12. přednáška

Rekurze

Dokazování správnosti programů

Rekurze

O struktuře se říká, že je rekurzivní jestliže je definována sama sebou, nebo jestli částečně obsahuje sama sebe.

Rekurze je zvlášť významná v matematických definicích. např.:

Přirozená čísla:

a) 1 je přirozené číslo

b) každý následník přirozeného čísla je přirozené číslo

Faktoriál:

a) 0!=1

b) pro n>0 je n!=n*(n-1)!

Mocnost rekurze

- Mocnost rekurze spočívá ve schopnosti definovat nekonečnou množinu objektů konečným popisem a to i v případě, že ve struktuře není explicitně uvedena iterace.

Efektivní vyčíslitelnost

- Funkce $f(x_1,x_2,...,x_n)$ je efektivně vyčíslitelná, existuje-li postup na základě určitých pravidel (neintuitivní), pomocí něhož můžeme stanovit hodnotu $f(k_1,k_2,...,k_n)$ pokaždé, jsou-li zadány relevantní parametry $k_1,k_2,...,k_n$. Takový postup se nazývá algoritmem.

Algoritmus, Markovovy a Turingovy algoritmy

- Algoritmus splňuje tři základní vlastnosti:
  - determinovanost,
  - Masovost,
  - resultativnost.
- Normální algoritmy Markova vycházejí z principu přepisovacího systému, jehož zvláštním případem jsou gramatiky.
- Turingovy algoritmy se definují pomocí zvláštního automatu - Turingova stroje, který slouží jako model univerzálního počítače. Jinou možností definice algoritmu jsou rekurzivní funkce.

Churchova teze – rekurzívní vyčíslitelnost

- Churchova teze říká, že jakoukoli intuitivně vyčíslitelnou funkci lze popsat s pomocí rekurzívních funkcí. Ukazuje se, že Markovovy algoritmy, Turingovy algoritmy i rekurzivní funkce mají **stejnou vyjadřovací sílu** pro popis algoritmů a že Markovovy i Turingovy algoritmy lze převést na definici jisté rekurzivní funkce. Výhodou rekurzívních algoritmů je jednoduchá struktura a homogenní forma.
- Programování ve stylu rekurzívních funkcí se označuje také jako funkcionální programování.

Nástroj rekurze

- Nezbytným a postačujícím nástrojem rekurze je pojmenovaná procedura (funkce), podprogram či operace, jejichž aktivitu lze vyvolat jménem této operace.
- Podle místa a způsobu vyvolání hovoříme o přímé a nepřímé rekurzi.

Konečnost rekurze

- Podobně jako cyklus, který konečným zápisem umožňuje popis nekonečného procesu či struktury, i rekurze musí řešit v praxi problém konečnosti. Obecně může být rekurze definována jako kompozice sekvence příkazů S neobsahující rekurzívní operaci P a operace P samotné.

  P=**n**[S,P]

  Kde **n** je úroveň zanoření rekurze.

Schéma rekurzívního algoritmu, zabezpečujícího konečnost je pak

P = if B then n[S,P]

nebo P = n[S, if B then P]

Má-li se konečnost zajistit, musí se zabezpečit, aby došlo k situaci, v níž B=false. Je-li možno zabudovat do rekurzívního algoritmu celočíselnou proměnnou n>0, pak lze rekurzívní funkci zapsat:

P(n) = if n>0 then n[S,P(n-1)]

nebo P(n) = n[S, if n>0 then P(n-1)]

Vztah rekurze a iterace

- Použití rekurze vede k rozkladu (redukci) řešené úlohy na podúlohy, které se pak řeší podobným způsobem. Po konečném počtu redukcí vzniknou úlohy, které lze řešit přímo. Řešení nadřazených úloh se získá skládáním řešení podúloh. Jde tedy o využití principu **"divide et impera"** . Pojmu rekurze se v této souvislosti někdy používá obecně pro jakýkoliv rozklad úlohy na podúlohy.

Rekurzívní technika je vhodná tam, kde jde úlohu rozložit v rozumném čase na dílčí podúlohy, jejichž celková složitost není příliš velká. Je-li **n** složitost dané úlohy a součet složitostí podúloh je **a*n** pro nějaké konstantní **a>1**, pak má odpovídající rekurzívní algoritmus polynomiální složitost. Jestliže rozklad úlohy o složitosti **n** vede na **n** úloh o složitosti **n-1**, pak má rekurzívní algoritmus exponenciální složitost.

Jedním z hlavních problémů rekurze je otázka efektivnosti časové i paměťové složitosti. Velká prostorová složitost plyne z toho, že rekurzívní postup po návratu z nižší úrovně na vyšší může potřebovat hodnoty dat nižší úrovně. Je tedy nutno pro každou úroveň rekurze **založit v paměti nový blok zpracovávaných datových struktur**.

