Procedurálne a deklaratívne programovanie

V procedurálnom alebo imperatívnom programovaní tvorí program postupnosť príkazov. Príkazy predpisujú vykonanie operácií. Ak riadiaca štruktúra neurčí inak, príkazy sa vykonávajú v takej postupnosti, v akej sú zapísané. Programovací jazyk obsahuje jazykové konštrukcie pre vetvenie (príkaz if, case, switch,...) a pre cyklus (príkaz while-do, repeat-until,...). Programovací jazyk obsahuje pre zvolenú množinu operácií príkazy, ktorými ich možno volať. Pre vytvorenie ďalších operácií poskytuje programovací jazyk možnosť naprogramovať ich pomocou procedúr a volať volaním procedúry vrátane možnosti odovzdania parametrov. Údaje sú ukladané do pamäťových miest, ktoré v programe sa pomenujú prostredníctvom premenných. Príkazy postupne menia obsahy pamäťových miest takým spôsobom, aby sa nakoniec dosiahol požadovaný výsledok.

Vo funkcionálnom programovaní sa program chápe ako množina funkcií. Na rozdiel od procedurálneho programovania, ktoré vychádza z modelu výpočtov založeného na von Neumannovej architektúre počítača, opiera sa funkcionálne programovanie o tzv. Lambda kalkul ako jednoduchý model výpočtov.

Hodnota výrazu = výsledok výpočtu

Výrazy a príkazy

Príklad výrazov – aritmetické, boolovské, relačné výrazy. Výrazy sa vyskytujú na pravej strane priraďovacích príkazov a v ďalších kontextoch, keď sa požaduje hodnota (parametre procedúr a funkcií). Cieľom výrazu je získanie hodnoty.
Procedurálne jazyky obsahujú tiež príkazy:

- príkazy, ktoré spôsobujú zmenu riadiaceho toku (cykly, príkaz skoku, vetvenie)
- príkazy, ktoré menia stav pamäte (priraďovací príkaz, vstupno-výstupné prílazy)

Pri príkazoch má veľký význam poradie, v ktorom sa vykonávajú.

Priraďovacie príkazy:

```
i = i + 1;  a = a + i
```

majú odlišný účinok ako príkazy:

```
a = a * i;  i = i + 1
```

Príklad:
Uvažujeme tento priraďovací príkaz:

```
z  =  (a * y + b)  *  (a * y + c)
```

Výraz na pravej strane priradenia obsahuje spoločný podvýraz a * y. Väčšina kompilátorov spoločný podvýraz vyhodnocuje iba raz zámenou pôvodného priradenia nasledujúcimi dvoma priradeniami:

```
t  =  a * y ;   z =  (t + b)  *  (t + c)
```

Pri výrazoch takáto zámena je bezpečná, nakoľko ľubovolný podvýraz daného výrazu bude mať v danom kontexte vždy tú istú hodnotu.
Príklad:

Podobná situácia ale vo svete príkazov. Máme nasledujúce dve priradenia:

```
y = a * y + b
z = a * y + c
```

Opäť sa objaví spoločný podvýraz, ktorý vyberieme:

```
t = a * y
y = t + b;     z = t + c
```

výsledok priradení sa tým zmení. Problém vzniká v tom, že hodnota y sa medzi dvoma výskytmi spoločného podvýrazu a * y zmenila, a preto hodnota podvýrazov je rôzna, nakoľko sa vyhodnocujú v rôznych kontextoch.

Avšak ak chceme vykonať analýzu programu kvôli zisteniu, či spoločné podvýrazy v rôznych príkazoch je možné vyhodnocovať iba raz, je to veľmi náročný proces. Na tomto mieste vieme oceniť jednu veľkú výhodu výrazov v porovnaní s príkazmi.

