Programowanie w logice i funkcyjne (Prolog), wyklady 5 i 6 (prezentacja niekompletna i przed korektą)

Konrad Zdanowski

Outline

- Prolog zawiera wiele wbudowanych predykatów.
- Predykaty rozróżniane są przez ich nazwę oraz liczbę argumentów.
- Liczbę argumentów możemy oznaczyć przez $/n$ po nazwie predykatu.
- Argumenty predykatu w Prologu są oznaczane w zależności od tego, czy powinny być określone czy też nie.
- Przykłady:
  - sort(+List,-Sorted)
  - between(+Low,+High,?Value).

Oznaczenia argumentów predykatów

- `++` – argument nie może zawierać zmiennych (ugruntowany),
- `+` – podstawienie argumentu musi, że argument spełnia żądany typ,
- `-` – argument pełni rolę argumentu wyjściowego, może być zmienną ale może też być (częściowo) wyspecyfikowany,
- `--` – argument musi być zmienną,
- `?` – argument powinien być wartościowany do odpowiedniego typu (zmienna jest dowolnego typu),
- `@` – argument nie będzie dokładniej wartościowany.

Przykłady predykatów

- `between(+Low, +High, ?Value),`
- `length(?List, ?Int).`
  - `length(X,N)` zwróci wiele odpowiedzi.
  - `length(f(X),N)` zwróci błąd.

Outline

Arytmetyka w Prologu

- Wszystko jest termem.
- ```
  ?- 2+2 == 4.
  false.
  ?- 2+2 = 4.
  false.
  ```

- Baza wiedzy

  ```
  ladna(3).
  ladna(X+Y).
  ```

  Wtedy

  ```
  ?- ladna(3).
  true.
  ?- ladna(5+2).
  true.
  ?- ladna(4-1).
  false.
  ```
  ```

Arytmetyka w Prologu

- Arytmetyka wymaga oddzielnego zestawu operatorów, żeby Prolog wiedział, że ma potraktować termy jak zapis liczb.
- Operatory te wykonują ewaluację termów arytmetycznych.
- `=:=, =\=, <, >, =<, >=`
- Aby Prolog mógł obliczyć `t =:= s` termy `t` oraz `s` muszą posiadać wartość przy aktualnie skonstruowanym wartościowaniu.

- Predytaktów arytmetycznych nie można zmienić.

```
?- assert(2>4).
ERROR: No permission to modify static procedure `(>)/2'
ERROR: In:
ERROR:     [8] assert(2>4)
ERROR:     [7] <user>
```

- Aby „przypisać” zmiennej wartość arytmetyczną (czyli zunifikować wartość arytmetyczną ze zmienną) używamy

```
N is term.
```

- Uzgodnienie `N is term` polega na przypisaniu N wartości arytmetycznej wyrażenia `term`. Wartość ta musi istnieć.
- Inne operatory arytmetyczne w Prologu to m.in.
  - `+, -, *.`
  - `X / Y` – dzielenie wymierne,
  - `X // Y` – dzielenie całkowito liczbowe,
  - `X mod Y`,
  - `sgn/1, sqrt/1, ...`

- W wyrażeniu `N is term` po lewej stronie występuje zmienna lub stała, po prawej wyrażenie arytmetyczne, które można wyliczyć.
- W `term` mogą występować zmienne ale muszą mieć przypisaną wartość arytmetyczną.
- Wartość termu `term` jest wyliczana a następnie jest unifikowana z N.

- W Prologu możemy podstawić bardziej złożony term za zmienną, pod którą już coś podstawiliśmy

```
?- X=f(Z), X=f(h(w,W)).
X = f(h(w, W)),
Z = h(w, W).
```

- Wartości arytmetyczne to stałe, więc nie możemy zmienić wartości zmiennej, która posiada już wartość numeryczną.

```
?- X is 4, X is X+1.
false.
```

- Aby używać zmienną o nowej wartości, trzebą wprowadzic nową zmienną

```
?- X is 4, Y is X+1.
X = 4,
Y = 5.
```

Przykłady użycia `is`

- ```
  ?- Y is 2+X.
  ERROR: Arguments are not sufficiently instantiated
  ERROR: In:
  ERROR:     [8] _13204 is 2+_13212
  ERROR:     [7] <user>
  ?- X=3, Y is 2+X.
  X = 3,
  Y = 5.
  ```
- ```
  ?- 2+3 is 2+3.
  false.

