Anotace

- Aritmetické výrazy, notace a převody mezi nimi,
- grafy a jejich reprezentace,

Zápis výrazu

- Jakými způsoby lze zapsat aritmetický výraz?
- $(10 + 5) * (15 - 4)/2$ – tzv. infixní notace (operátor je mezi operandy),
- $/ \ * \ + \ 10 \ 5 \ - \ 15 \ 4 \ 2$ – tzv. prefixní notace (operátor je před operandy),
- $/ \ (* \ (+ \ 10 \ 5) \ (- \ 15 \ 4)) \ 2$ (uzávorkování pro větší názornost),
- $10 \ 5 \ + \ 15 \ 4 \ - \ * \ 2 \ /$ – postfixní notace (operátor je za operandy),
- $((10 \ 5 \ +) \ (15 \ 4 \ -) \ *) \ 2 \ /$ (uzávorkováno),
- stromem: Ve vrcholu buďto operátor (vrchol má dva syny) nebo operand (číslo, takový vrchol je listem).
- Pozor, evaluace bude jen naznačena, na slidech bude pseudokód, který je třeba interpretovat s fantazií!

Aritmetické výrazy a různé notace

- Výhody a nevýhody jednotlivých notací...
- Lze všchny tyto notace (zápisy) vyhodnotit?
- Lze mezi těmito notacemi převádět?
- Ano a dokonce velmi snadno pomocí stromového zápisu.

Evaluace prefixní notace

Rekurzí: `function vyhodnot:integer;`

```
begin
      if (na vstupu je číslo) then
            vyhodnot:=hodnota_čísla_na_vstupu
      else
      begin operator:=na_vstupu();
            arg1:=vyhodnot;
            arg2:=vyhodnot;
            vyhodnot:=proved(operator,arg1,arg2);
      end;
end;
```

Strom z prefixní notace

```
function pref_strom:strom;
begin
      if(na vstupu je číslo) then
            pref_strom:=list(hodnota_na_vstupu);
      else
      begin pom:=vrchol(operator);
            pom.arg1:=vyhodnot;
            pom.arg2:=vyhodnot;
            vyhodnot:=pom;
      end;
end;
```

funkce `list` vytvoří list,
funkce `vrchol` vytvoří vrchol stupně 2,
vrchol stupně 2 obsahuje prvky `arg1` a `arg2`.

Jak ze stromu vygenerovat všechny notace?

- Rekurzívně:
- Prolezeme strom tak, že v jedné fázi vlezeme do levého syna,
- v jedné fázi do pravého syna a v jedné fázi vypíšeme údaj o operátoru.
- Všechny tři notace získáme správným uspořádáním těchto tří fází.
- Do pravého syna půjdeme vždy až po návštěvě levého syna, lišit se tedy bude jen okamžik výpisu údajů!

Generování prefixní notace

```
procedure gen_pref(v:vrchol);
begin
      if(list(v)) then
            vypis(v);
      else
      begin vypis(v);
            gen_pref(v.arg1);
            gen_pref(v.arg2);
      end;
end;
```

Funkce `vypis` vypíše operátor resp. číslo.

funkce `list` zjistí, zda je současný vrchol list.

Generování postfixní notace

```
procedure gen_post(v:vrchol);
begin
      if(list(v)) then
            vypis(v);
      else
      begin gen_post(v.arg1);
            gen_post(v.arg2);
            vypis(v);
      end;
end;
```

Funkce `vypis` vypíše operátor resp. číslo.

funkce `list` zjistí, zda je současný vrchol list.

Generování infixní notace

skoro správně

```
procedure gen_post(v:vrchol);
begin
      if(list(v)) then
            vypis(v);
      else
      begin gen_post(v.arg1);
            vypis(v);
            gen_post(v.arg2);
      end;
end;
```

Funkce `vypis` vypíše operátor resp. číslo.

funkce `list` zjistí, zda je současný vrchol list.

Generování infixní notace

správně leč ošklivě

```
procedure gen_post(v:vrchol);
begin
      if(list(v)) then
            vypis(v);
      else
      begin write('(');
            gen_post(v.arg1);
            vypis(v);
            gen_post(v.arg2);
            write(')');
      end;
end;
```

K vyhodnocení postfixní notace

Opakování:
Zásobník je datová struktura osazená operacemi:

- `push` – přidej na konec zásobníku,
- pop – uber z konce zásobníku,
- tedy kdo později přijde, ten je dříve odejit.

Evaluace postfixní notace

```
function eval_post:integer;
begin
      while not eof do
      begin if (na_vstupu_cislo) then
                  push(cislo);
            if (na_vstupu_operator) then
            begin arg2:=pop;
                  arg1:=pop;
                  push(operator(arg1,arg2));
            end;
      end;
      writeln(pop);{Výsledek je na vrchu zásobníku}
end;
```

Strom z postfixní notace

```
function strom_post:vrchol;
begin
      while not eof do
      begin if (na_vstupu_cislo) then
                  push(list(cislo));
            if (na_vstupu_operator) then
            begin pom:=vrchol(operator);
                  pom.arg2:=pop;
                  pom.arg1:=pop;
                  push(pom);
            end;
      end;
      strom_post:=pop;{Výsledek na vrchu zásobníku}
end;
```

Evaluace stromu

je snad jasná, ale přesto:

```
function eval_tree(v:vrchol);
begin
      if(list(v)) then
            eval_tree:=value(v)
      else
      begin arg1:=eval_tree(v.arg1);
            arg2:=eval_tree(v.arg2);
            op:=operator(v);
            eval_tree:=op(arg1,arg2);
      end;
end;
```

Evaluace infixní notace

aneb Masakr u katova stromu

- Jedna možnost je najít operátor, který se provede jako poslední,
- výraz podle něj rozdělit, zavolat se na každou část zvlášť a nakonec provést operátor.
- Výhoda: Po rozmyšlení, jak najít poslední operátor snadno implementovatelné.
- Nevýhoda: Neustále traverzujeme výrazem a hledáme operátory.
- Jak třeba lze najít poslední provedený operátor:
  1. Najdi poslední operátor sčítání nebo odčítání mimo závorky,
  2. pokud nebyl nalezen, hledej poslední operátor násobení mimo závorky,
  3. pokud koukáme na samotné číslo, vyhodnoť ho,
  4. jinak oloupej závorky,
  5. Vyřeš body 1–4 pro vnitřní výraz.

