Programowanie deklaratywne

Artur Michalski
Informatyka II rok

Plan wykładów

- 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 technika programowania w Prologu

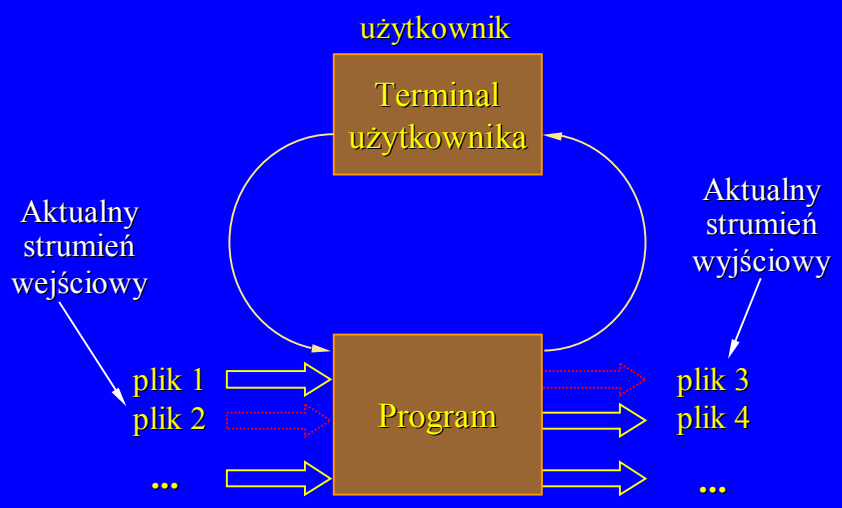
Operacje wejścia/wyjścia w Prologu

- Operacje na plikach sekwencyjnych
- Przetwarzanie plików termów
- Manipulowanie danymi znakowymi
- Kompozycja i dekompozycja atomów
- Wczytywanie programów prologowych: *consult* i *reconsult*

Operacje na plikach sekwencyjnych

- W języku Prolog operacje na plikach opierają się na koncepcji *strumienia*
- Program prologowy czyta dane ze *strumienia wejściowego* i zapisuje dane do *strumienia wyjściowego*
- Strumieniem wejściowym i/lub wyjściowym może być dowolny plik o dostępie sekwencyjnym (tekstowy)
- Terminal użytkownika jest również traktowany jak strumień
- W trakcie wykonywania programu w danej chwili realizowana może być operacja odczytu i zapisu odpowiednio z/do jednego strumienia wejściowego i jednego strumienia wyjściowego
- *Domyślnym strumieniem* wejściowym i wyjściowym jest terminal użytkownika

Operacje na plikach sekwencyjnych



Operacje na plikach sekwencyjnych

Otwieranie plików

Operacja zmiany *aktualnego strumienia wejściowego*

```
see(<nazwa_pliku>)
```

jeżeli plik jest już otwarty to nadal pozostanie w trybie odczytu (nie będzie błędu!)

Operacja zmiany *aktualnego strumienia wyjściowego*

```
tell(<nazwa_pliku>)
```

jeżeli plik jest już otwarty to nadal pozostanie w trybie zapisu (nie będzie błędu!)

Operacje na plikach sekwencyjnych

Zamykanie plików

Operacja zamknięcia *aktualnego strumienia wejściowego*

```
seen
```

Predykat ten jest zawsze spełniony. Po wykonaniu strumieniem wejściowym zostaje terminal.

Operacja zamknięcia *aktualnego strumienia wyjściowego*

```
told
```

Predykat ten jest zawsze spełniony. Po wykonaniu strumieniem wyjściowym zostaje terminal.

Operacje na plikach sekwencyjnych

Identyfikacja strumieni

Operacja identyfikacji *aktualnego strumienia wejściowego*

```
seeing(Str)
```

Zmienna `Str` jest unifikowana z identyfikatorem strumienia (wygenerowanym automatycznie przez system).

Operacja identyfikacji *aktualnego strumienia wyjściowego*

```
telling(Str)
```

Zmienna `Str` jest unifikowana z identyfikatorem strumienia (wygenerowanym automatycznie przez system).