Čisté výrazy
Nevykonávajú žiadne priraďovacie operácie.
Vlastnosti:

- hodnota (výsledok výrazu) nezávisí od poradia vyhodnocovania
- výraz možno vyhodnocovať paralelne

- v danom kontexte je nahradenie podvýrazu jeho hodnotou úplne nezávislé od výrazu, v ktorom sa to deje (*referenčná priehľadnosť)*
- vyhodnocovanie nespôsobuje žiadne vedľajšie účinky
- vstupy operácie sú zrejmé zo zápisu výrazu
- výsledky (účinky) operácie sú zrejmé zo zápisu výrazu

Vyhodnotiť výraz = získať jeho hodnotu
Hodnota výrazu závisí od kontextu vyhodnocovania (od hodnôt použitých mien vo výraze)

Churchova-Rosserova vlastnosť
Nezávislosť od poradia vyhodnocovania
Konštrukcia kompilátorov – zdroje počítača čo najlepšie využiť.

Výraz v jazyku C:
x + 2 * F(b)

```
int  F  (int y)                    int  G  (int y)
     {    return ( y * y);                 {  x = x + 1;
                                              return    (y * y);
```

Referenčná priehľadnosť
Čisté výrazy umožňujú redukciu spoločných podvýrazov.
v nemennom kontexte zámena podvýrazu jeho hodnotou nezávisí od výrazu a jeho vyhodnotenia.

Umožňuje optimalizácie, napr. redukciu spoločných podvýrazov (platí iba pre čisté výrazy).

Vstupy a výstupy operácie sú zrejmé zo **zápisu operácie.**
Pri procedúre alebo funkcii v procedurálnom programovacom jazyku výsledok často závisí od hodnôt globálnych premenných.
Pseudofunkcia R:

```
int  H  (int y)
     {  x  =  x + 1;
        return ( x * y );
      }
```

Použitie pseudofunkcie vo výraze – nepoznáme spôsob ako zistiť hodnotu funkcie H pre dané y, keď nie je známa hodnota x.

Funkcionálny program má formu **aplikatívnych výrazov** (konštanta alebo výraz vytvorený aplikáciou iba čistých funkcií na ich argumenty).
Zápis aplikatívnych výrazov často v prefixovom tvare.
(uvedie sa najprv operátor, za ním jeho operandy)
Príklad:
2 * a * x + b
+ (* (* (2, a), x), b)

Vo funkcionálnom programovaní poznáme iba jednu vstavanú syntaktickú konštrukciu, a to:
aplikácia funkcie na jej argumenty.
Implicitne sa reprezentuje (základná f.)

Príklad
sin (Φ)..... aplikácia funkcie sin na hodnotu Φ
sum (x, 1)....aplikácia funkcie sum na dvojicu hodnôt,
t.j. hodnoty pomenovanej x a čísla 1.

Základný princíp ***funkcionálneho*** programovania:

Všetky hodnoty vytvorené ako medzivýsledky aplikáciami funkcií sú vo vzťahu s inými časťami programu iba prostredníctvom použitia aplikácií funkcií, t.j. ako argumenty alebo výsledky funkcií.

Procedurálne jazyky používajú mechanizmus uloženia týchto hodnôt (medzivýsledkov) do pamäťových buniek. Tie sa potom používajú príp. modifikujú pri ďalších výpočtoch.

Programátori často používajú v niektorých programovacích jazykoch (LISP) oba princípy, oba mechanizmy komunikácie medzivýsledkov výpočtu. Tu už ale nehovoríme o čistom funkcionálnom programovaní.

Definícia funkcie

Program = výraz + definície všetkých mien použitých vo výraze

Totálna funkcia – priraďuje práve jeden prístup pre každý možný vstup.

Čiastočná funkcia – môže a nemusí každému vstupu priraďovať výstupnú hodnotu.

Príklady funkcií:

Vstup	*Funkcia*	*Výstup*
Text → (zdrojový program)	Kompilátor →	Preložený program
Štát →	Hlavné mesto →	Mesto