Z teorie rekurzívních funkcí vyplývá, že pomocí rekurze **lze popsat každý program**. Postup výpočtu, který by kopíroval tento rekurzívní mechanismus, má charakter "shora-dolů" tzn, že začíná se zadaným argumentem a přes jeho "rozklad" na "menší" argumenty postupuje až k dosažení platnosti ukončovací podmínky. Tento postup bývá často neefektivní. Hlavní příčina je často ve vícenásobně (zbytečně) opakovaném vyhodnocování funkce pro některé hodnoty argumentů.

Některé iterativní programy, které využívají opačného postupu, tedy "zdola-nahoru", zabraňují několikanásobnému výpočtu průběžných hodnot a ještě přitom vystačí s jednoduchou správou paměti (bez zásobníku).

Z výše uvedeného vyplývá, že zkoumání vztahu mezi rekurzí a iterací má základní důležitost z hlediska optimalizace výpočetních postupů definovaných pomocí rekurze.

Je nutno se zabývat jak samotným problémem zamezení opakovaného vyhodnocování, tak převodem rekurzívních algoritmů
na iterativní.

Dá se ukázat, že využití iterativní řídicí struktury při výpočtu rekurzívně definované funkce spočívá ve využití tzv. **akumulačního parametru** (akumulátoru), v němž se shromažďuje informace postupně získaná na jednotlivých úrovních rekurze. Nově získaná informace může přitom překrýt předcházející informaci a tak velikost požadované paměti je určena pouze rozsahem posledně vytvořené informace, protože informace z různých úrovní rekurze není třeba ukládat "vedle sebe" do zásobníku.
Viz příklad výpočtu faktoriálu zapsaného iterací.

Použití rekurze

Rekurzívní algoritmus se nejlépe využije tam, kde řešený problém sám je definován rekurzívně. To ale neznamená, že odpovídající rekurzívní algoritmus je nejlepším řešením daného problému.

Program, v němž je vhodné se vyhnout rekurzi má tvar:

P = if B then (S; P) **(A)**

nebo P = (S; if B then P)

Pozn. P = ... rozumíme - P je definováno jako ....

Pro faktoriál lze tento problém zapsat takto:

```
P= if i<n then (i:=i+1; F:=i*F; P)
i:=0; F:=1; P
```

První řádek lze přepsat do procedury:

```
procedure P;
begin
   if i<n
   then begin i:=i+1; F:=i*F; P
   end
end;
```

Častější, ale v podstatě ekvivalentní, je použití funkce, v níž **i** hraje roli parametru a **F** je přímo jménem funkce, takže P není potřebné:

```
function F(i:integer):integer;
begin
   if i>0
   then F:=i*F(i-1)
   else F:=1
end
```

Je zcela jasné, že v tomto případě je vhodné rekurzi nahradit iterací

```
i:=0; F:=1;
while i<n do
begin
   i:=i+1; F:=i*F
end
```

Obecně lze programy zapsané schématem **(A)** zapsat

P = (x:=x0; while B do S)

Ještě výraznější a složitější příklad lze uvést v souvislosti s Fibonacciho posloupností:

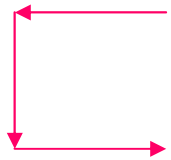
```
function Fib(n:integer):integer;
begin
   if n=0
   then Fib:=0
   else
      if n=1
      then Fib:=1
      else Fib:=Fib(n-1) + Fib(n-2)
end
```

Hilbertovy křivky

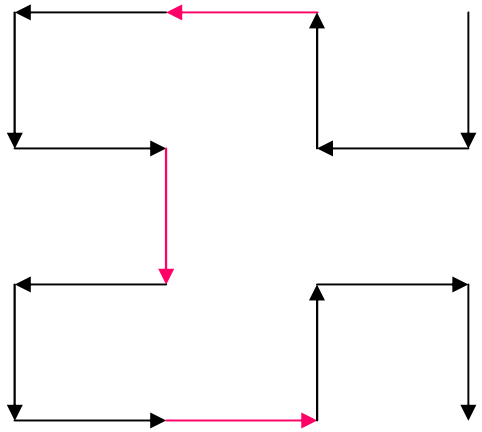
Zajímavé rekurzívní algoritmy se váží ke grafickým problémům. Jedním z nich jsou tzv Hilbertovy křivky.