  ?- 2+X = 2+3.
  X = 3.
  ```
- ```
  ?- 5 is 2+3.
  true.

  ?- 5 = 2+3.
  false.
  ```
  ```

Przykłady z ’=:=’

- ’=:=’ oblicza wartość obu argumentów i sprawdza czy są równe.
- ```
  ?- 1 =:= sin(pi/2).
  true.

  ?- 1 is sin(pi/2).
  false.

  ?- 1.0 is sin(pi/2).
  true.

  ?- 1 = 1.0.
  false.
  ```
  ```

A pętlę w Prologu robimy tak ...

- Nieskończona pętla:

```
unbdloop(L,L).
unbdloop(L,X):-unbdloop(L,Z), X is Z+1.

?- unbdloop(4,X).
X = 4 ;
X = 5 ;
X = 6 ;
X = 7 ;
X = 8 ;
...
```

- `unbdloop(L,X):-unbdloop(L,X-1).` nie zadziała bo 'X-1' to term a nie liczba.

A pętlę w Prologu robimy tak ...

- Nieskończona pętla:
- 'repeat' jest atomem, który odnosi sukces dowolnie wiele razy,
- możemy go użyć do stworzenia pętli nieskończonej,
- Nie próbujmy tego w domu:

```
?- repeat, write('Jestem w pętli'), fail.
```

Pętlę w Prologu robimy tak ...

- Pętla ograniczona

```
bdloop(L,H):- L=<H,
              unbdloop(L,X),
              write(X),nl,
              X>=H,!.

?- bdloop(4,6).
4
5
6
true.
```

Pętlę w Prologu robimy tak ...

- Prolog posiada wbudowany predykat

  `between(+Low, +High, ?Value),`

  który uzgadnia `Value` z kolejnymi liczbami pomiędzy `Low` i `High`.
- `High` może mieć wartość `inf` lub `infinite`.
- Daje to nam łatwy sposób stworzenia pętli

```
loop(L,H) :- between(L,H,X), write(X), nl, fail.
loop(L,H).

?-loop(3,5).
3
4
5
true.
```

Outline

Wypisywanie informacji

- 'write/1' wypisuje wartość argumentu (term) na aktualny strumień wyjściowy,
- 'nl' – wypisuje nową linię,
- Jeśli nazwa atomu zaczyna się dużą literą lub zawiera spacje, potrzebujemy apostrofów.

```
?- write(Jan).
_16048
true.

?- write('Jan').
Jan
true.

?- write('Jan').
Jan
true.

?- write(jan kowalski).
ERROR: Syntax error: Operator expected
ERROR: write(jan
```

Wczytywanie informacji

- ’read/1’ oczytuje informacje z aktualnego wejścia,
- ’read(X)’ uzgadania X z odczytanym wejściem,
- aby X był poprawnym termem ciąg znaków powinien kończyć się kropką (która nie jest wczytywana) i spacją lub ’enter’,

Pojedyńcze znaki

- ’put(znak)’, put(number)’,
- ’get(X)’ – wczytuje znak pomijając spacje,
- ’get0(X)’ – wczytuje znak (również spacje),
- obie wersje ’get’ unifikują X z wartością ASCII.

Praca z plikami

- Prolog operuje na
  - *current input stream*,
  - *current output stream*.
- Domyślnie są to klawiatura i ekran (nazywane *user*).
- Poza tymi strumieniami można otwierać pliki.
- Jest tylko jeden strumień wejściowy i jeden wejściowy.
- Żaden plik nie może być jednocześnie wejściowy i wyjściowy.

Praca z plikami

- ’tell/1(nazwa pliku)’ ustala strumień wyjściowy,
  - jeśli plik nie istnieje, to jest tworzony,
  - jeśli plik nie jest otwarty, to jego zawartość jest kasowana,
- ’told/0’ zamyka plik, który jest aktualnym strumieniem wyjściowym, tylko taki plik może być zamknięty,
- po wykonaniu ’told’ strumieniem wyjściowym staje się *user*,
- ’telling/1(X)’ unifikuje X z nazwą struminia wyjściowego,
- ’append/1’ tak jak ’tell’ ale dopisuje do pliku.

Praca z plikami

- ’see/1(nazwa pliku)’ ustala strumień wejściowy,
  - jeśli plik nie istnieje, zgłaszany jest błąd,
  - plik nie może być jednocześnie strumieniem wejściowym i wyjściowym,
- ’seen/0’ zamyka plik, który jest aktualnym strumieniem wejściowym, tylko taki plik może być zamknięty,
- po wykonaniu ’seen’ strumieniem wyjściowym staje się *user*,
- ’seeing/1(X)’ unifikuje X z nazwą struminia wejściowego,

Przykład (za Bramer 2013)

```
readline:-get0(X),process(X).
process(10).
process(X):-X=\=10,put(X),nl,readline.

