KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 9: Hloubková rekurze na seznamech

Učební materiál k přednášce 30. listopadu 2006
(pracovní verze textu určená pro studenty)

JAN KONEČNÝ, VILÉM VYCHODIL

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 2006

Lekce 9: Hloubková rekurze na seznamech

Obsah lekce: V této lekci pokračujeme v problematice rekurzivních procedur. Nejprve uvedeme několik metod zastavení rekurze. Dále ukážeme, že rekurzivní procedury je možné vytvářet i bez speciální formy `define`. V další části lekce představíme speciální formu `letrec` jakožto další variantu formy `let` umožnující definovat lokální rekurzivní procedury. V poslední části se budeme věnovat hloubkové rekurzi na seznamech a hloubkovým akumulačním procedurám.

Klíčová slova: hloubková rekurze na seznamech, speciální forma `letrec`, y-kombinátor, zastavení rekurze.

9.1 Metody zastavení rekurze

V předchozí lekci jsme uvedli množství definic rekurzivních procedur. Limitní podmínku rekurzivních procedur a výraz vyhodnocený po jejím dosažení jsme přitom vždy vyjadřovali pomocí speciálních forem `if` nebo `cond`. Tyto speciální formy jsme tedy v rekurzivních programech používali k „zastavení rekurze“, to jest k zastavení postupných aplikací téže procedury. Existují však i další metody, kterými lze zastavit rekurzi. V této kapitole se budeme věnovat právě těmto metodám, a příkladům jejich použití.

Rekurzi je možné zastavit

- *pomocí speciálních forem* `if` *a* `cond`

  Tato možnost byla bohatě prezentována v předcházející lekci. Nyní jen pro srovnání s dalším bodem, uvedeme (bez dalšího komentáře) vlastní implementaci predikátu `list?`, který zjišťuje, zda je jeho argument seznam:

```
(define list?
  (lambda (l)
    (if (null? l)
        #t
        (and (pair? l)
             (list? (cdr l))))))
```

- *pomocí speciálních forem* `and` *a* `or`

  V definicích 2.22 a 2.23 na stranách 64 a 66 jsme popisovali, jak probíhá aplikace těchto speciálních forem. V obou případech je pro nás důležitým rysem to, že tyto speciální formy vyhodnocují své argumenty sekvenčně jeden po druhém. Navíc platí, že vyhodnocení každého z argumentů je podmíněno výsledkem vyhodnocení předcházejícího argumentu. Například pokud se jeden z argumentů pro `and` vyhodnotí na „nepravda“, další argumenty již nebudou vyhodnocovány. Analogické situace platí pro `or` v případě, kdy se argument vyhodnotí na „pravda“, viz lekci 2.

  Jako příklad použití těchto speciálních forem k zastavení rekurze uveďme nejdříve predikát `list?`, který jsme definovali předchozím bodě pomocí speciální formy `if`:

```
(define list?
  (lambda (l)
    (or (null? l)
        (and (pair? l)
             (list? (cdr l))))))
```

  V těle λ-výrazu máme tedy použitu speciální formu `or`. Jejím prvním argumentem `(null? l)`. Po vyhodnocení tohoto prvního argumentu máme dvě možnosti. Buďto se vyhodnotil na pravdu (seznam navázaný na `l` je prázdný), a pak je pravda výsledkem aplikace speciální formy `or` a také výsledkem aplikace predikátu `list?`, nebo se vyhodnotil na nepravdu (seznam navázaný na `l` je neprázdný) a pak pokračujeme vyhodnocením dalšího argumentu. Tím je seznam

```
(and (pair? l) (list? (cdr l)))
```

  a protože se jedná o poslední argument, bude výsledek jeho vyhodnocení také výsledkem aplikace

speciální formy `or` a tedy i aplikace predikátu `list?`. Tato část kódu už je shodná s alternativní větví speciální formy `if` v programu uvedeném v předchozím bodě.

