Anotace

- Aritmetické výrazy, notace a převody mezi nimi,
- Pascalův trojúhelník,
- pořadí násobení matic.
- Demonstrace ladění programu.

Zápis výrazu

- Jakými způsoby lze zapsat aritmetický výraz?

Zápis výrazu

- Jakými způsoby lze zapsat aritmetický výraz?
- $(10 + 5) * (15 - 4)/2$ – tzv. infixní notace (operátor je mezi operandy),

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),

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),

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),

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),

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).

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:

Jak ze stromu vygenerovat všechny notace?

- Rekurzívně:
- Prolezeme strom tak, že v jedné fázi vlezeme do levého syna,

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.

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.

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;
```

Pascalův trojúhelník

- Obsahuje kombinační čísla,

Pascalův trojúhelník

- Obsahuje kombinační čísla,
- n-tý řádek konkrétně obsahuje hodnoty $\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}$,

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 $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$,

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
```

Násobení matic

- Násobení matic není komutativní, ale je asociativní.
- Máme-li vynásobit několik matic, nemůžeme jejich pořadí měnit,
- můžeme součin všelijak závorkovat.
- Různá uzávorkování jsou různě výhodná.
- Které z nich je to nejvýhodnější?
- Předpokládáme, že máme abstraktní funkce zjistící ze zadaných rozměrů složitost součinu matic správných tvarů.

Násobení matic

- Některé násobení provedeme jako poslední.

Násobení matic

- Některé násobení provedeme jako poslední.
- To úlohu rozdělí na (nejvýše) dvě další instance.

Násobení matic

- Některé násobení provedeme jako poslední.
- To úlohu rozdělí na (nejvýše) dvě další instance.
- V každé (menší) instanci opět nějaké násobení provedeme jako poslední.

Násobení matic

- Některé násobení provedeme jako poslední.
- To úlohu rozdělí na (nejvýše) dvě další instance.
- V každé (menší) instanci opět nějaké násobení provedeme jako poslední.
- Ideální podhoubí pro rekurzi!

Násobení matic

- function slozitost(od,az_do:integer):longint;
- for i:=od to az_do-1 do begin
- slozitost:=slozitost(od,i)+ slozitost(i+1,az_do)+samotne_nasobeni;

Opět budeme zbytečně násobit stále to samé.
Opět se toho zbavíme stejně.

Násobení matic

- function slozitost(od,az_do:integer):longint;
- if cache[od,az_do]=0 then
- for i:=od to az_do-1
- cache[od,az_do]:=slozitost(od,i)+ slozitost(i+1,az_do)+samotne_nasobeni;
- slozitost:=cache[od,az_do];

Násobení matic – poučení

- Tomuto říkáme *dynamické programování.*
- Zpravidla implementujeme pouze fázi vyplňování tabulky.
- Na tom je nepříjemné určovat, odkud vyplňování zahájit.
- Není tudíž špatné navrhnout si aspoň na počátku rekurzívní řešení a až pak rekurzi "rozbít":
- `cache[od,az_do]:=min{cache[od,i]+cache[i,az_do]+ samotne_nasobeni}`
- což stačí udělat cyklem.

Konec

Děkuji za pozornost...