?- readline.
|: wiersz
w
i
e
r
s
z
true
```

Przykład (za Bramer 2013)

Kopiowanie znaków z terminala do pliku wyjściowego aż do przeczytania '!'.

```
copychars(Outfile):- telling(T),tell(Outfile),
copy_characters,told,tell(T).
copy_characters:-get0(N),process(N).
/* 33 is ASCII value of character ! */
process(33).
process(N):-ND\D33,put(N),copy_characters.

?-copychars('plikwyjsciowy.txt').
```

Przykład (za Bramer 2013)

Niech plik people.txt zawiera:

```
jan. kowalski. 34. elektryk.
adam. abacki. 43.  weterynarz.
```

```
 setup:-seeing(S),see('people.txt'),
read_data,
write('Data read'),nl,
seen,see(S).
read_data:-
read(A),process(A).
process(end).
process(A):-
read(B),read(C),read(D),
assertz(person(A,B,C,D)),read_data.
```

Outline

Negacja w Prologu

- W Prologu funkcjonuje założenie o domkniętości świata (closed world assumption, cwa).
- Oznacza ono, że nie jest prawdziwe nic, co nie jest zawarte w bazie wiedzy lub z niej nie wynika.
- Na podstawie cwa, negacja predykatu, $\neg P$, mogłaby być uznana za prawdziwą, jeśli nie uda się jej udowodnić P.
- Ale brak dowodu jest nierozstrzygalny (jest co-rekurencyjnie przeliczalny).

Negacja w Prologu

- Dowód P to skończony świadek na prawdziwość P.
- Brak dowodu P nie można wyrazić za pomocą skończonego obiektu.
- Brak dowodu to informacja, że która mówi, że żaden z nieskończenie (przeliczalnie) wielu obiektów nie jest dowodem P.
- Tylko gdy ograniczymy zbiór możliwych dowodów do skończonego, wiemy, że dowód nie istnieje.

Negacja jako skończona refutacja

- Negacja jako skończona refutacja (negation as failure, naf) to reguła, która mówi, że jeśli Prolog w skończonym wnioskowaniu (obliczeniu) wykluczy możliwość dowodu P, to negacja P kończy się sukcesem.
- Jeśli w bazie wiedzy mamy tylko klauzulę `na(a,b)` to pytanie `?-na(b,a)` ma skończoną refutację.
- Nie istnieją w bazie wiedzy klauzule, których głowa unifikuje się z `na(b,a).`
- `not(na(a,b))` jest potwierdzone przez Prolog.

Negacja jako skończona refutacja

- Niech baza wiedzy zawiera:

```
na(a,b).
na(X,Y):-na(X,Y).
```

- `na(b,a)` wciąż nie jest dowodliwe.
- `not(na(b,a))` jest prawdziwe przy założeniu o domkniętości świata.
- `not(na(a,b))` nie ma skończonej refutacji, bo druga klauzula może być wykorzystana w dowodzie dowolnie wiele razy.
- `not(na(b,a))` nie jest prawdziwe przy interpretacji negacji jako skończonej refutacji.
- Prolog w swoim obliczeniu używa negacji jako skończonej refutacji.

Negacja jako skończona refutacja

- Prolog przestaje być oparty tylko o wnioskowanie w logice.
- Żadna negacja nie wynika ze zbioru formuł Hornowskich.
- Wnioskowanie w Prologu przestaje być poprawne.
- Potrzebujemy innej semantyki.

Negacja jako skończona refutacja

- Jeśli Prolog udowni P(X) to poda podstawienie ($X \mapsto t$) takie, że P(t) też jest dowodliwe, t jest świadkiem na P(X).
- Jeśli Prolog nie udowodni P(X) to znaczy, że nie istnieje t, dla którego Prolog udowodniłby P(t).
- Jeśli Prolog udowodni not(P(X)) to nie konstruuje podstawienia ($X \mapsto t$), bo dowód not(P(X)) to informacja, że P(X) nie ma dowodu.
- Jeśli Prolog nie udowodni not(P(X)) to nie znaczy, że nie uda mu się udowodnić not(P(t)) dla pewnego termu t.

Przykład – negacja

- Niech baza wiedzy to