Operacje na plikach sekwencyjnych

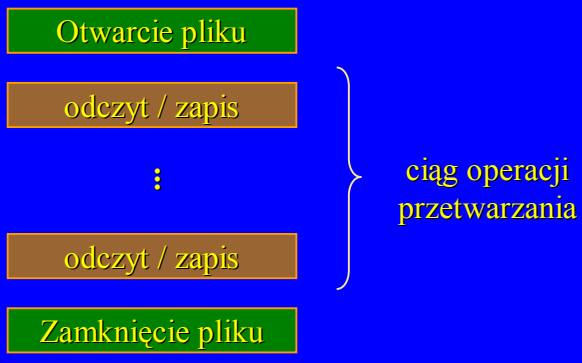
Operacje otwarcia i zamknięcia (`see`, `tell`, `seen`, `told`) służą wyłącznie do przetwarzania plików tekstowych (sekwencyjnych).

Podczas przetwarzania plików sekwencyjnych każda operacja (zapis/odczyt) powoduje automatyczne przejście do następnej pozycji w pliku.

Osiągnięcie końca pliku sygnalizowane jest zawsze specjalną wartością.

Operacje na plikach sekwencyjnych

Ogólnych schemat przetwarzania pliku



Sekwencyjny charakter przetwarzania plików jest niezgodny z deklaratywną naturą programowania za pomocą reguł.

Operacje na plikach sekwencyjnych

Rodzaje plików tekstowych:

- Pliki znakowe:
  plik składa się z pojedynczych bajtów/znaków
- Pliki termów:
  podstawowym składnikiem pliku jest term

Operacje na plikach sekwencyjnych

Operacje odczytu i zapisu dla plików znakowych:

`get(X)` - odczyt pojedynczego i *nie białego znaku* z aktualnego strumienia

`get_byte(X)` - odczyt jednego, dowolnego bajtu z aktualnego strumienia

`put(X)` - zapis pojedynczego znaku do aktualnego strumienia

Osiągnięcie końca pliku sygnalizowane jest wartością `-1`.

Operacje na plikach sekwencyjnych

Operacje odczytu i zapisu dla plików termów:

`read(X)` - odczyt pojedynczego termu z aktualnego strumienia wejściowego

`write(X)` - zapis pojedynczego termu do aktualnego strumienia wyjściowego

Osiągnięcie końca pliku sygnalizowane jest predefiniowanym atomem **`end_of_file`**.

Pliki termów muszą spełniać wymogi składni termów języka Prolog.

Przetwarzanie plików termów

Operacja odczytu termu ze strumienia wejściowego

Predykat **`read(X)`** oznacza odczyt pojedynczego termu z aktualnego strumienia wejściowego i unifikację termu ze zmienną **`X`**.

Brak dopasowania dla argumentu **`X`** predykatu **`read`**, nie będącego zmienną wolną, doprowadzi do błędu (nie nastąpi nawrót w celu odczytania następnego termu!).

Termy zawarte w pliku wejściowym muszą być zakończone znakiem kropki.

Pełna wersja operacji odczytu pozwala również wskazywać jakiego strumienia (**`Str`**) dotyczy operacja: **`read(Str,X)`**.

Przetwarzanie plików termów

Operacja zapisu termu do strumienia wyjściowego

Predykat `write(X)` oznacza zapis pojedynczego termu `X` do aktualnego strumienia wyjściowego.

Term `X` zostanie zapisany do pliku skojarzonego ze strumieniem wyjściowym w formie identycznej z wykorzystywaną dla domyślnego strumienia wyjściowego (terminala).

Termy zapisywane do pliku mogą mieć dowolny stopień złożoności (zagnieżdżenia).

Pełna wersja operacji zapisu pozwala również wskazywać jakiego strumienia (`Str`) dotyczy operacja: `write(Str,X)`.

Przetwarzanie plików termów

Pozostałe operacje dla strumienia wyjściowego

Predykat `append(X)` otwiera plik `X` w trybie dopisywania i wiąże go z aktualnym strumieniem wyjściowym.

Predykat `tab(N)` oznacza zapis `N` znaków odstępu do aktualnego strumienia wyjściowego, przy czym `N` musi być większe od 0.

Predykat `nl` (bezargumentowy) powoduje zapis znaku nowego wiersza (przejścia do nowej linii) do pliku skojarzonego z aktualnym strumieniem wyjściowym.