Dále můžeme napsat třeba efektivní verze predikátů `forall` a `exists`, které jsme poprvé definovali v lekci 6. Jejich definice už nebudeme podrobně popisovat, jedná se o podobný přepis jako v případě predikátu `list?`:

```
(define forall
  (lambda (f l)
    (or (null? l)
        (and (f (car l))
             (forall f (cdr l))))))

(define exists
  (lambda (f l)
    (and (not (null? l))
         (or (f (car l))
             (exists f (cdr l))))))
```

Na první pohled by se mohlo zdát, že zastavení rekurze využívající speciální formy `and` a `or` lze použít pouze při programování predikátů (procedur vracejících jako výsledky aplikace pravdivostní hodnoty). Jako protipříklad můžeme uvést definici procedury `fak` na výpočet faktoriálu:

```
(define fak
  (lambda (n)
    (or (and (= n 0) 1)
        (* n (fak (- n 1))))))
```

Zatímco u předchozích příkladů na zastavení rekurze pomocí speciálních forem `and` a `or`, které byly definicemi predikátů, je tato definice poněkud nepřehledná. Třeba v případě predikátu `list?` jsme mohli kód intuitivně číst jako: „Argument `l` je seznam, pokud je to prázdný seznam *nebo* je to pár *a současně* druhý prvek tohoto páru je zase seznam.“ U definice procedury `fak` toto provést nelze. Při programování je kromě samotné funkčnosti programu potřeba dbát i na jeho čitelnost a výše uvedené rekurzivní procedura počítající faktoriál, která zastavuje rekurzi pomocí `or` je spíš odstrašující příklad.

- *i když „zdánlivě nemá limitní podmínku“.*

  V některých případech je limitní podmínka rekurze v proceduře „skrytá“, to jest je obsažena až „hlouběji v těle“ procedury. Jako příklad uvažujme třeba proceduru `depth-map`, která pro zadanou proceduru a seznam, jehož prvky mohou být opětně seznamy, vytvoří seznam, který má stejnou strukturu jako původní seznam, ale jeho prvku jsou výsledky aplikace zadané procedury na původní prvky seznamu. Tedy například:

  ```
  (depth-map - '(1 (2 () (3)) (((4))))) ⟹ (-1 (-2 () (-3)) (((-4))))
  ```

  Definici takto popsané procedury bychom mohli napsat například takto:

  ```
  (define depth-map
    (lambda (f l)
      (map (lambda (x)
             (if (list? x)
                 (depth-map f x)
                 (f x)))
           l)))
  ```

  Procedura na každý prvek daného seznamu aplikuje buďto sama sebe nebo zadanou proceduru – to podle toho, jestli se jedná o seznam. K rekurzivní aplikaci „sebe sama“ tedy nedojde, pokud seznam nebude obsahovat další seznamy. Přímá formulace této limitní podmínko ovšem nikde v kódu není, ale je jaksi rozložena v použití procedury `map` a v jejím prvním argumentu. Dalším příkladům na tento typ limitní podmínky se budeme věnovat v sekci 9.5.

Jak již jsme v předchozí lekco naznačili, teoreticky máme rovněž možnost „nezastavovat rekurzi“, to jest psát rekurzivní procedury bez limitní podmínky. Zatím to ale nemá příliš velký smysl. Programy pak generují nekonečný výpočetní proces, který sestává z nekončící série aplikací jedné procedury. Třeba následující kód je definicí procedury, jejíž aplikace způsobí nekončící sérii aplikací:

```
(define loop-to-infinity
  (lambda ()
    (loop-to-infinity)))
```

Při pokusu aplikovat `loop-to-infinity` (bez argumentu) dojde k „zacyklení výpočtu“.

Již v lekci 1 jsme naznačili, že bychom si s konceptem „`if` jako procedura“ nevystačili. Hlavním důvodem k tomuto tvrzení je fakt, že `if` jako procedura *nezastaví rekurzi*. Příčinou by právě byl její procedurální charakter, který způsobuje, že před samotnou aplikací dochází k vyhodnocení všech předaných argumentů. Vyhodnoceny by tedy byly obě „větvě“ rekurzivní procedury (vžraz následující za limitní podmínkou i předpis rekurze) bez ohledu na výsledek vyhodnocení podmínky. Při použití „`if` jako procedury“ k zastavení rekurze ale jedna z větví obsahuje rekurzivní volaní procedury, a tak dochází k nekonečné smyčce aplikací této procedury. Viz následující příklad, který zachycuje pokus o vytvoření procedury na výpočet faktoriálu pomocí „`if` jako procedury“:

```
(define if-proc
  (lambda (condition expr altex)
    (if condition expr altex)))

(define fak-loop
  (lambda (n)
    (if-proc (= n 1)
             1
             (* n (fak-loop (- n 1))))))
```

Tímto způsobem definovaná procedura `fak-loop` bude vždy cyklit.

