprzez rozbicie definicji klauzuli na dwie alternatywne klauzule

Styl programowania

Przykład **złego** stylu programowania (nadużycia średnika)

Predykat **`merge(L1,L2,Lwy)`** łączenia posortowanych list wejściowych **`L1`** i **`L2`** w posortowaną listę wynikową:

```
merge(L1,L2,Lwy):-
  L1=[],!,Lwy=L2;
  L2=[],!,Lwy=L1;
  L1=[X|T1],L2=[Y|T2],
  (X<Y,!,Z=X,merge(T1,L2,T3);
  Z=Y,merge(L1,T2,T3)),
  Lwy=[Z|T3].
```

Styl programowania

Przykład **dobrego** stylu programowania:

Predykat **`merge(L1,L2,Lwy)`** łączenia posortowanych list wejściowych **`L1`** i **`L2`** w posortowaną listę wynikową:

```
merge([],L,L):- !.
merge(L,[],L).
merge([X|T1],[Y|T2],[X|T3]):-
  X<Y,!,
  merge(T1,[Y|T2],T3).
merge(L1,[Y|T2],[Y|T3]):-
  merge(L1,T2,T3).
```

Efektywność programów prologowych

Zasadnicze aspekty efektywności:

- Brak zgodności między architekturą komputera a sposobem przetwarzania realizowanym przez mechanizm wnioskowania w Prologu, skutkiem czego szybciej napotykamy na ograniczenia czasowe lub pamięciowe
- Jednak deklaratywny charakter programowania często skraca w stopniu znaczącym czas potrzebny na napisanie programu
- Prostsza implementacja algorytmów opartych na przetwarzaniu symbolicznym i strukturalnych formach reprezentacji danych
- Mniej efektywna w Prologu implementacja algorytmów przetwarzania numerycznego

Efektywność programów prologowych

Ogóle metody poprawy efektywności:

- Wybór, o ile to możliwe, kompilacji a nie interpretacji programu prologowego
- Zmiana interpretacji proceduralnej programu prologowego: szukanie lepszego porządku klauzul, innej kolejności podcelów, zastosowanie odcięć itp.
- Zmiana sposobu poszukiwania rozwiązania:
  - unikanie niepotrzebnych nawrotów,
  - unikanie analizy nadmiarowych alternatywnych ścieżek wnioskowania
- Dobór lepszych z punktu widzenia efektywności form reprezentacji danych
- Zastosowanie mechanizmów przechowywania wyników pośrednich

Efektywność programów prologowych

Przykład analizy efektywności programu Prologowego:

Problem kolorowania mapy

Każdemu z krajów trzeba przypisać jeden z czterech kolorów tak, aby żadne dwa sąsiadujące ze sobą kraje nie miały tego samego koloru. Istnieje twierdzenie, które gwarantuje, że zawsze jest to możliwe.

Opis mapy:

Predykat **`ngb(Country,Neighbours)`** np.:

```
ngb(andorra,[france,spain]).
ngb(lithuania,[poland,russia,latvia,belarus]).
ngb(latvia,[lithuania,belarus,russia,estonia]).
...
```

Efektywność programów prologowych

Przykład analizy efektywności programu Prologowego:

Problem kolorowania mapy c.d.

Opis wyników:
Lista termów postaci **`Country/Colour`** np.:
`[albania/C1, andorra/C2, austria/C3,...].`
gdzie **`C1`**, **`C2`**, **`C3`** to kolory, które trzeba przypisać krajom.

Rozwiązanie:
Predykat **`colours(Country_colour_list)`** jest prawdziwy, o ile lista **`Country_colour_list`** spełnia ograniczenia narzucone przez relację **`ngb`** i treść zadania (m.in. cztery dostępne kolory: **`yellow`**, **`blue`**, **`red`**, **`green`**).

Efektywność programów prologowych

Przykład analizy efektywności programu Prologowego:

Problem kolorowania mapy c.d.