Intenzionálna definícia funkcie
Pravidlo opisujúce vzťah medzi vstupom a výstupom
Extenzionálna definícia funkcie
Množina dvojíc v tvare (vstupy, výstupy)- graf funkcie
1.
Log. Funkciu“not“ je možné definovať ako množinu dvojíc (vymenovanie dvojíc vstupov a výstupov):

not (pravda) = nepravda
not (nepravda) = pravda

funkcie s konečným a malým počtom kombinácií rôznych vstupov a výstupov
2.
Funkcia ako kompozícia už definovaných funkcií

implikácia (x, y) = or (not (x), y)

vyhodnotenie pomocou substitúcie
príklad
na vyhodnotenie výrazu:

implikácia (pravda, nepravda)

substituujeme argumenty do definujúceho výrazu funkcie implikácia:

or (not (pravda), nepravda)

vyhodnotíme výraz na hodnotu nepravda.
Niekedy funkciu nie je možné vyjadriť ako kompozíciu iných funkcií. Existuje niekoľko prípadov, z ktorých každý je zviazaný s inou kompozíciou
Príklad

Definícia funkcie signum

sign x = 1, ak x > 0
= 0, ak x = 0
= -1, ak x < 0

Niekedy je potrebné definovať funkciu ako nekonečný počet kompozícií:

```
m * n  =  0,        ak m = 0
       =  n,        ak m = 1
       =  n+n       ak  m = 2
       = n+n+n   ak  m = 3
            .............
```

Nekonečný počet prípadov nie je možné zapísať, preto táto metóda je užitočná iba vtedy ak existuje pravidlo umožňujúce generovať nezapísané prípady už zo zapísaných – rekurzívna definícia násobenia

```
m * n  =  0,                    ak  m = 0
       =  n  +  (m – 1) * n,    ak  m  > 0
```

Ľubovoľný výraz je možné vyhodnotiť opätovne pomocou substitúcie.

Príklad: výraz 2 * 3 - vyhodnotenie:

```
2 * 3  = 0,                  ak  2=0
         3 + (2-1) * 3,      ak  2>0
       =  3 + 1 * 3
       =  3 + 0,             ak  1=0
         3 + 3 + (1–1) *3    ak 1>0
       =  6 + 0 * 3
       =  6 + 0,             ak  0=0
       =  6 + (0-1) * 3,      ak 0>0
       =  6
```

Špecifikácia definičného oboru a oboru hodnôt funkcie. Tieto zoskupenia rôznych hodnôt na základe ich určitej spoločnej vlastnosti (napr. čísla) sa označujú ako ***typy.*** Použitie funkcií ma spravidla zmysel s určitým typom hodnôt.

Určenie typov poskytuje informáciu o tom ako sa má funkcia použiť. Určía sa typy vstupných argumentov funkcie a typ výsledku.
Typy poskytujú mechanizmus umožňujúci programovať spoľahlivejšie aj vo väčšom rozsahu.

Systém ***typov vo funkcionálnom jazyku*** obvykle zahŕňa malý počet elementárnych typov (napr. int, bool, string) spolu s pridruženými elementárnymi operátormi a konštantami (napr. and, or, true, false sú združené s typom bool).
Ak sú T_1 a T_2 typy, tak ($T_1 \rightarrow T_2$) je typ funkcie, ktorej definičný obor pozostáva z hodnôt typu T_1 a ktorej obor hodnôt pozostáva z hodnôt typu T_2.
Okrem ***typového operátora*** $\rightarrow$ sa na zostrojovanie nových typov používa ešte typový operátor $\times$ (kartézsky súčin).
Príklad:

Funkcia func zreťazujúca dva reťazce je typu:
(string × string) → string

Označenie typu pre funkciu func:

func	: :	(string × string)	→	string
meno funkcie		*typy parametrov f.*		*typ výsledku f.*

Niekedy vznikne potreba, aby jedna funkcia mohla spracovávať údaje rôznych typov.
Príklad:
Funkcia fst (pre ľubovolnú dvojicu vráti jej prvý prvok bez ohľadu na typ jednotlivých prvkov dvojice).

1. ak by to mali byť iba hodnoty typu int, typ funkcie first by bol (monomorfná funkcia):

fst : : (int × int) → int

2. pre polymorfnú funkciu vyjadríme jej typ:

fst : : (α × ß) → α

α , ß sú typové premenné.

Určenie typov funkcií rozširuje možnosti verifikácie programu (odhalenie chyby esšte pred aplikáciou f).