9.2 Rekurzivní procedury definované pomocí y-kombinátoru

V lekci 2 jsme vysvětlili vznik uživatelských procedur. Uživatelské procedury vznikají vyhodnocováním λ--výrazů a každou uživatelskou proceduru lze chápat jako trojici hodnot: seznam argumentů, tělo, a prostředí vzniku. Připomeňme, že na vzniku procedur se nijak nepodílí speciální forma `define`. V této a předchozí lekci jsme vytvářeli procedury, které ve svém těle aplikovaly samy sebe. Tento typ „sebeaplikace“ bylo možné provést, protože symbol, na který byla procedura navázána pomocí `define`, se nacházel v prostředí jejího vzniku. Na první pohled se tedy může zdát, že v případě rekurzivních procedur hraje `define` významnou roli. Tato role sice nespočívá v samotném vytvoření procedury (tu má pořád na starost speciální forma `lambda`), ale v umožnění odkázat se z těla procedury na sebe sama prostřednictvím hodnoty navázané na symbol (jméno procedury) pomocí `define`. V této kapitole ukážeme, že `define` není z pohledu aplikace rekurzivních procedur potřeba, což může být jistě pro řadu čtenářů překvapující závěr.

Procedury vytvořené vyhodnocením λ-výrazů mohou být přímo aplikovány s danými argumenty. Například vyhodnocení výrazu `((lambda (x) (* 2 x)) 10)` vede na jednorázovou aplikaci procedury jež vynásobí svůj argument s dvojkou. Otázkou je, zda-li jsme schopni definovat „jednorázovou rekurzivní proceduru“, aniž bychom provedli definici vazby pomocí speciální formy `define`. Je zřejmé, že pokud má být daná procedura rekurzivní, musí mít k dispozici „sebe sama“ prostřednictvím vazby některého symbolu. Na první pohled by se tedy použití `define` mohlo zdát jako nevyhnutelné.

Našim cílem tedy bude vytvořit pojmenovanou rekurzivní proceduru bez použití `define`. Nejprve si ukažme, kudy cesta nevede (a proč). Jako modelovou proceduru si vezmeme rekurzivní proceduru na

výpočet faktoriálu. Jelikož je naším úkolem tuto proceduru vytvořit jako pojmenovanou, leckoho možná napadne „pouze nahradit `define` pomocí `let`“ následujícím způsobem:

```
(let ((fak (lambda (n)
             (if (= n 0)
                 1
                 (* n (fak (- n 1)))))))
  (fak 6))
```

Vyhodnocení předchozího kódy však končí chybou, jejíž vznik by nám měl být v tuto chvíli jasný. Předchozí kód je totiž ekvivalentní programu:

```
((lambda (fak)
   (fak 6))
 (lambda (n)
   (if (= n 0)
       1
       (* n (fak (- n 1))))))
```

Symbol `fak`, který se nachází v těle procedury vzniklé vyhodnocením λ-výrazu `(lambda (n) ···)` zřejmě nemá žádný vztah k symbol `fak`, který je vázaný v λ-výrazu `(lambda (fak) ···)`. Uvědomte si, že procedura vytvořená vyhodnocením λ-výrazu `(lambda (n) ···)` vznikla v globálním prostředí (kde `fak` nemá vazbu). Z naprosto stejného důvodu by chybou skončil i následující program, který se od předchozího liší použitím `let*` místo `let` (vzhledem k tomu, že v případě vytvoření jedné vazby se `let*` a `let` shodují se navíc jedná o týž program jako v předchozím případě).

```
(let* ((fak (lambda (n)
              (if (= n 0)
                  1
                  (* n (fak (- n 1)))))))
  (fak 6))
```

Problém zavedení rekurzivní procedury bez `define` vyřešíme tak, že uvažované rekurzivní proceduře předáme sebe sama *prostřednictvím nového argumentu*. Například v případě procedury pro výpočet faktoriálu by situace vypadala následovně:

```
(lambda (fak n)
  (if (= n 0)
      1
      (* n (fak fak (- n 1)))))
```