Sformułowanie rozwiązania w Prologu:

```
colours([]).
colours([Cn/Cl|T]):- colours(T),
 member(Cl,[yellow,red,green,blue]),
 not(member(Cn1/Cl,T),neighbour(Cn,Cn1)).

neighbour(Cn,Cn1):-
 ngb(Cn,Nghbs),member(Cn1,Nghbs).
```

Efektywność programów prologowych

Przykład analizy efektywności programu Prologowego:

Problem kolorowania mapy c.d.

Pomocniczy predykat: **`country(C):- ngb(C,_).`**
Wywołanie :

```
?- setof(Cn/Cl,country(Cn),CnClList),
   colours(CnClList).
```

Przedstawione rozwiązanie jest bardzo nieefektywne, gdyż silnie zależy od kolejności krajów na liście **`CnClList`** , zaś ta jest alfabetyczna (**`setof`**!) i nie ma nic wspólnego z geograficznym położeniem krajów, co spowoduje dużą liczbę nawrotów w trakcie poszukiwania rozwiązania.

Efektywność programów prologowych

Przykład analizy efektywności programu Prologowego:

Problem kolorowania mapy c.d.

1-sza propozycja poprawy efektywności:

W czasie przypisywania kolorów, kraje powinny być analizowane w kolejności od kraju z największa liczbą sąsiadów do kraju z najmniejszą liczbą sąsiadów, liczba nawrotów będzie wtedy dużo mniejsza. Odpowiednią listę krajów można utworzyć ręcznie albo za pomocą dodatkowego predykatu sortującego.

Efektywność programów prologowych

Przykład analizy efektywności programu Prologowego:

Problem kolorowania mapy c.d.

2-ga propozycja:
porządkowanie zgodnie z sąsiedztwem

```
makelist(L):-
  collect([germany],[],L).
collect([],Closed,Closed).
collect([X|Open],Closed,L):-
  member(X,Closed),!,
  collect(Open,Closed,L).
collect([X|Open],Closed,L):-
  ngb(X,Nghbs),
  conc(Nghbs,Open,Open1),
  collect(Open1,[X|Closed],L).
```

Predykat **makelist** rozpoczyna porządkowanie od zadanego z góry kraju i tworzy listę **Closed**. Każdy kraj najpierw umieszczany jest na innej liście, nazwanej **Open**, i dopiero potem przenoszony na listę **Closed**. Przy przenoszeniu jego sąsiedzi dopisywani są do listy **Open**.

Efektywność programów prologowych

Przykład analizy efektywności programu Prologowego:

Problem kolorowania mapy c.d.

Wywołanie :

```
?- makelist(CnList),
   bagof(Cn/Cl,member(Cn,CnList),CnClList),
   colours(CnClList).
```

Przedstawione rozwiązanie jest znacznie efektywniejsze od poprzedniego, gdyż kolejności krajów na liście **CnClList** nie jest alfabetyczna, lecz zależy od geograficznego sąsiedztwa krajów, co powoduje wyraźny spadek liczby nawrotów w trakcie poszukiwania rozwiązania.

Efektywność programów prologowych

Poprawa efektywności poprzez zmianę struktury danych:

Modyfikacja operacji konkatenacji

Dotychczasowa definicja:

```
conc([],L,L).
conc([X|L1],L2,[X|L3]):- conc(L1,L2,L3).
```

Nieefektywna w sytuacji, gdy pierwsza lista **L1** jest długa.
Zapytanie: **?- conc([a,b,c],[d,e],L).**
prowadzi do następującej sekwencji wywłań:

```
conc([a,b,c],[d,e],L)
 conc([b,c],[d,e],L')     gdzie: L   = [a|L']
  conc([c],[d,e],L'')     gdzie: L'  = [b|L'']
   conc([],[d,e],L''')    gdzie: L'' = [c|L''']
    True                  gdzie: L'''= [d,e]
```

Program analizuje całą pierwszą listę, tak długo, aż nie dojdzie do końca i dopiero wtedy następuję faktyczne łączenie list.

Efektywność programów prologowych

Poprawa efektywności poprzez zmianę struktury danych:

Modyfikacja operacji konkatenacji c.d.