```
p(a).
```

-
```
:-p(X).
X=a.
:- not(p(X)).
false.
:-not(p(b)).
true.
```

Klauzule uogólnione

- Negacji w Prologu możemy używać w założeniach klauzuli.
- *Klauzula uogólniona* to klauzula postaci

$$P \leftarrow P_1, \ldots, P_n,$$

gdzie P to formuła atomowa a $P_1, \ldots, P_n$ to formuły atomowe lub ich negacje.

Przykład – negacja w uogólnionych klauzulach

- Dodanie negacji w założeniach klauzuli zmienia zachowanie Prologu.
- Przestaje działać zasada, że brak dowodu P(X) skutkuje brakiem dowodu P(t), dla dowolnego podstawienia ($X \mapsto t$).
- Niech baza wiedzy to

```
p(a).
q(X):- \+ p(X).
```

- ```
  :-q(X).
  false.
  :- q(a).
  false.
  :-q(b).
  true.
  ```
  ```

Przykład – negacja jako skończona refutacja

```
p:- not(q).
q:-r(ala).
r(ola).

?-p.
true.
```

Negacja jako skończona refutacja

- ```
  r(ola).
  e(X):-r(X).

  ?- \+ e(X).
  false.
  ?-\+ e(ala).
  true.
  ```

- `?- \+ e(X).` odnosi porażkę bo `?-e(X).` odnosi sukces dla `X=ola`.
- Porażka `\+ e(X).` nie oznacza, że dla bardziej szczegółowego podstawienia (np. X=ala) nie uzyskamy sukcesu.
- W przypadku formuł atomowych (bez negacji), porażka `P(X)` implikuje porażkę dla wszystkich podstawień.
  ```

Negacja jako skończona refutacja

- Sukces `?- \+ P(X).` oznacza, że `?-P(X).` ma (skończoną) refutację.
- Refutacja `?-P(X)` oznacza, że dla żadnego podstawienia `X=t` nie można udowodnić `P(t)`.
- Oznacza to, że $\forall X \neg P(X)$.

Negacja jako skończona refutacja

- Porażka `?- \+ P(X).` oznacza, że `?-P(X).` udało się udownić dla pewnego podstawienia `X=t`.
- Oznacza to, że $\exists X P(X)$.
- Wciąż możliwe jest, że `?-\+ P(t)` odniesie sukces dla pewnego podstawienia `X=t`.

Negacja jako skończona refutacja

- Podczas sprawdzania negacji predykatu Prolog uwzględnia skonstruowane podstawienie.
- ```
  w(f(ala)).
  u(X):- X=h(Z), \+ w(X).

  ?- u(X).
  X = h(_896).
  ```
  ```

Przykład

- `liar(X) :- \+ liar(X).`

- `liar(X).` odniesie sukces (true), jeśli `not(liar(X)).` odniesie sukces.
- `not(liar(X))` odniesie sukces, jeśli `liar(X)` ma skończoną refutację.
- Wniosek, jeśli `liar(X)` ma skończoną refutację, to `liar(X)` odniesie sukces.
- `liar(X)` odniesie porażkę (false), jeśli `not(liar(X))` odniesie porażkę.
- `not(liar(X))` odniesie porażkę, jeśli `liar(X)` odniesie sukces.
- Wniosek: jeśli `liar(X)` odniesie sukces, to `liar(X)` odniesie porażkę.

Przykład

- `liar(X) :- \+ liar(X).`

- Jeśli `liar(X)` ma skończoną refutację, to `liar(X)` odniesie sukces.
- Jeśli `liar(X)` odniesie sukces, to `liar(X)` odniesie porażkę.
- Wobec tego `liar(X)` nie może mieć ani skończonej refutacji ani nie może odnieść sukcesu.
- Obliczenie Prologa na `liar(X)` nie zakończy się.

Stratyfikacja programów

- Prolog nie zmusza nas do stratyfikacji programów.
- Stratyfikacja może pomóc w zrozumieniu co się dzieje z programem.
- Program P jest stratyfikowalny jeśli predykaty występujące w P można podzielić na zbiory $P_0, \ldots, P_k$ tak, że dla każdego $i \leqslant k$ i dla każdego $q \in P_i$,

  jeśli klauzula w P ma cel q, to w ciele tej klauzuli mogą wystąpić negacje predykatów tylko z $P_0, \ldots, P_{i-1}$.