Všimněte si, že pokud proceduru vytvořenou vyhodnocením předchozího λ-výrazu aplikujeme s prvním argumentem jímž bude *ta sama procedura*, pak bude zcela legitimně probíhat rekurzivní volání, protože v těle procedury bude na symbol `fak`, který je nyní jedním z argumentů, navázána právě volaná procedura. Pochopitelně, při rekurzivním volání musí být procedura opět předávána, což se v kódu promítlo do tvaru volání `(fak fak (- n 1))`. Nyní tedy zbývá vyřešit poslední problém, jak zavolat předchozí proceduru tak, aby na svém prvním argumentu byla tatáž procedura navázána. K vyřešení tohoto problému použijeme takzvaný y-kombinátor. Předtím než jej obecně popíšeme si všimněte, že pokud se nám podaří následující proceduru

```
(lambda (y)
  (y y 6))
```

aplikovat s argumentem jímž bude procedura vzniklá vyhodnocením `(lambda (fak n) ···)`, pak v těle procedury vzniklé vyhodnocením `(lambda (y) (y y 6))` proběhne aplikace procedury vzniklé vyhodnocením `(lambda (fak n) ···)`, přitom na prvním argumentu bude navázána právě aplikovaná procedura a druhým argumentem bude číslo `6`, což přesně povede na požadovaný rekurzivní výpočet. Výše popsanou aplikaci však můžeme provést jednoduše spojením předchozích dvou částí dohromady tak, jak to ukazuje program 9.1.

Program 9.1. Rekurzivní procedura pro výpočet faktoriálu aplikovaná pomocí y-kombinátoru.

```
((lambda (y)
   (y y 6))
 (lambda (fak n)
   (if (= n 0)
       1
       (* n (fak fak (- n 1))))))
```

Kód v programu 9.1 tedy způsobí aplikaci rekurzivní procedury pro výpočet faktoriálu jejímž prvním argumentem je samotná procedura pro výpočet faktoriálu a druhým argumentem je hodnota, pro kterou chceme faktoriál počítat (v tomto konkrétním případě hodnota `6`). Zápis předchozího kódy bychom mohli zjednodušit pomocí speciální formy `let` následovně:

```
(let ((y (lambda (fak n)
           (if (= n 0)
               1
               (* n (fak fak (- n 1)))))))
  (y y 6))
```

Principiálně se ale jedná o totéž jako v předchozím případě.

Nyní můžeme obecně popsat y-kombinátor a jeho použití při zavedení rekurzivních procedur. Pod pojmem y-kombinátor máme na mysli právě λ-výraz v následujícím tvaru:

```
(lambda (y)
  (y y ⟨argument_1⟩ ⟨argument_2⟩ ··· ⟨argument_n⟩))
```

Argumenty $\langle argument_1\rangle \cdots \langle argument_n\rangle$ v předchozím výrazu reprezentují argumenty, které chceme předat volané (rekurzivní) proceduře navázané na symbol `y`. Prvním předaným argumentem je ovšem hodnota navázaná na `y`, tedy samotná procedura. Proceduru vzniklou vyhodnocením předchozího λ-výrazu (y--kombinátoru) zavoláme s jediným argumentem jímž bude procedura $n+1$ argumentů. y-kombinátor je tedy část programu, která je odpovědná za *aplikaci rekurzivní procedury*, konkrétně za aplikaci procedury spojené s předáním prvního argumentu jímž je sama procedura. Následující program demonstruje použití y-kombinátoru při aplikaci rekurzivní procedury dvou argumentů (spojení dvou seznamů):

```
((lambda (y)
   (y y list-a list-b))
 (lambda (append2 s1 s2)
   (if (null? s1)
       s2
       (cons (car s1) (append2 append2 (cdr s1) s2)))))
```

Předchozí kód provede spojení seznamů navázaných na symbolech `list-a` a `list-b`.

V předchozích příkladech jsme pomocí y-kombinátoru provedli vždy jen jednu aplikaci procedury. Nic ale nebrání tomu, abychom rekurzivní proceduru vytvořenou pomocí y-kombinátoru lokálně pojmenovali prostřednictvím vazby vytvořené speciální formou `let` a pak ji použili opakovaně. Viz následující příklad:

```
(let ((append2
       (lambda (list-a list-b)
         (let ((y (lambda (append2 s1 s2)
                    (if (null? s1)
                        s2
                        (cons (car s1)
                              (append2 append2 (cdr s1) s2))))))
           (y y list-a list-b)))))
```

```
  ... výrazy ...
(append2 '(1 2 3) '(a b c))
  ... výrazy ...)
```

Lokální definicí rekurzivních procedur se budeme zabývat i v další části této lekce. Praktické použití y-kombinátorů ukážeme v lekci 12.