Przetwarzanie plików termów

Typowy schemat przetwarzania plików termów

Odczyt:

```
...,
seeing(Old),
see('dane.txt'),
...,
read(X),
...,
seen,
see(Old),
...,
```

Zapis:

```
...,
telling(Old),
tell('wyniki.txt'),
 ...,
write(X),
 ...,
told,
tell(Old),
...,
```

Przetwarzanie plików termów

Przykład przetwarzania plików tekstowych – odczyt z terminala

Interaktywne obliczanie sześcianu liczb dla ciągu wartości wczytanych z terminala od użytkownika:

```
cube:- read(X), process(X).
process(stop):- !.
process(X):- N is X*X*X, write(N), cube.
```

```
?- cube.
|:2.
8
|:5.
125
|:stop.
Yes
```

Przykład użycia

Przetwarzanie plików termów

Przykład przetwarzania plików tekstowych c.d. – odczyt z terminala

Wersja pierwotna:

```
cube:- read(X), process(X).
process(stop):- !.
process(X):- N is X*X*X, write(N), cube.
```

Skrócona wersja eksperymentalna:

```
cube:- read(stop),!.
cube:- read(X), N is X*X*X, write(N), cube.
```

Przetwarzanie plików termów

Przykład przetwarzania plików tekstowych c.d. – odczyt z terminala

Skrócona wersja eksperymentalna:

```
cube:- read(stop),!.
cube:- read(X), N is X*X*X, write(N), cube.
```

Wyniki:

```
?- cube.
|:2.
```

Dane są tracone, bo brak unifikacji dla `read(stop)`!

```
|:stop.
```

Po unifikacji `read(X)` błąd typu w `N is X*X*X` dla stałej `stop`

```
ERROR: Arithmetic:'stop/0' is not a function
```

Przetwarzanie plików termów

Przykład przetwarzania plików tekstowych c.d. – odczyt z terminala

Wersja „user-friendly”:

```
cube:-  write('Podaj liczbe: '),
        read(X),process(X).
process(stop):- !.
process(X):- N is X*X*X,
             write('Sześcian '),write(X),
             write(' wynosi '), write(N),nl,
             cube.
```

```
?- cube.
Podaj liczbe: 2.
Sześcian 2 wynosi 8
Podaj liczbe: stop.
Yes
```

Przykład użycia

Przetwarzanie plików termów

Przykład przetwarzania struktur rekurencyjnych - listy

Klauzula `writelist(L)` wyświetla listę `L` na ekranie w taki sposób, że każdy term pojawia się w kolejnym wierszu:

```
writelist([]).
writelist([H|T]):- write(H), nl,
                   writelist(T).
```

Wczytywanie danych z terminala na listę :

```
readlist(L):- read(X), process(X,L).

process(stop,[]):- !.
process(X,[X|T]):- readlist(T).
```

Przetwarzanie plików termów

Przykład przetwarzania struktur rekurencyjnych - listy

Jeżeli składowymi listy będą ***wyłącznie*** listy, to wszystkie elementy jednej podlisty powinny być w jednym wierszu:

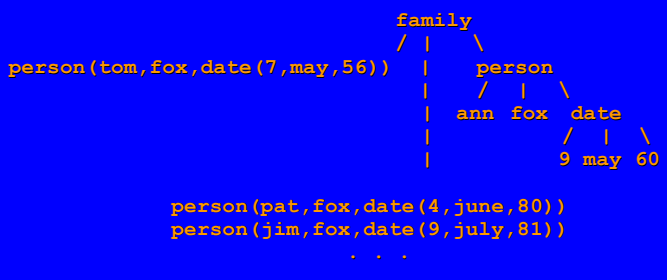
```
writelist2([]).
writelist2([H|L]):-
                  doline(H),nl,
                  writelist2(L).

doline([]).
doline([H|L]):- write(H),tab(1),doline(L).
```

Zagnieżdżanie rekurencyjnych struktur wymaga zdefiniowania oddzielnego predykatu przetwarzania na każdym poziomie złożonej struktury.

Przetwarzanie plików termów

Operacje we/wy na złożonych strukturach danych



Złożone (zagnieżdżone) termy mogą być nieczytelne!

```
family(person(tom,fox,date(7,may,56)),
       person(ann,fox,date(9,may,60)),
      [person(pat,fox,date(4,june,80)),
       person(jim,fox,date(9,july,81))]).
```

Przetwarzanie plików termów

Operacje we/wy na złożonych strukturach danych c.d.

```
family(person(tom,fox,date(7,may,56)),
       person(ann,fox,date(9,may,60)),
      [person(pat,fox,date(4,june,80)),
       person(jim,fox,date(9,july,81))]).
```