Obserwacja:
Zamiast analizować pierwszą listę, lepiej „przeskoczyć" od razu na jej koniec i tam dołączyć druga listę. W standardowej reprezentacji list problemem jest jednak znalezienie końca pierwszej listy.
W tym celu potrzebna nowa reprezentacja tzw. ***listy różnicowe***:

Lista **L** będzie reprezentowana jako różnica list **L1-L2**, przy czym **L2** jest dowolną (najczęściej niepustą) końcówką listy **L1**.

Przykład starej i nowej reprezentacji list:

```
L= [a,b,c]        L1-L2 = [a,b,c]-[]
L= [a,b,c]        L1-L2 = [a,b,c,d,e|T]-[d,e|T]
L= [a,b,c]        L1-L2 = [a,b,c|T]-T
```

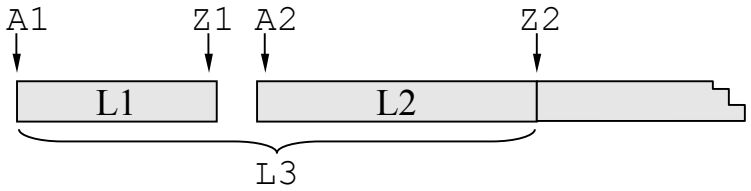
Lista pusta **[]** reprezentowana będzie teraz przez różnicę **L-L** .

Efektywność programów prologowych

Poprawa efektywności poprzez zmianę struktury danych:

Modyfikacja operacji konkatenacji c.d.

Nowa definicja: **`concat(A1-Z1,Z1-Z2,A1-Z2).`**



Lista **`L1`** reprezentowana jest przez różnicę **`A1-Z1`** .
Lista **`L2`** reprezentowana jest przez różnicę **`A2-Z2`** .
Wynik **`L3`** reprezentowany jest przez różnicę **`A1-Z2`**, o ile **`Z1=A2`**.

Efektywność programów prologowych

Poprawa efektywności poprzez zmianę struktury danych:

Modyfikacja operacji konkatenacji c.d.

Zastosowanie nowej operacji:

Konkatenacja list **`[a,b,c]`** i **`[d,e]`**:

```
?- concat([a,b,c|T1]-T1,[d,e|T2]-T2,L).
T1= [d,e|T2]
L= [a,b,c,d,e|T2]-T2
```

Zastosowanie list różnicowych powoduje znaczny wzrost efektywności większości zredefiniowanych predykatów przetwarzania list. Nowe formy tych predykatów nie są jednak tak uniwersalne w zastosowaniu jak definicje pierwotne.

Efektywność programów prologowych

Wzrost efektywności poprzez zapamiętywanie wyników pośrednich

W trakcie przetwarzania niektóre cele spełniane są wielokrotnie. Prolog pozbawiony jest jednak mechanizmów wykrywania powtarzających się obliczeń.

Przykład

Obliczanie n-tego wyrazu ciągu Fibonacciego:

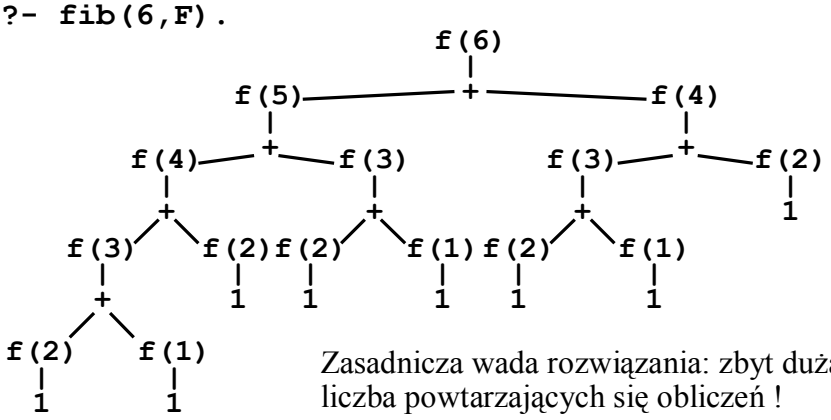
```
fib(1,1).              % pierwszy wyraz
fib(2,1).              % drugi wyraz
fib(N,F):-             % N-ty wyraz (N>2)
 N>2,
 N1 is N-1, fib(N1,F1),
 N2 is N-2, fib(N2,F2),
 F is F1+F2.           % suma dwóch poprzednich
```

Efektywność programów prologowych

Wzrost efektywności poprzez zapamiętywanie wyników pośrednich

Przykład realizacji zapytania

```
?- fib(6,F).
```



Zasadnicza wada rozwiązania: zbyt duża liczba powtarzających się obliczeń !