Hilbertova křivka řádu i>1 se získá spojením čtyř Hilbertových křivek řádu (i-1) o poloviční velikosti spojkou a s odpovídající rotací:

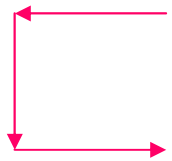
Hilbertova křivka prvého řádu H1 – čtyři Hilbertovy křivky 0-řádu (příslušně narotované) jsou spojeny (červenými) spojkami



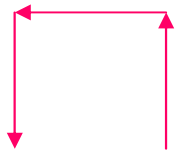
Hilbertova křivka 2. řádu H2 – 4ryři Hilbertovy křivky H1 – příslušně narotované, jsou spojeny (červenými) spojkami



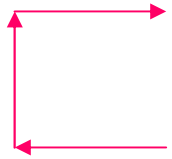
Rekurzívní schema je dáno popisem:



A: D (doleva) A (dolů) A (doprava) B



B: C (nahoru) C (doleva) B (dolů) A



C: B (doprava) D (nahoru) C (doleva) D

D: A (dolů) D (doleva) D (nahoru) C

```
procedure A(i:integer);
begin if i>0
        then begin D(i-1); Line(x,y,x-h,y); x:=x-h;
                   A(i-1); Line(x,y,x,y-h); y:=y-h;
                   A(i-1); Line(x,y,x+h,y); x:=x+h;
                   B(i-1)
             end
end;
```

Podobným způsobem lze definovat i procedury B,C a D.
Program pro vytvoření Hilbertových křivek řádu až n má tvar:

```
program Hilbert;
const n=4; h0=512;
var i,h,x,y,x0,y0:integer;
procedure A(i:integer);
begin ..... end;
procedure B(i:integer);
begin ..... end;
procedure C(i:integer);
begin ..... end;
procedure D(i:integer);
begin ..... end;
begin
   i:=0; h:=h0; x0:=h div 2; y0:=X0;
   repeat (* Kresli křivku o řádu i *)
      i:=i+1; h:=h div 2;
      x0:=x0 + (h div 2); y0:=y0 + (h div 2);
      x:=x0; y:=y0;
      A(i)
  until i=n;
end.
```

Algoritmy s návratem

Pro algoritmy s návratem je typické, že řešení se nehledá předpisem pro jeho dosažení, ale metodou pokusů a omylů. Na této cestě, na které je mnoho "křižovatek", se vydáme vždy prvním z několika možných směrů, a zaznamenám si cestu. Jsme-li donuceni vrátit se do tohoto místa pro neúspěch, volíme příště další z možností. Algoritmus skončí neúspěšně, vyčerpají-li se všechny možnosti a nedojde-li ke splnění podmínek, které řešení vyžaduje.

Cesta koně a N dam

Typickými příklady algoritmů této třídy jsou spojeny s šachovnicí: "Cesta koně" a "N dam". Algoritmus "Cesta koně" je nalezení posloupnosti polí na šachovnici, kterými kůň projde celou šachovnici tak, aby prošel všemi poli a žádným dvakrát.

"N dam" je nalezení takového umístění N dam na šachovnici o rozměrech N*N, aby se podle pravidel šachu vzájemně
neohrožovaly.