Klauzula `wfamily(F)` formatuje dane z termu `F` w czytelny sposób.

```
wfamily(family(H,W,Ch)):-
  nl,write(parents),nl,nl,
  writeperson(H),nl,
  writeperson(W),nl,nl,
  write(children),nl,nl,
  writepersonlist(Ch).
```

Przetwarzanie plików termów

Operacje we/wy na złożonych strukturach danych c.d.

```
writeperson(person(N,SN,date(D,M,Y))):-
  tab(4),write(N), tab(1),write(SN),
  write(',born '),
  write(D),tab(1),
  write(M),tab(1),
  write(Y).

writepersonlist([]).
writepersonlist([H|T]):-
  writeperson(H),nl,
  writepersonlist(T).
```

Definicje klauzul służących do przetwarzania złożonych danych strukturalnych powinny odzwierciedlać ich strukturę.

Przetwarzanie plików termów

Operacje we/wy na złożonych strukturach danych c.d.

Rezultat działania klauzuli `wfamily(F)`:

```
parents

   tom fox born, 7 may 56
   ann fox born, 9 may 60

children

   pat fox born, 4 june 80
   jim fox born, 9 july 81
```

Przetwarzanie plików termów

Przetwarzanie pliku termów z niestandardowego strumienia

Typowy schemat przetwarzania plików dyskowych:

```
start:- see('dane.txt'), procfile, seen.
procfile:- read(Term), process(Term).
process(end_of_file):- !.
process(Term):- treat(Term),procfile.
```

Przykład przetwarzania pliku dyskowego - wyświetlanie zawartości pliku z numerowaniem termów

```
showfile(N):- read(Term), show(Term,N).
show(end_of_file,_):- !.
show(Term,N):- write(N),tab(2),write(Term),nl,
               N1 is N+1, showfile(N1).
```

Przetwarzanie plików termów

Przykład przetwarzania termów z niestandardowego strumienia wejściowego i wyjściowego

Katalog towarów zawarty jest w pliku składającym się z termów postaci:

```
item(numer,opis,cena,dostawca)
```

Chcemy wygenerować nowy plik zwierający tylko towary od jednego wyznaczonego dostawcy `Sup`. Przetwarzanie wymaga odczytu z jednego pliku np. `dane` i zapisu do drugiego pliku np. `wyniki`.

Zapytanie celu głównego:

```
?- see('dane'),tell('wyniki'),Sup=lloyd,
makefile(Sup),told,seen.
```

Przetwarzanie plików termów

Przykład przetwarzania termów z niestandardowego strumienia wejściowego i wyjściowego

Klauzula `makefile(Sup)` wymaga podania nazwy dostawcy:

```
makefile(Sup):- write(Sup),write('.'),
                nl,makerest(Sup).
makerest(Sup):- read(Item),proc(Item,Sup).
proc(end_of_file,_):- !.
proc(item(N,D,P,Sup),Sup):- !,
  write(item(N,D,P)),write('.'),nl,
  makerest(Sup).
proc(_,Sup):- makerest(Sup).
```

Znak kropki jest dopisywany na końcu każdego termu w pliku wynikowym w celu umożliwienia ich ewentualnego odczytu za pomocą predykatu `read`.

Manipulowanie danymi znakowymi

Operacje odczytu i zapisu znaku z/do strumienia wyjściowego

Predykat **`get_byte(X)`** oznacza odczyt pojedynczego znaku z aktualnego strumienia wejściowego i unifikację jego kodu ASCII ze zmienną **`X`**.

Predykat **`get(X)`** jest wariantem klauzuli **`get`** przeznaczonym do odczytu znaków kodu ASCII nie będących tzw. białymi znakami – wszystkie wiodące białe znaki są pomijane, aż do napotkania widocznego znaku.

Predykat **`put(X)`** oznacza zapis pojedynczego kodu ASCII znaku **`X`** do aktualnego strumienia wyjściowego.

Manipulowanie danymi znakowymi

Przykład przetwarzania plików znakowych

Usuwanie nadmiarowych znaków odstępu z wczytanego napisu wejściowego - predykat **`squeeze(X)`**. Napis musi być zakończony znakiem kropki.