Pozorní čtenáři si jistě pamatují, že v sekci 2.5 jsme uvedli následující kód `((lambda (y) (y y)) (lambda (x) (x x)))`, který rovněž způsobí nekončící sérii aplikací téže procedury. V druhé lekci, kde byl tento kód poprvé uveden, jsme si celou situaci možná nebyli schopni zcela představit. Nyní se ale na výše uvedený kód můžeme dívat jako na kód, ve kterém je použit y-kombinátor pro rekurzivní aplikaci procedury bez argumentu. Ve skutečnosti tedy použitá proceduru „jeden argument má“, díky němuž má k dispozici sebe sama.

9.3 Lokální definice rekurzivních procedur

V předchozí sekci jsme ukázali, že pomocí speciálních forem `let` a `let*` nemůžeme lokálně „definovat“ rekurzivní procedury. Můžeme však běžným způsobem použít interní definice. Jako příklad uveďme proceduru na výpočet kombinačního čísla $\binom{n}{k}$ podle vzorce

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Proceduru pro výpočet faktoriálu ale budeme definovat lokálně. Mohli bychom ji například napsat takto:

```
(define comb
  (lambda (n k)
    (define fak (lambda (n)
                  (if (<= n 1) 1 (* n (fak (- n 1))))))
    (/ (fak n) (fak k) (fak (- n k)))))
```

Uvedená definice má ale jeden závažný nedostatek. Tímto nedostatkem je skutečnost, že při každém volání procedury `comb` vytváříme stále znovu proceduru pro výpočet faktoriálu. To je zbytečné, protože vytvoření této procedury je nezávislé na argumentech procedury `comb`. To můžeme snadno vyřešit tak, že nebudeme proceduru `fak` definovat v těle procedury `comb`, ale proceduru `comb` vytvoříme až lokálním prostředí ve kterém bude definována procedura `fak`. Kód definice procedury `comb` by pak vypadal takto:

```
(define comb
  (let ()
    (define fak (lambda (n)
                  (if (<= n 1) 1 (* n (fak (- n 1))))))
    (lambda (n k)
      (/ (fak n) (fak k) (fak (- n k))))))
```

Podobně bychom mohli proceduru `comb` napsat pomocí další varianty speciální formy `let`, a to speciální formy `letrec`. Nyní uvedeme definici procedury `comb` napsanou s použitím této speciální formy a vzápětí tuto speciální formu podrobně popíšeme.

```
(define comb
  (letrec ((fak (lambda (n)
                  (if (<= n 1) 1 (* n (fak (- n 1)))))))
    (lambda (n k)
      (/ (fak n) (fak k) (fak (- n k))))))
```

Teď tedy k speciální formě `letrec`:

Definice 9.1 (Speciální forma `letrec`). Speciální forma `letrec` se používá ve stejném tvaru jako speciální forma `let*`, tedy ve tvaru

```
(letrec ((⟨symbol_1⟩ ⟨element_1⟩)
         (⟨symbol_2⟩ ⟨element_2⟩)
          ⋮
         (⟨symbol_n⟩ ⟨element_n⟩))
  ⟨výraz_1⟩
  ⟨výraz_2⟩
   ⋮
  ⟨výraz_m⟩),
```

kde n je nezáporné číslo a m je přirozené číslo; $\langle symbol_1\rangle$; $\langle symbol_2\rangle$, ... $\langle symbol_n\rangle$ jsou symboly; $\langle element_1\rangle$, $\langle element_2\rangle$, ... $\langle element_n\rangle$ a $\langle výraz_1\rangle$, $\langle výraz_2\rangle$, ... $\langle výraz_m\rangle$ jsou libovolné výrazy. Celý výraz nazýváme `letrec`-blok, výrazy $\langle výraz_1\rangle$, $\langle výraz_2\rangle$, ... $\langle výraz_m\rangle$ nazýváme souhrnně tělem `letrec`-bloku.