Modele stabilne – stable model semantics

- Niech P program bez negacji.
- Niech M model o universum Herbranda.
- Model stabilny M dla P to minimalny model dla P.
- Jeśli P_{KRZ} to wszystkie podstawienia bez zmiennych wolnych klauzul w P, to w M prawdziwe są tylko te atomowe fakty, które dadzą się udowodnić w P_{KRZ} (w rachunku zdań).

Modele stabilne – stable model semantics

- Niech P program z negacjami a P_{KRZ} jak na poprzednim slajdzie.
- Niech M model o uniwersum Herbranda.
- Niech P_M to program powstały z P_{KRZ} gdzie:
  - jeśli w P_{KRZ} jest klauzula C postaci

    $$q \leftarrow \ldots, not(r(\bar{t}), \ldots$$

    i $r(\bar{t})$ jest prawdziwe w M to wykreślamy C,
  - usuwamy wszystkie literały z negacjami z pozostałych klauzul (te literały są prawdziwe w M).
- Model stabilny M dla P to minimalny model dla P_M.

Modele stabilne

Twierdzenie 1

Jeśli M jest modelem stabilnym dla P i z P wynika $q(\bar{t})$, to M spełnia $q(\bar{t})$.

Uwaga. Model stabilny może nie istnieć lub może nie być wyznaczony jednoznacznie. **Uwaga.** Powyżej mamy twierdzenie o poprawności SLDNF rezolucji względem modeli stabilnych. **Uwaga.** Twierdzenia o pełności dla SLDNF rezolucji wymagałyby wprowadzenia nowych pojęć.

Przykład – modele stabilne

- Program L={`liar(X) :-\+ liar(X).`} nie ma modelu stabilnego.
- Niech c będzie jedyną stałą. Mamy więc dwa modele
  - M_0 odpowiadający teorii pustej $\emptyset$,
  - M_1 odpowiadający teorii { liar(c) }.
- Program odpowiadający M_0 to `liar(c).` ale M_0 go nie spełnia.
- Program odpowiadający M_1 to program pusty, ale M_1 nie jest dla niego modelem minimalnym, a więc M_1 nie jest modelem stabilnym.

Outline

- Dodatkowe spójniki zdaniowe (w tym negacja) pozwalają na lepszą kontrolę obliczeniem Prologu.
- Zwiększają one jednak czas obliczenia.
- Zmniejszają też czytelność programu.
- Powinny być stosowane z rozwagą.

Alternatywa

- Równoważna jest koniunkcja następujących dwóch formuł
  - $A \wedge B \wedge C \rightarrow D$,
  - $A \wedge E \wedge C \rightarrow D$,
- z formułą
  - $A \wedge (B \vee E) \wedge C \rightarrow D$.
- Alternatywa, z punktu widzenia siły wyrazu logiki, nie jest w Prologu potrzebna.

Alternatywa

- Alternatywę wyraża średnik ';'.
- Alternatywa pozwala na lepszą kontrolę sterowaniem inferencji w Prologu.
- Prolog starając się uzgodnić alternatywę (A;B) najpierw uzgadania A a następnie, kiedy wyczerpie wszystkie możliwości uzgodnienia A, uzgadania B.
- Zakładamy tutaj, że A nie jest implikacją, czyli nie jest postaci (C->D).

Implikacja

- Rozważmy implikację postaci

```
(Warunek -> Wtedy), Kontynuacja.
```

- Aby uzgodnić `(Warunek -> Wtedy)` Prolog wykonuje `Warunek,!,Wtedy`.
- Jeśli się to powiedzie, to wykonywana jest `Kontynuacja`.
- Jeśli `Warunek` się nie powiedzie, to cała implikacja uznawana jest za nieuzgodnioną (inaczej niż klasycznie).
- Uwaga! `Warunek` jest uzgodniany tylko raz (odcięcie).

Implikacja ’if-then-else’

- Rozważmy implikację postaci

```
(Warunek -> Wtedy;Przeciwnie), Kontynuacja.
```

- Aby uzgodnić `(Warunek -> Wtedy;Przeciwnie)` Prolog uzgadnia `Warunek`.
- Jeśli się to powiedzie, to wykonywana jest `!,Wtedy`, a potem, w przypadku sukcesu, `Kontynuacja`.
- Jeśli `Warunek` się nie powiedzie, wykonywane jest `Przeciwnie`, i w przypadku sukcesu `Kontynuacja`.