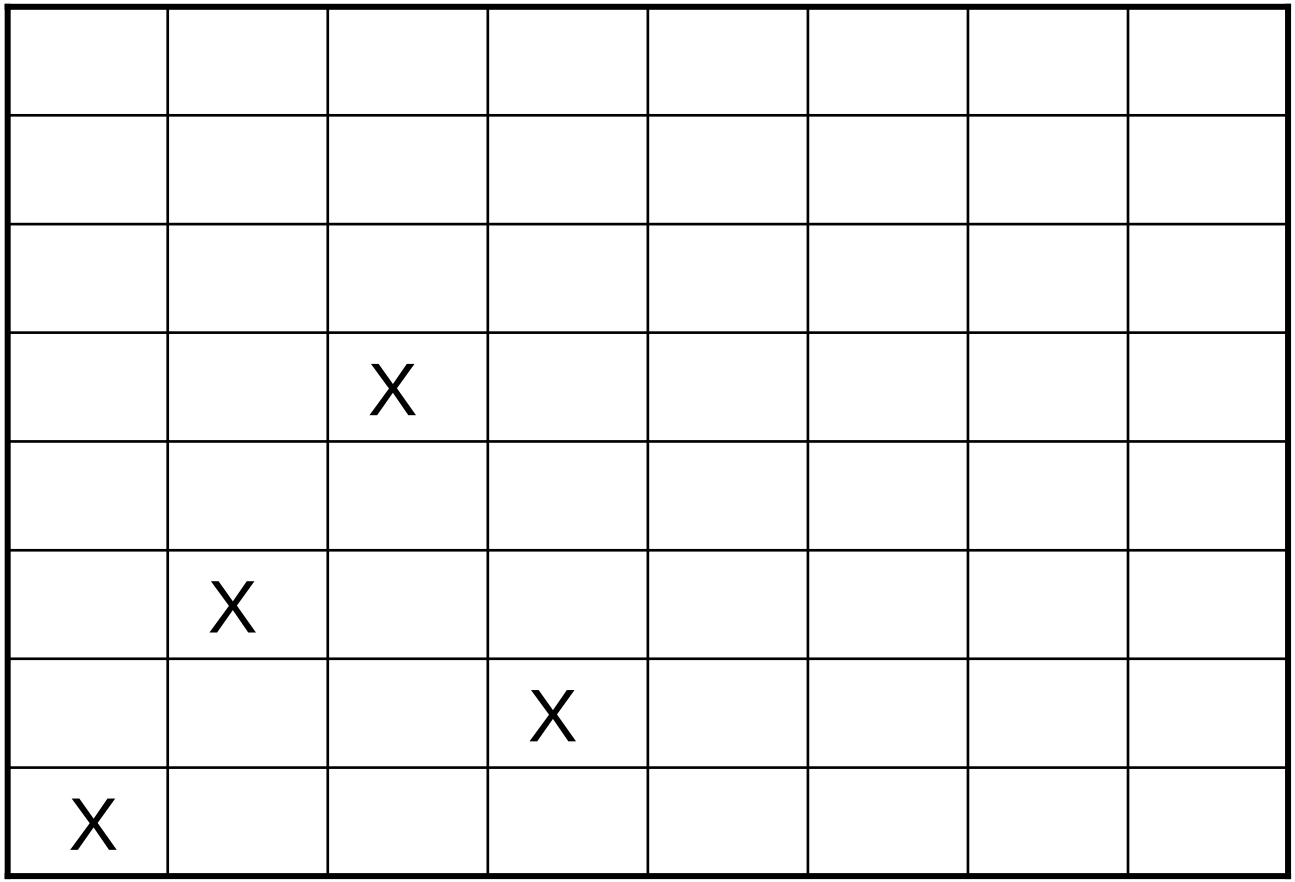
V prvním přístupu se pokusme definovat pokus o další tah koně z prvního problému:

```
procedure "pokus o další tah"
  begin "inicializace výběru tahu";
        repeat
          "vyber následující z možných tahů"
          if "tah je přijatelný"
             then begin
                "zaznamenej tah";
                if "šachovnice není vyčerpaná"
                then begin
                    "pokus o další tah"
                     if "pokus se nezdařil"
                     then "vymaž zaznamenaný tah"
                end (* if „šachovnice …“*)
              end (* if „tah je přijatelný“ *)
         until ("tah byl úspěšný") or ("není další možný tah")
  end;
```

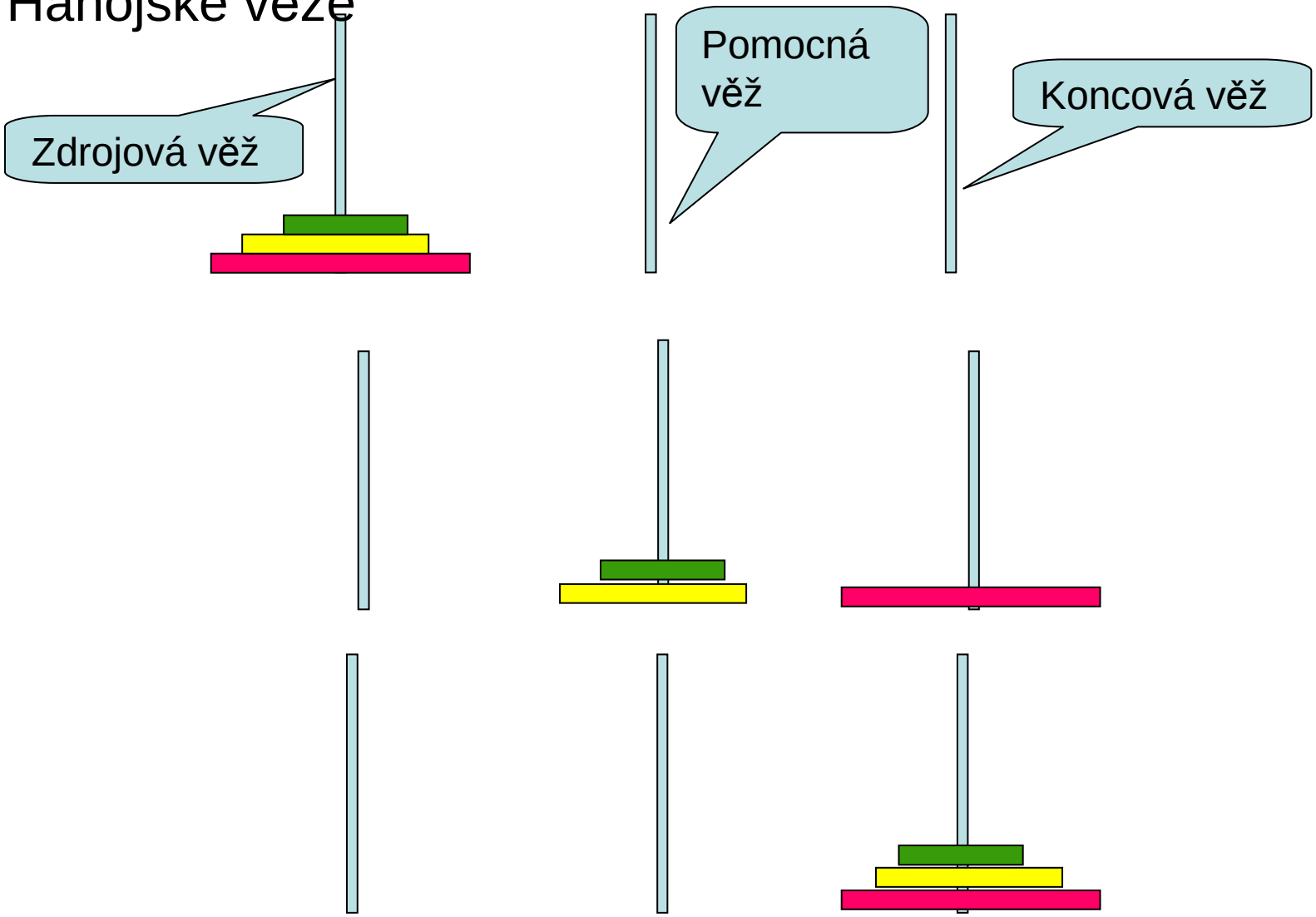
Problém Osmi dam lze rekurzívně zapsat takto:

```
procedure zkus(i:integer);
begin
   "inicializuj výběr pozice pro i-tou dámu";
   repeat
      "vyber další pozici";
      if "dáma nikoho neohrožuje"
      then begin
          "ustav dámu";
           if i<8 then begin
               zkus(i+1);
               if "nepodařilo se"
               then "odstraň dámu"
           end (* if i<8 *)
      end (* if „dáma nikoho …“ *)
   until "podařilo se" or "není další pozice pro dámu";
```