`letrec`-blok vyhodnocuje stejným způsobem jako výraz

```
(let ((⟨symbol_1⟩ undefined)
      (⟨symbol_2⟩ undefined)
       ⋮
      (⟨symbol_n⟩ undefined))

  (define ⟨symbol_1⟩ ⟨element_1⟩)
  (define ⟨symbol_2⟩ ⟨element_2⟩)
    ⋮
  (define ⟨symbol_n⟩ ⟨element_n⟩)
  ⟨výraz_1⟩
  ⟨výraz_2⟩
   ⋮
  ⟨výraz_m⟩),
```

kde *undefined* je speciální element jazyka zastupující „*nedefinovanou hodnotu*“. ■

Příklad 9.2. `letrec`-blok ve tvaru

```
(letrec ((x 10)
         (y (+ x 2)))
  (list x y))  ⟹  (10 12)
```

se přepisuje na následující výraz, který je vyhodnocen:

```
(let ((x undefined)
      (y undefined))
  (define x 10)
  (define y (+ x 2))
  (list x y))  ⟹  (10 12)
```

Poznámka 9.3. (a) Standard R^{5}RS jazyka Scheme přesně nezavádí, jak chápat nedefinovanou hodnotu. Nedefinovaná hodnota, který hraje roli ve speciální formě `letrec` je ve většině interpretů jazyka Scheme zastoupena speciálním elementem jazyka. Obvykle je tento element odlišný od elementu „nespecifikovaná hodnota“, který jsme zavedli již v první lekci (připomeňme, že například vyhodnocením `(if #f #f)` získáme element zastupující nespecifikovanou hodnotu). Vzhledem ke způsobu jakým se přepisují `letrec`-bloky, můžeme napsat výraz, jehož výsledek vyhodnocení bude právě nedefinovaná hodnota použitá jako počáteční vazba symbolů v `letrec`-blocích:

```
(letrec ((x x)) x)  ⟹  undefined
```

(b) Speciální forma `letrec` je stejně jako ostatní dříve uvedené varianty speciální formy `let` nadbytečná v tom smyslu, že jsou nahraditelné jiným kódem.

(c) Na rozdíl od použití speciální formy a `let*` je možné odkazovat se ve $\langle element_1 \rangle$, $\langle element_2 \rangle$, ... $\langle element_n \rangle$ nejen „dozadu“ (na vazby definované výše) ale i „dopředu“. Například následující kód bude fungovat:

```
(letrec ((x y)
         (y 10))
  (list x y))  ⟹  (undefined 10)
```

Výsledkem jeho vyhodnocení je seznam `(`*undefined* `10)`. Uvedený kód se totiž přepíše na `let`-blok

```
(let ((x undefined)
      (y undefined))
  (define x y)
  (define y 10)
  (list x y))  ⟹  (undefined 10)
```

Během definice `(define x y)` je na symbol `y` navázána nedefinovaná hodnota. Proto i po vyhodnocení této definice bude na symbol `x` navázána nedefinovaná hodnota. Až poté je na symbol `y` navázáno číslo `10`. Tělo `(list x y)` původního `letrec`-bloku je tedy vyhodnoceno na výraz `(`*undefined* `10)`.

9.4 Implementace vybraných rekurzivních procedur

V předchozích lekcích jsme ukázali implementace mnoha procedur pracujících se seznamy. Nyní se k nim vrátíme a naimplementujeme je s použitím rekurze. Nejdříve ale uvedeme implementace samotných akumulačních procedur `foldr` a `foldl` (pro jeden seznam):

```
(define foldr
  (lambda (f basis l)
    (if (null? l)
        basis
        (f (car l) (foldr f basis (cdr l))))))
```

Jedná se tedy o rekurzivní proceduru, jejíž limitní podmínkou je prázdnost seznamu. Pokud je tato splněna vrací se terminátor `basis`. Jinak je aplikována procedura `f` na první prvek seznamu a výsledek rekurzivního volání procedury `foldr` se zkráceným seznamem. Nyní předejme k definici procedury `foldl`:

```
(define foldl
  (lambda (f basis l)
    (let iter ((l l)
               (accum basis))
      (if (null? l)
          accum
          (iter (cdr l)
                (f (car l) accum))))))
```

V těle této procedury definujeme a aplikujeme pomocnou proceduru `iter`. Tato iterativní procedura přijímá dva argumenty – seznam doposud nezpracovaných prvků `l` (na začátku celý seznam) a výsledek akumulace zpracovaných prvků `accum` (na začátku terminátor `basis`). Pokud je seznam `l` prázdný, je vrácen `accum`, jinak voláme proceduru `iter` se zkráceným seznamem `l` a s nabalením jeho prvního prvku na argument `accum`.