Przykład zastosowania

```
?- squeeze.
: Robot  wylał    wodę na  parapet.
Robot wylał wodę na parapet
```

Definicja:

```
squeeze:- get_byte(C),put(C),do_rest(C).
do_rest(46):- !. %znak kropki-koniec
do_rest(32):- !,get(C),put(C),do_rest(C).
do_rest(C):- squeeze.
```

Kompozycja i dekompozycja atomów

Predykat systemowy `name(A,S)` służy do przekształcania wczytanej informacji reprezentowanej w postaci listy kodów znaków (`S`) w stałą symboliczną lub liczbę (`A`). Konwersji można dokonywać w obie strony.

Przykłady

```
?- name(kot,X).
X=[107,111,116]
?- name(Y,[112,105,101,115])
Y=pies
```

Alternatywna reprezentacja list kodów znaków w Prologu - napis ograniczony znakami cudzysłowu

```
?- [112,105,101,115]="pies".
Yes
?- name(Y,"pies").
Y=pies
```

Kompozycja i dekompozycja atomów

Zastosowanie predykatu `name`

Przetwarzanie identyfikatorów numerowanych np. postaci:

`item1,item2,...,itemk,...`

Predykat `isitem(X)` ma służyć do sprawdzania, czy mamy do czynienia z właściwym identyfikatorem:

```
isitem(X):- name(X,Xlist),
            name(item,T),
            conc(T,_,Xlist).

conc([],L,L). % przypomninie definicji
conc([H|T],L,[H|T2]):- conc(T,L,T2).
```

Kompozycja i dekompozycja atomów

Zastosowanie predykatu **name** c.d.:

Przekształcanie napisów języka naturalnego w następującą reprezentację wewnętrzną:
- każdy pojedynczy napis jest atomem
- całe zdanie jest listą atomów

Zdanie w języku naturalnym – przykładowe dane wejściowe:

Tomek był zadowolony z postępów robota.

Rezultat – przykładowe dane wyjściowe:

```
[Tomek,był,zadowolony,z,postępów,robota]
```

Kompozycja i dekompozycja atomów

Zastosowanie predykatu **name** c.d.:

Program:

```
getsntc(Wlist):-
 get_byte(C),getrest(C,Wlist).

getrest(46,[]):- !. %znak końcowy - kropka
getrest(32,Wlist):- !,getsntc(Wlist).
getrest(L,[W|Wlist]):- getlttrs(L,Ls,Next),
  name(W,Ls),getrest(Next,Wlist).

getlttrs(46,[],46):- !.
getlttrs(32,[],32):- !.
getlttrs(L,[L|Ls],Next):- get_byte(C),
  getlttrs(C,Ls,Next).
```

Kompozycja i dekompozycja atomów

Zastosowanie predykatu **name** c.d.:

Komentarz

W predykacie **getrest** rozważamy trzy przypadki:
L - jest znakiem końca (kropką)
L - jest znakiem białym: pomijamy go i **getsntc** dla reszty
L - jest literą: najpierw odczyt całego słowa **W**, rozpoczynającego się od **L**, potem reszta zdania **Wlist** za pomocą **getsntc**, a wynik skumulowany w **[W|Wlist]**

```
getrest(46,[]):- !. %znak końcowy - kropka
getrest(32,Wlist):- !,getsntc(Wlist).
getrest(L,[W|Wlist]):- getlttrs(L,Ls,Next),
  name(W,Ls),getrest(Next,Wlist).
```

Kompozycja i dekompozycja atomów

Zastosowanie predykatu **name** c.d.:

Komentarz

W predykacie **getlttrs(L,Ls,Nc)** argumenty oznaczają:
L - aktualny znak (już wczytany) czytanego słowa
Ls - lista znaków (zaczynająca się od **L**) do końca słowa
Nc - znak, który znajduje się bezpośrednio za wczytanym słowem; musi być znakiem białym

```
getlttrs(46,[],46):- !.
getlttrs(32,[],32):- !.
getlttrs(L,[L|Ls],Next):- get_byte(C),
  getlttrs(C,Ls,Next).
```

Wczytywanie programów prologowych: *consult* i *reconsult*

W języku Prolog wczytywanie plików tekstowych zawierających program odbywa się na dwa sposoby:

- predykat `consult(F)` ładuje wszystkie klauzule z pliku `F`, które następnie są wykorzystywane do osiągnięcia zadanego celu; ponowne wykonanie tego predykatu spowoduje dopisanie nowych klauzul do już istniejących
- predykat `reconsult(F)` działa podobnie; jedynie w przypadku wystąpienia w pliku `F` klauzul relacji zawartych już w pamięci nastąpi ich redefinicja; pozostałe klauzule pozostaną nie zmienione