Šachovnice pro nerekurzívní řešení 8 dam



Hanojské věže



Rekurzívní verze:

```
procedure PRESUNVEZ(Vyska:integer; var Odkud,
Kam, Pom: integer);
begin
   if Vyska>0
   then begin PRESUNVEZ(Vyska-
1,Odkud,Pom,Kam);
        Presundisk(Odkud,Kam);
        PRESUNVEZ(Vyska-1,Pom,Kam,Odkud)
   end
end;
```

Nerekurzívní verze

```
procedure Strkej(V,O,K,P:integer);
var Pom:integer ;
begin
     while V<>0 do begin
        Push(V,O,K,P);  K:=:P; V:=V-1
     end
end;
procedure PRESUNVEZ(V,O,K,P:integer);
begin
     StackInit;
     Strkej(V,O,K,P);
     while not SEmpty do begin
        Pop(V,O,K,P);
        PrenesDisk(O,K);
        Strkej(V-1,P,K,O)
   end (* while *)
end;
```

Dokazování správnosti algoritmu

- Správnost algoritmu
- Dokazování správnosti algoritmů je významnou oblastí výzkumu. Je v neustálém vývoji a hledá pro důkazy nejvhodnější postupy a metody a nástroje, včetně počítačové podpory.
- Základním principem dokazování je stanovení oborů hodnot proměnných a vztahů mezi proměnnými pro každý příkaz programu.
- Vztahy se vyjadřují tvrzeními (logickými výroky), vztahujícími se k jednotlivým místům programu a nezávislými na cestě, po níž se k danému místu postup v algoritmu dostane.

Pravidlo 1:

- Pro každý příkaz algoritmu jsou nalezeny podmínky - tvrzení, které platí před a po provedení příkazu S. Tvrzení platné před příkazem se nazývá antecendence (angl. precondition) a po provedení příkazu se nazývá konsekvence (postcondition).

  ...

  (* Platí antecendence P *)

  S;

  (* Platí konsekvence Q *) ...

- Nechť příkaz S má jako antecendenci tvrzení P a konsekvenci Q, pak to zapisujeme notací (S,P)=>Q.

Pravidlo 2:

- Spojuje-li se před příkazem T několik větví algoritmu, pak konsekvence všech předcházejících příkazů S_i |(1<i<n) musí logicky implikovat antecendenci následujícího příkazu T.

```
P:
case i of
   1: S1; (* platí konsekvence Q1 *)
   2: S2; (* platí konsekvence Q2 *)
      ...
   n: Sn; (* platí konsekvence Qn *)
end (* case *);
(* Platí Qi=>P |(1<i<n) *) kde P je antecendence příkazu T *)
T;
```

Pravidlo 3:

Platí-li před podmíněným příkazem s podmínkou B tvrzení P, pak konsekvence příkazu za "then" je "B and P" a konsekvence příkazu za "else" je "not B and P".