Przykład

- Term

  `(Warunek->Wtedy;true)`

  odpowiada klasycznej implikacji, która jest prawdziwa, gdy jej poprzednik jest fałszywy.
- Gdy `Warunek` nie powiedzie się, nie jest tworzone żadne nowe podstawienie.

Outline

Struktury rekurencyjne

- Struktury rekurencyjne (lista, drzewo, . . .) reprezentujemy w Prologu jako termy.
- Szczegóły reprezentowania takich struktur były prezentowane na poprzednich wykładach.
- Teraz zajmiemy się pewnymi ciekawymi technikami programowania z wykorzystaniem list.

Outline

Programowanie z akumulatorem

- Przeglając struktury często potrzebujemy zachowywać elementy, które już odwiedziliśmy lub częściowe wyniki obliczenia.
- Służy do tego technika programowania z akumulatorem.

- Możemy policzyć długość listy używając akumulatora

```
listlen(L,N):-listlenacc(L,0,N).

listlenacc([],A,A).
listlenacc([G|O],A,N):- A1 is A+1, listlenacc(O,A1,N).
```

- W akumulatorze przechowujemy ile elementów listy już widzieliśmy.

Przeglądanie grafu

- Czy wierzchołek Y jest osiągalny z wierzchołka X?

```
e(a,b).
e(b,c).
e(c,a).
```

- ```
  dfs(X,X,A):-!.
  dfs(X,Y,A):-e(X,Z), \+ member(Z,A), dfs(Z,Y, [Z|A]).

  dfs(X,Y):-dfs(X,Y,[X]).
  ```
- Akumulator przechowuje listę odwiedzonych wierzchołków.
- Dzięki temu program nie zapętla się choć na różnych ścieżkach może odwiedzić ten sam wierzchołek.
  ```

Outline

Listy różnicowe

- Jest to jedna z technik programowania w Prologu.
- Używając jej, reprezentujemy listę przez dwie zmienne.
- Pierwsza zmienna to lista wynikowa i ogon, druga to ogon.
- Wtedy możemy wstawiać nowe elementy przed ogonem.
- Druga zmienna musi być nieukonkretniona.

Listy różnicowe

- Listę [a,b,c] reprezentujemy jako dwa termy
  - [a,b,c|X] oraz X.
- [a,b,c] = [a,b,c|X] - X.
- Możemy dodać element na koniec listy [a,b,c] przez wykonanie unifikacji.
  - Jeśli [a,b,c|X], X to lista różnicowa to dodanie d na koniec listy odpowiada unifikacji X=[d|Z] i nowa para to [a,b,c,d|Z](==[a,b,c|X]) i Z.
  - Czyli możemy dodawać na koniec listy różnicowej w czasie stałym.

Listy różnicowe

- Aby zmienić listę różnicową Z,X na zwykłą listę L wystarczy zunifikować X z listą pustą.
  - X=[], L=Z.
- Aby utworzyć listę różnicową Z,X z listy L trzeba dołożyć do niej ogon, czyli wykonać
  - append(L,X,Z),

  gdzie Z to nieukonkretniona zmienna.

Odwracanie listy przy pomocy list różnicowych

```
reverse([],[]).
reverse([X|G],L):- reverse(G,M),
                   append(M,[X],L).
```

- Koszt tego rozwiązania jest kwadratowy względem długości odwracanej listy.
- Dla listy długości n wykonujemy $(n-1)$ razy predykat append, którego koszt to długość pierwszego argumentu.
- Suma $1+2+...+(n-1) = O(n^2)$.

Odwracanie listy przy pomocy list różnicowych

- ```
  reverse_diff([], X,X).
  reverse_diff([A|O],Y,X):- Z=[A|X],
                            reverse_diff(O,Y,Z).
  reverse(L,R):-  reverse_diff(L,R,[]).
  ```
- Długość obliczenia (dowodu) jest liniowa.
- Przykładowe obliczenie reverse_diff:

  ```
  reverse_diff([a,b,c],R,[]).
  Z1=[a|[]]=[a], reverse_diff([b,c],R,Z1).
  Z2=[b|Z1]=[b,a], reverse_diff([c],R,Z2).
  Z3=[c|Z2]=[c,b,a], reverse_diff([],R,Z3).
  R=Z3=[c,b,a].
  ```
  ```

Koniec