Podľa toho, kedy systém, ktorý vyhodnocuje výrazy, vykonáva typovú kontrolu, rozlišujeme:

- *Statická typová kontrola* – vykonáva sa pred vyhodnotením výrazu, t.j.bez znalosti aktuálnych hodnôt argumentov funkcie;
- *Dynamická typová kontrola* – vykonáva sa až v čase vyhodnotenia výrazu v závislosti od aktuálnych hodnôt argumentov funkcií.

LISP iba dynamickú typovú kontrolu vykonáva.

Pri špecifikácii a implementácii čiastočnej funkcie je potrebné riešiť prípady takých argumentov, pre ktoré nie je funkcia definovaná.

Príklad:

Funkcia stringcopy (n,s) – vytvorí reťazec pozostávajúci z n výskytov reťazca s (n je prirodzené číslo).

stringcopy : : (int × string) → string

táto funkcia je definovaná len pre prirodzené čísla n. Je to čiastočná funkcia.

Uvažujeme rekurzívnu definíciu tejto funkcie:

stringcopy (n, s) = " ", ak n = 0
= stringcopy (n-1, s)^s, inak

Znak ^ označuje zreťazenie. Výpočet v prípade, ak prvým argumentom bude záporné číslo, vedie k nekonečnému cyklu, lebo argument n nikdy nedosiahne hodnotu 0. Z uvedeného je zrejmé, že definovať funkciu, v ktorej sa uvažujú iba niektoré prípady jednotlivých argumentov je nebezpečné. Zvlášť je to dôležité vtedy, ked neuvažované prípady vedú k zbytočným, prípadne nekonečným výpočtom.

Každá definícia funkcie sy vytvára na základ špecifikácie. Keď nechceme špecifikovať výsledok – Napr. hodnotu stringcopy (n, s) pre n<0, môžeme to v špecifikácii funkcie vyjadriť nasledovne:

stringcopy (n, s) je nedefinované pre n<0

V tomto prípade je špecifikácia zámerne neúplná. Ten, kto implementuje špecifikáciu sa môže rozhodnúť implementovať nedefinované prípady rôznymi nasledujúcimi spôsobmi:

- vytvoriť nejaký zmysluplný výsledok vhodného typu (napr. prázdny reťazec " " pre funkciu stringcopy)
- vytvoriť chybový výsledok vhodného typu (napr. reťazec "err")
- umožniť ľubovolný výsledok

- implementovať funkciu tak, že jej vyhodnotenie s nedefinovanými argumentami vyvolá nekonečný cyklus
- definovať funkciu tak, aby bola explicitne pre dané prípady nedefinovaná (chybové ukončenie namiesto nekonečného cyklu)

Nechať nedefinované prípady v špecifikácii nerozhodnuté vedie k ohrozeniu prenosnosti vytvorených programov. Rôzni ľudia, ktorí implementujú danú špecifikáciu, sa väčšinou rozhodnú rozdielne. V snahe predísť tomuto, môžeme napr.určiť, že takéto výrazy s nedefinovanými hodnotami sú chyby.
Jeden z častých a bezpečných spôsobov špecifikácie nedefinovaných hodnôt je zakázanie ľubovolnej hodnoty podľa typu výsledku funkcie. V prípade funkcie stringcopy (n, s) je to možné vyjadriť priamo pomocou nerovností:

stringcopy (n, s) $\neq$ t pre n<0,

kde t je ľubovolný reťazec.

Vtedy napríklad“

stringcopy (-5, “ aa“)

nemá žiadnu hodnotu, lebo pre ľubovolný reťazec t musí platiť:

stringcopy (-5, “ aa“) $\neq$ t

Potom musí platiť aj:

stringcopy (-5, “ aa“) $\neq$ “ “

a nie je možné nedefinovaný prípad implementovať výsledkom prázdny reťazec. Nakolko špecifikácia nepovoľuje, aby funkcia vrátila nejakú hodnotu, nie je pri implementácii iná možnosť ako prerušiť program alebo naprogramovať nekonečný cyklus.

Pretty/printing

```
(define  (cena_s_dph  cena  dph)
    (+  cena
        (* cena  ( /  dph  100) ) ) )
```