Příklad:

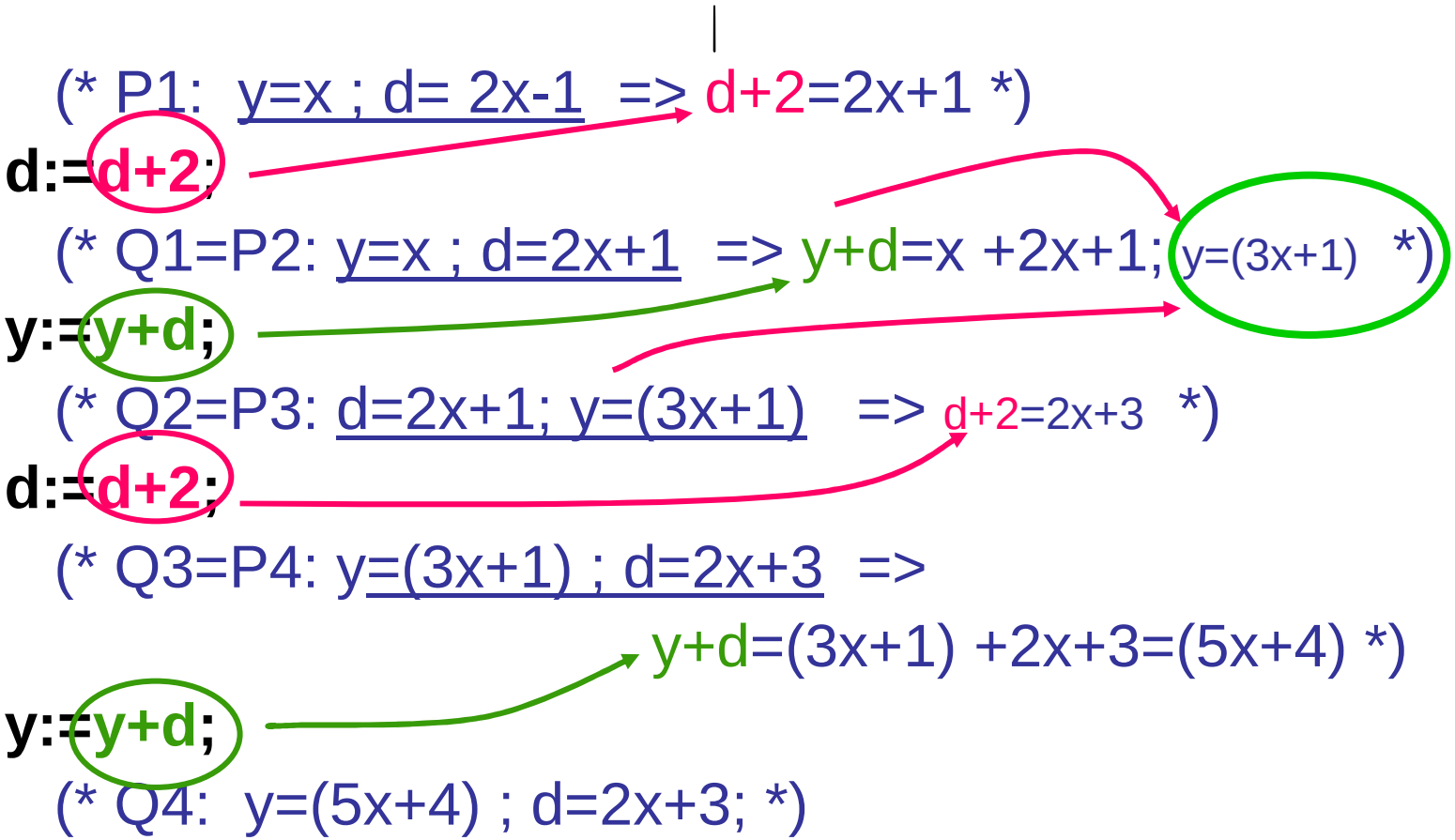
```
...
(* P = (-10<x<10)  *)
if x<0   (* B *)
  then (* Qthen = (-10<x<0) *)
  else (* Qelse = ( 0<=x<10) *)
```

Pravidlo 4:

Pro přiřazovací příkaz **v:=e** platí: Nechť P_{ev} je antecendence příkazu **v:=e**. Konsekvenci tohoto příkazu dostaneme tak, že každý výskyt výrazu (expression) **e** v antecendenci nahradíme proměnnou (variable) **v**.

