KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 1 ⋄ poznámky k přednášce

4. Rekurze 2

1 Jak probíhá rekurzivní výpočet

Minule jsme si ukázali rekurzivní proceduru na výpočet obecné mocniny daného čísla:

```
(define (power a n)
  (if (= n 0)
      1
      (* a (power a (- n 1)))))
```

K pochopení, jak procedura pracuje, si můžete představit, že pro každou hodnotu exponentu (parametr `n`) jsme napsali zvláštní proceduru. V první části přednášky jsme podrobně zkoumali, jak probíhá výpočet při použití těchto procedur i procedury `power` (v čistě procedurální i iterativní verzi), a dále jaká prostředí při tom vznikají. Také jsme si ukázali, jak lze výpočet podstatně zrychlit. Podrobnosti můžete vidět na prezentaci.

2 Stromově rekurzivní výpočetní proces

Definition 1 (lineárně rekurzivní výpočetní proces)**.** Výpočetní proces je *lineárně rekurzivní*, když během aplikace procedury dojde nejvýše jednou k přímé aplikaci téže procedury.

- dosud uvedené rekurzivní procedury generovaly lineárně rekurzivní výpočetní proces

Definition 2 (stromově rekurzivní výpočetní proces)**.** Výpočetní proces je *stromově rekurzivní*, když během aplikace procedury dojde k více přímým aplikacím téže procedury.

Jako příklad si ukážeme dvě procedury, které generují stromově rekurzivní výpočetní proces.

Fibonacciho posloupnost

Jde o tuto posloupnost: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$ Její prvky jsou dány následujícím předpisem:

$$a_0 = 0$$
$$a_1 = 1$$
$$a_n = a_{n-2} + a_{n-1}$$

(Každý prvek kromě prvních dvou je součtem předcházejících dvou prvků.) Následující procedura vrací daný prvek Fibonacciho posloupnosti:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (#true (+ (fib (- n 2)) (fib (- n 1))))))
```

Jak je vidět, procedura generuje stromově rekurzivní výpočetní proces.

Rozměňování

Máme k dispozici šest druhů mincí (50, 20, 10, 5, 2 a 1 Kč) a chceme v nich vyplatit danou částku. Řešíme problém, kolika způsoby je možné to udělat. Na přednášce jsem podrobně vysvětloval, jak se napíše procedura `count-change`, která to zjistí. Tady už pouze ukážu výsledek:

```
(define (count-change amount)
  (cc amount 6))

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (#true (+ (cc amount (- kinds-of-coins 1))
                  (cc (- amount (first-denom kinds-of-coins))
                      kinds-of-coins)))))

(define (first-denom kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 2)
        ((= kinds-of-coins 3) 5)
        ((= kinds-of-coins 4) 10)
        ((= kinds-of-coins 5) 20)
        ((= kinds-of-coins 6) 50)))
```

Poznámka na konec

Kdo tento dlouhý text dočetl až sem, ten se teď dozví, že na cvičení příští týden (tedy 17., 18. nebo 19. října) budete programovat krátký test. Jeho výsledky neovlivní zápočet, ale aspoň se dozvíme, jak na tom zatím jste. Programovat budete na počítači a výsledek pošlete příslušnému cvičícímu mailem.

Otázky a úkoly na cvičení

1. Je možné v proceduře `fast-power` místo procedury `power2` použít proceduru `fast-power`? Jinými slovy, je možné místo aplikace

```
(power2 (fast-power a (/ n 2)))
```

   napsat

```
(fast-power (fast-power a (/ n 2)) 2)  ?
```

   Odhadněte a zdůvodněte, k čemu by to vedlo. Teprve potom to vyzkoušejte.

2. Napište proceduru `divides?`, která zjistí, zda dané celé číslo dělí beze zbytku jiné celé číslo:

```
> (divides? 5 10)
#true
> (divides? 1 17)
#true
> (divides? 10 17)
#false
```

   Ke zjištění zbytku po dělení můžete použít proceduru `remainder`, kterou jsme si ukázali minule.

3. Mimochodem, kdybyste proceduru `remainder` neměli k dispozici, můžete si ji nějak jednoduše napsat sami?

4. Prvočíslo je, jak známo, celé číslo větší než 1, které je dělitelné jen sebou samým a jedničkou. Napište predikát, který zjistí, zda dané číslo je prvočíslo:

```
> (prime? 5)
#true
> (prime? 17)
#true
> (prime? 9)
#false
```

Je možné predikát napsat bez použití speciálního operátoru `if`?

5. *Dokonalé číslo* (*perfect number*) je číslo, které se rovná součtu všech svých dělitelů kromě sebe sama. Například číslo 6 je dokonalé, protože jeho dělitelé kromě 6 jsou 1, 2 a 3 a $1 + 2 + 3 = 6$. Číslo 12 není dokonalé, protože $1+2+3+4+6 = 16 \neq 12$. Napište predikát `perfect?`, který zjistí, zda dané číslo je dokonalé:

```
> (perfect? 6)
#true
> (perfect? 12)
#false
```

6. Jeden příklad na stromovou rekurzi. Toto je *Pascalův trojúhelník*:

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
 & 6 & & 15 & & 20 & & 15 & & 6 & \\
 & & & & & \vdots & & & & &
\end{array}
$$

   Každý řádek (kromě prvního) má o jeden prvek víc než předchozí řádek. Řádky vždy začínají a končí jedničkou, ostatní čísla se vypočítají jako součet dvou čísel ležících nad nimi. (Z obrázku by to mělo být vidět; trojúhelník samozřejmě pokračuje směrem dolů donekonečna.)

   Řádky i prvky v nich budeme číslovat od nuly, takže například prvek na řádku číslo 6 (tedy na sedmém řádku), který má na tom řádku pozici 2 (je tedy na řádku třetí), je 15.

   Napište proceduru, která vrátí daný prvek Pascalova trojúhelníka vypočítaný uvedeným způsobem:

```
> (pascal 0 0)
1
> (pascal 3 1)
3
> (pascal 3 3)
1
> (pascal 4 2)
6
> (pascal 6 2)
15
```