Programowanie

Egzamin poprawkowy — rozwiązania

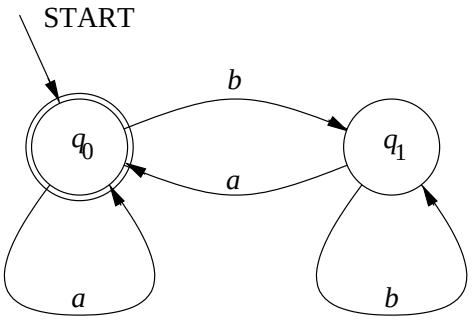
17 września 2002

Zadanie 1. Po pierwsze zauważmy, że wyrażenie można znacznie uprościć. Niech $e_1 = e_2$ oznacza, że wyrażenia e_1 i e_2 opisują ten sam język oraz niech $e_1 \subseteq e_2$ oznacza, że jezyk opisywany wyrażeniem e_1 jest podzbiorem języka opisywanego wyrażeniem e_2.

Każde słowo z języka opisanego wyrażeniem abb^*a jest konkatenacją pewnych słów z języków opisywanych wyrażeniem $(ab)^*a$ oraz bb^*a. Składnik abb^*a można więc opuścić, ponieważ każde słowo tej postaci da się wyprowadzić z pozostałych składników „w dwóch przebiegach". Formalnie, mamy $a \subseteq (ab)^*a + bb^*a$ oraz $bb^*a \subseteq (ab)^*a + bb^*a$, więc $abb^*a \subseteq ((ab)^*a + bb^*a)^2$, gdzie napis e^2 oznacza wyrażenie ee. Zatem

$$((ab)^*a + bb^*a + abb^*a)^* = ((ab)^*a + bb^*a)^*$$

Zauważmy, że $(ab)^*a = a(ba)^*$. Możemy więc wyrażenie przekształcić do postaci $(a(ba)^* + bb^*a)^*$. Słowo z języka opisanego wyrażeniem $a(ba)^*$ jest postaci ax, gdzie x to słowo z języka $(ba)^*$. Zauważmy, że x należy do języka opisanego wyrażeniem $(a + bb^*a)^*$. Człon $(ba)^*$ pierwszego składnika można więc opuścić, ponieważ słowo x można wyprowadzić w „wielu przebiegach" z drugiego składnika. Formalnie, mamy $(ba)^* \subseteq (a + bb^*a)^*$, więc $a(ba)^* \subseteq (a + bb^*a)^*$. Stąd $(a(ba)^* + bb^*a)^* = (a + bb^*a)^*$. Mamy zatem zbudować automat rozpoznający język opisany wyrażeniem $(a + bb^*a)^*$. (Zauważmy, że wyrażenie to opisuje język złożony ze słowa pustego i wszystkich słów zakończonych literą a). Domknięcie Kleene'go na zewnątrz wyrażenia sugeruje, że stan początkowy q_0 naszego automatu powinien być stanem akceptującym (ponieważ słowo puste należy do języka). Za każdym razem, gdy automat przeczyta słowo z języka opisanego wyrażeniem $a + bb^*a$ powinien do tego stanu powrócić. Język ten zawiera słowo a oraz słowa złożone z co najmniej jednej litery b i zakończone literą a. Automat więc będzie wyglądał następująco:



Zadanie 2. (Zob. też np. Ben-Ari, wyd. 2, str. 114, Habermann-Perry, str. 278–289).

```
task BUFOR is
   entry WSTAW (X : in ELEM);
   entry POBIERZ (X : out ELEM);
end BUFOR;
```

```
task body BUFOR is
   ROZMIAR : constant INTEGER := 100;
   subtype INDEKS is INTEGER range 0..(ROZMIAR-1);
   T : array (INDEKS) of INTEGER;
   P,Q : INDEKS := 0;
   N : INTEGER range 0..ROZMIAR := 0;
begin
   loop
      select
         when N<ROZMIAR =>
         accept WSTAW (X : in ELEM) do
            T(Q) := X;
            Q := (Q+1) mod ROZMIAR;
            N := N+1;
         end WSTAW;
      or
         when N>0 =>
         accept POBIERZ (X : out ELEM) do
            X := T(P);
            P := (P+1) mod ROZMIAR;
            N := N-1;
         end POBIERZ;
      end select;
   end loop;
end BUFOR;
```

Zadanie 3. Niech cel `notDivisible`(n, l) będzie spełniony, gdy n nie dzieli się przez żadną z liczb z listy l (term n powinien być liczbą, l listą liczb; oba parametry są więc parametrami wejściowymi i muszą być ukonkretnione w chwili wywołania predykatu). Odpowiedni predykat `notDivisible/2` można zdefiniować następująco:

```
notDivisible(_,[]).
notDivisible(X,[H|T]) :-
   X mod H =\= 0,
   notDivisible(X,T).
```

Operację poszukiwania kolejnej liczby X nie mniejszej niż n nie przechodzącej przez sito l (niepodzielnej przez żadną liczbę z listy l) realizuje cel `sieve`(n, [X|l]). Termy n i l są tu parametrami wejściowymi i muszą być ukonkretnione, X zaś może być zarówno nieukonkretnioną zmienną, jak i atomem całkowitoliczbowym.

```
sieve(H,[H|T]) :-
   notDivisible(H,T),
   !.
sieve(X,L) :-
   Y is X+1,
   sieve(Y,L).
```

Jeśli pragniemy sprawdzić, czy podana liczba jest pierwsza, wówczas należy iterować operację wyszukiwania kolejnej liczby pierwszej tak długo, aż natrafimy na tą liczbę, lub znajdziemy wszystkie liczby pierwsze nie mniejsze od niej nie natrafiwszy na nią:

```
nextPrime(H,[H|_]).
nextPrime(N,[H|_]):-
   integer(N),
   N<H,
   !,
   fail.
nextPrime(N,[Y|T]) :-
   Z is Y+1,
   sieve(Z,[X,Y|T]),
   nextPrime(N,[X,Y|T]).
```

Predykat `prime/1` sprawdza wpierw, czy jego parametr jest nieukonkretnioną zmienną i wówczas przechodzi do generowania kolejnych liczb pierwszych. W przeciwnym przypadku próbuje wyliczyć wartość argumentu (zrywając obliczenia, gdy parametr okaże się nie być termem stałym reprezentującym wyrażenie arytmetyczne) i następnie przechodzi do sprawdzenia, czy wyznaczona wartość jest liczbą pierwszą.

```
prime(X) :-
   var(X),
   nextPrime(X,[2]).
prime(X) :-
   Y is X,
   nextPrime(Y,[2]).
```

Przy kolejnych nawrotach do listy liczb pierwszych dołączane są kolejne wyszukane liczby. W liście tej przechowujemy więc liczby w porządku malejącym. Wadą przedstawionego rozwiązania jest to, że „kandydatów” na kolejną liczbę pierwszą przesiewamy przez listę liczb pierwszych w malejącej kolejności (wykonując wiele niepotrzebnych dzieleń). Korzystając z list otwartych można by odwrócić tę kolejność.

Zadanie 4.

```
fun perm (x::xs) =
    let
      fun revapp (ys,zs) = foldl op:: zs ys
      fun ins vs (ws as u::us,wss) =
             ins (u::vs) (us,revapp(vs,x::ws)::wss)
        | ins vs (nil,wss) = revapp(vs,[x])::wss
    in
      foldl (ins nil) nil (perm xs)
    end
  | perm nil = [nil]
```