Pascalův trojúhelník

- Obsahuje kombinační čísla,
- n-tý řádek konkrétně obsahuje hodnoty $\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}$,
- při výpočtu využíváme toho, že $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$,
- pro jednoduchost chceme spočítat jen číslo $\binom{n}{k}$.

Pascalův trojúhelník rekurzívní řešení

- Získali jsme rekurenci $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$,
- z této snadno sestrojíme rekurzívní program:

```
function kombin(n,k:integer):longint;
begin
      if (k=0) or (k=n) then kombin:=1;
      else  kombin:=kombin(n-1,k-1)+kombin(n-1,k);
end;
begin
      kombin(100,50);
end.
```

Mále stejný problém, pořád počítáme to samé mockrát.
Opět přidáme cache na výsledky.

Pascalův trojúhelník

```
const MAX=100;
var cache:array[0..MAX,0..MAX] of longint;
function kom(n,k:integer):longint;
begin
    if cache[n,k]=0 then
    begin if (k=0) or (k=n) then cache[n,k]:=1
       else cache[n,k]:=kom(n-1,k-1)+kom(n-1,k);
    end;
    kom:=cache[n,k];
end;
var n,k:integer;
begin
    read(n,k);
    writeln(kom(n,k));
end
```

Definice grafu

Definition

Grafem nazveme uspořádanou dvojici $G = (V, E)$, kde V nazveme množinou vrcholů a $E \subseteq \binom{V}{2}$ nazveme množinou hran.

Definition

Uspořádanou dvojici $G = (V, E)$ nazveme orientovaným grafem s množinou vrcholů V a množinou hran E, jestliže $E \subseteq V \times V$.

- O grafech jste se učili na Diskrétní matematice, měli jste zřejmě i vybrané algoritmy. A algoritmy jsou typicky určeny k naprogramování.
- Definice je pěkná, ale při programování nám nepomůže. Podbízí se otázka:
- Jak graf reprezentovat při programování?

Reprezentace

- Z diskrétní matematiky znáte:
- Matici sousednosti A_G
  – je čtvercová matice obsahující nuly a jedničky, jejíž řádky a sloupce jsou indexovány vrcholy. Jednička odpovídá tomu, že mezi dotyčnými vrcholy hrana vede, nula znamená, že hrana nevede.
- Matici incidence B_G – řádky jsou indexovány vrcholy, sloupce hranami, jednička na pozici $B_G[i, j]$ říká, že hrana j přiléhá k vrcholu i.
- Výhody a nevýhody?
- Jak mezi těmito reprezentacemi převádět?

Převod A_G na B_G a zpět

```
snuluj(B_G);
index_hrany:=1;
for i:=1 to n do begin
      for j:=i+1 to n do begin
      if(A_G[i,j]=1) then
      begin
            B_G[i,index_hrany]:=1;
            B_G[j,index_hrany]:=1;
            inc(index_hrany);
      end;
end;
```

B_G na A_G

Buďto podobnou analýzou matice incidence, nebo:
$A_G := B_G \times {B_G}^T$;
`for i:=1 to n do`
$A_G[i, i] := 0$;

Důkaz.

Snadné cvičení z Kombinatoriky a grafů I. □

Další reprezentace grafů

- Seznam vrcholů a k nim přilehlých hran:
- Tedy udržujeme seznam vrcholů a ke každému vrcholu vedeme seznam hran, které z něj vedou.
- Jelikož zatím neumíte spojové seznamy, bylo by potřeba nejspíše použít pole.

Funkce (proměnné) potřebné (resp. postačující) pro práci s grafem:

- `najdi_sousedy(v)`,
- `vrcholy`,
- `hrany` nebo `hrana(u,v)`, to ale umíme zjistit pomocí `vrcholy` a `najdi_sousedy`,
- případně další (`vaha_vrcholu(v)`, `vaha_hrany(e)...`).
- Výhody a nevýhody?
- Je-li graf orientovaný, musíme reprezentaci modifikovat.

Sled, tah, cesta, kružnice

Definition

- Sledem délky k nazveme posloupnost hran tvaru $\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$.
- Tahem nazveme sled, v němž se každá hrana vyskytne nejvýše jednou.
- Cestou nazveme tah (nebo sled), ve kterém se každý vrchol vyskytuje nejvýše jednou (přesněji kde se každý vrchol vyskytuje právě ve dvou po sobě jdoucích hranách).
- Tah nazveme kružnicí, pokud začíná a končí v tomtéž vrcholu a pokud se v něm každý vrchol objeví právě jednou.

Souvislost, strom

Definition

- Graf je souvislý, pokud se lze z každého jeho vrcholu dostat do každého jiného (vrcholu).
- Graf je strom, pokud je souvislý a neobsahuje kružnice.

- Definice jsou pěkné, ale pomohou nám při programování?
- Jak ověříte, zda je graf souvislý?
- Použijeme vhodné tvrzení.
- Jak zjistíte, zda je graf strom?
- Podobně.