- Inverzní pravidlo říká, že je-li Q konsekvence přiřazovacího příkazu **v:=e**, pak jeho antecendenci získáme náhradou každého výskytu proměnné "**v**" v konsekvenci výrazem „**e**"

Ilustrace pravidla :



Příklad

Dokažte, že sekvence příkazů:

```
x:=x+y;
y:=x-y;
x:=x-y;
```

Provede výměnu proměnných x a y

P1: (x=X;y=Y) -> x+y=X+Y

x:=x+y;

Q1=P2: (y=Y; x=X+Y) -> x-y=X+Y-Y=X

y:=x-y;

Q2=P3: (x=X+Y;y=X) -> x-y=X+Y-X=Y

x:=x-y;

Q3: (x=Y; y=X) Q.E.D.

Aplikace na násobení celých čísel pomocí sčítání a odčítání

Násobení z=x*y

(* P1: (z+u*y=x*y; u>0;) => z+y+(u-1)*y=x*y *)

z:=z+y;

(* Q1=P2: (z+(u-1)*y=x*y; (u-1)>0)*)

u:=u-1;

(* Q2: (z+u*y=x*y; u>0) *)

Z toho plyne, že pro **u=0** je **z=x*y**

Přepsáno do celkového tvaru algoritmu:

```
(* x>0; y>0 *)
   z:=0;
   u:=x;
      (* z=0; u=x *)
   repeat
       (* z+u*y=x*y; u>0 *)
     z:=z+y;
     u:=u-1;
   until u=0;
     (* z=x*y; u=0 *)
```

Dělení q = x div y

kde r je zbytek (remainder) a q je podíl (quotient)

```
(* P1: ( r+q*y=x; (r-y)>0 )=>(r-y)+(q+1)*y=x  *)
    r:=r-y;
       (* Q1=P2: ( r+(q+1)*y=x; r>0 *)
    q:=q+1;
       (* Q2 : ( r + q*y = x; r>0 *)
```

Je-li 0<r<y pak je q podíl a r zbytek

Přepsáno do celkového tvaru algoritmu

```
(* x>0, y>0 *)
    q:=0;
    r:=x;
       (* q=0; r=x *)
    while r>y do
    begin
      r:=r-y;
      q:=q+1;
        (*  q*y+r=x; r=>0   *)
    end; (* while *)
    (* q*y+r=x; 0<=r<y *)
```

Pravidlo 5a pro cyklus while

- Nechť je dáno tvrzení P, které je invariantní (neměnné) vzhledem k příkazu S (provedení příkazu S nemá na tvrzení P žádný vliv; tvrzení je současně antecendencí i konsekvencí příkazu). Pak pro cyklus S' typu "while B do", jehož vnitřním příkazem je příkaz S platí konsekvence ve tvaru P and not B.
- (S',P) => P and not B

```
(* P *)
 while B do (* příkaz S' *)
 begin
   S
 end (* while *)
   (* konsekvence:  P and not B *)
```

Pravidlo 5b pro cyklus repeat:

Nechť pro příkaz S platí dva předpoklady:

(S,P) => Q

(S,Q and not B) => Q

Pak cyklus S'' typu *repeat ... until B*, který obsahuje uvnitř příkaz S, má antecendenci P a konsekvenci Q and B

(S'',P) => Q and B

```
(* P *)
    repeat
      S
    until B;
(* Q and B *)
```

S ohledem na skutečnost, že pro cyklus typu repeat-until se vyžaduje platnost obou předpokladů (P) a také (Q and not B), bývá tento cyklus častěji zdrojem chyb než cyklus typu while, i když jeho použití někdy vede ke kratšímu zápisu.

- Invarianty cyklu jsou důležité nejen pro důkaz správnosti algoritmu, ale při slabším využití i pro stanovení podmínek konečnosti cyklu. Minimální podmínkou konečnosti cyklu je požadavek, aby příkaz S měnil hodnotu alespoň jedné proměnné, na níž závisí hodnota B.
- Obecně se konečnost cyklu zajišťuje zavedením celočíselné funkce N takové, že platí: B = N>0. Je-li na počátku cyklu hodnota funkce N>0 a každé provedení příkazu uvnitř cyklu sníží hodnotu funkce N, je konečnost cyklu zaručena, jestli cyklus končí při splnění podmínky N<=0.

Správnost algoritmu (Correctness)

Sara Baase:Computer Algorithms - Introduction to Design and Analysis

Pro ověření správnosti programu je třeba udělat 3 kroky:

1) Definovat, co rozumíme správností
2) Definovat vstupní data a požadovaný výsledek na výstupu
3) Dokázat, že algoritmus z definovaných dat produkuje požadované výsledky

Algoritmus sám má dvě významné stránky:

- Metoda zvolená k řešení
- Posloupnost akcí (příkazů), které metodu realizují

- Důkaz správnosti metody může být velmi složitý a nespadá vždy do problematiky algoritmizace (ale např. do oblasti numerické matematiky apod.)
- Zvolenou metodu implementujeme programem. Je-li program krátký a jasný, stačí ke stanovení správnosti neformální (intuitivní) prostředky. Některé detaily prověříme pečlivěji (např. počáteční a koncové hodnoty počitadel cyklu). Tyto metody nemají charakter důkazu, ale pro malé programy mohou být postačující. Pro důkaz správnosti cyklů se často používají formálnější metody, jako stanovení invariantu cyklů matematickou indukcí.