Efektywność programów prologowych

Wzrost efektywności poprzez zapamiętywanie wyników pośrednich

Przechowywanie pośrednich wyników obliczeń za pomocą predykatu systemowego **asserta** pozwala zredukować czas obliczeń kosztem wzrostu wymagań pamięciowych.

Zmodyfikowana wersja

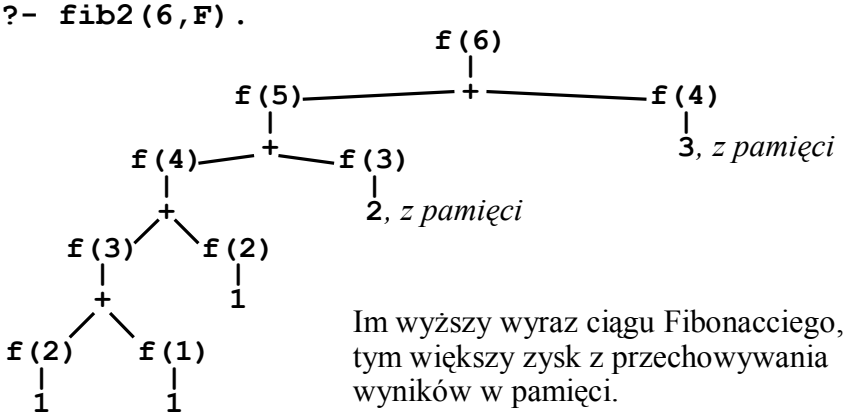
```
fib2(1,1).              % pierwszy wyraz
fib2(2,1).              % drugi wyraz
fib2(N,F):-             % N-ty wyraz (N>2)
 N>2,
 N1 is N-1, fib2(N1,F1),
 N2 is N-2, fib2(N2,F2),
 F is F1+F2,            % suma dwóch poprzednich
 asserta(fib2(N,F)).    % na początek pamięci!
```

Efektywność programów prologowych

Wzrost efektywności poprzez zapamiętywanie wyników pośrednich

Przykład realizacji zapytania

```
?- fib2(6,F).
```



Im wyższy wyraz ciągu Fibonacciego, tym większy zysk z przechowywania wyników w pamięci.

Efektywność programów prologowych

Wzrost efektywności poprzez zapamiętywanie wyników pośrednich

Przechowywanie pośrednich wyników obliczeń za pomocą dodatkowego parametru (tzw. *akumulatora*) zmienia charakter obliczeń na "iteracyjne", co powoduje znaczny wzrost ich wydajności.

Zmodyfikowana wersja z akumulatorem

```
fib3(N,F):- forwardfib(2,N,1,1,F).  % podcel

forwardfib(M,N,F1,F2,F2):- M >= N.  % koniec
forwardfib(M,N,F1,F2,F):-    % M-ty wyraz to F2
 M < N,          % N-tego wyrazu jeszcze nie ma
 NextM is M+1,
 NextF2 is F1+F2,      % suma dwóch poprzednich
 forwardfib(NextM,N,F2,NextF2,F). %
```

Efektywność programów prologowych

Podsumowanie

Proste sposoby poprawy efektywności:
- zmiana porządku klauzul, podcelów, zastosowanie odcięć itp.
- manipulowanie przebiegiem wnioskowania w celu unikania niepotrzebnych nawrotów i/lub niepotrzebnych (nadmiarowych) ścieżek wnioskowania

Zaawansowane sposoby poprawy efektywności:
- zmiana formy reprezentacji danych
- przechowywanie wyników pośrednich obliczeń
- opracowanie lepszych algorytmów