- **Invarianty cyklů** jsou podmínky a relace, které jsou splněny proměnnými na konci každého průchodu cyklem.
- **Invarianty cyklů** se konstruují velmi pečlivě tak, aby se mohlo dokázat, že na konci cyklu se dosáhlo požadovaného stavu. Invarianty cyklu se ustavují indukcí pro počet průchodů cyklem.
- Hlavním přínosem invariantu cyklu je skutečnost, že délka důkazu příkazu cyklu, který ve skutečnosti provádí opakovaně sekvenci příkazů, není úměrný celkovému počtu příkazů.

Tuto techniku budeme ilustrovat na jednoduchém příkladu algoritmu pro ***sekvenční vyhledávání hodnoty*** nebo indexu prvku v poli.

Algoritmus porovnává klíč s každou položkou, která je na řadě tak dlouho, pokud nenajde shodu nebo pokud není pole vyčerpáno. Není-li hledaná položka v seznamu, vrátí algoritmus hodnotu 0.

Algoritmus:

Vstup: Pole,n,klíč, kde pole "Pole" má n prvků téhož typu jako je klíč.

Výstup: Index - umístění položky rovné klíči v poli, nebo 0 v případě nenalezení.

1. `index:=1;`
2. `while (index<=n) and (Pole[index]<>klíč) do`
3. `  index:=index+1;`
4. `end (* while *);`
5. `if index > n then index:=0;`

Pozn. Předpokládejme zkrácené vyhodnocení booleovského výrazu, které zabrání referenci Pole[n+1].

- Než začneme dokazovat správnost, snažme se přesněji vyjádřit, co má algoritmus dělat. Požadavek může být prostý: Algoritmus určí, kolikátý prvek pole ***Pole*** o ***n*** položkách se rovná klíči. Nenalezne-li se žádný shodný prvek, je výsledná pozice rovna 0.
- Tato definice má dvě vady. Nezmiňuje se o výsledku, je-li v poli více položek se shodnou hodnotou rovnou klíči a neříká, pro jaké hodnoty ***n*** algoritmus pracuje. Můžeme předpokládat nezáporné n, ale co se stane pro n=0 ?

Přesněji lze činnost algoritmu vyjádřit takto:

Je dáno pole Pole o n prvcích (n>0) a je dán klíč stejného typu jako prvky pole. Sekvenční vyhledávání určí pořadí prvního výskytu prvku pole se shodnou hodnotou. V případě nenalezení vrátí algoritmus hodnotu 0.

Důkaz:

Indukcí ustavme následující výrok:

Pro 1<=k<=(n+1) a na konci řádku 2 platí tyto podmínky (invarianty cyklu):

- ***index=k***
- ***(ForAll index:1<=index<k)[Pole[index]<>klíč]***

a) Nechť k=1. Pak index=1 a druhá podmínka je splněna sama o sobě.

b) Předpokládejme, že podmínky jsou splněny pro nějaké **k<(n+1)** a dokažme, že výrok platí i pro ***k+1***. Na základě předpokladu indukce platí po k-tém průchodu (pro **1<=index<k**), že **Pole[index]<>klíč**. Provede-li se úspěšně test na řádku 2 (to znamená po **k+1** průchodech), můžeme předpokládat, že test byl *k* krát úspěšný (true) a to dále znamená, že **Pole[index]<>klíč** a tudíž i **Pole[k]<>klíč**. Index se ale v cyklu zvýší o 1, takže test na řádku č..2 se provede *k+1* krát. To ukončuje důkaz indukcí.

c) Nyní předpokládejme, že testy na řádku 2 se provedou právě *k* krát, kde **1<=k<(n+1).** Předpokládejme dále, že na řádku 5 mohou nastat dva případy. Výsledkem je **0** pro případ, že **k=n+1**. Podle výroku, který jsme právě dokázali platí, že

(ForAllindex:1<=index<(n+1)[Pole[index]<>klíč]

a výsledek 0 je tedy správný. Stojí za povšimnutí, že výsledek je správný i pro prázdné pole, tedy pro **n=0**. Je-li naopak **index=k<n**, pak se cyklus skončí pouze při splnění podmínky **Pole[k]=klíč**. Tím lze považovat důkaz za splněný.