V lekci 6 jsme implementovali mnoho procedur pomocí procedur pro akumulaci na seznamech `foldr` a `foldl`. Se znalostí toho, jak jsou tyto dvě procedury implementovány, je velmi snadné přepsat ukázkové procedury z lekce 6 na rekurzivní procedury, které ve svém těle nevolají akumulační procedury. Přepis je přitom v podstatě mechanickou záležitostí. V této sekci proto uvedeme jen tři příklady, a to proceduru

append na spojování (libovolného množství) seznamů, procedura mapování procedury přes libovolný počet seznamů map a procedura compose na skládání (libovolného množství) funkcí.

Nejprve tedy uvedeme definici rekurzivní procedury append. Jako první nadefinujeme proceduru append2 na spojení dvou seznamů a pomocí ní napíšeme rozšíření na libovolný počet argumentů. Viz program 9.3. Procedura append zastavuje rekurzi v případě, že je počet jejich argumentů nulový nebo jednotkový. Pak je spojení triviální. Jinak dva argumenty spojíme pomocnou procedurou append2 a aplikujeme pak append na menší počet argumentů.

Program 9.2. Spojovaní seznamů append naprogramované rekurzivně

```
(define append2
  (lambda (l1 l2)
    (if (null? l1) l2
        (cons (car l1) (append2 (cdr l1) l2)))))

(define append
  (lambda lists
    (cond ((null? lists) '())
          ((null? (cdr lists)) (car lists))
          (else (apply append
                       (append2 (car lists) (cadr lists))
                       (cddr lists))))))
```

Proceduru append bychom mohli napsat i bez použití pomocné procedury pro spojení dvou seznamů append2. Viz program 9.3. V něm postupně opětně zastavujeme rekurzi v případě, že je seznam seznamů určených ke spojení prázdný nebo jednoprvkový. Problém pak rozkládáme na spojení méně seznamů (pokud je první seznam prázdný), nebo na spojení stejného počtu seznamů, s tím, že první z nich je zkrácený o jeden prvek (a tedy strukturálně jednodušší) a připojení prvku na začátek seznamu.

Program 9.3. Spojovaní seznamů append naprogramované rekurzivně bez použití pomocné procedury

```
(define append
  (lambda lists
    (cond ((null? lists) '())
          ((null? (car lists)) (apply append (cdr lists)))
          (else (cons (caar lists)
                      (apply append
                             (cons (cdar lists) (cdr lists))))))))
```

V předchozí lekci jsme ukázali implementaci rekurzivní procedury map1, (viz program 8.18). Nyní pomocí ní naimplementujeme proceduru mapování procedury na libovolný počet seznamů map. Následuje kód této procedury:

```
(define map
  (lambda (f . lists)
    (if (null? (car lists))
        '()
        (cons (apply f (map1 car lists))
              (apply map (cons f (map1 cdr lists)))))))
```

Poslední procedurou, kterou v této sekci napíšeme, je procedura na skládání libovolného počtu funkcí (procedur reprezentujících zobrazení). Takovou proceduru jsme již implementovali v programu 7.10 na

straně 182 pomocí akumulační procedury `foldl`. Nyní uděláme totéž bez ní. Viz program 9.4.

Program 9.4. Skládání funkcí `compose` bez použití procedury `foldl`

```
(define compose
  (lambda functions
    (let iter ((flist functions)
               (f (lambda (x) x)))
      (if (null? flist)
          f
          (iter (cdr flist)
                (lambda (x)
                  ((car flist) (f x))))))))
```

V programu 9.4 tedy pomocí pojmenovaného `let` definujeme a aplikujeme pomocnou proceduru `iter` dvou argumentů. První argumentem je seznam funkcí ke skládání a druhý je akumulační argument, kde si pamatujeme prozatímní složení funkcí (na začátku identita) Tato procedura je rekurzivní a při splnění limitní podmínky, kterou je test prázdnosti seznamu skládaných funkcí, akumulační argument. Při nesplnění limitní podmínky je rekurzivně volána procedura `iter` se seznamem funkcí bez prvního prvku a složení akumulačního argumentu s první funkcí ze seznam funkcí. Viz následující aplikace této procedury:

```
(define s '(0 1 2 3 4))
(map (compose) s)                                                  ⟼  (0 1 2 3 4)
(map (compose (lambda (x) (* 2 x))) s)                             ⟼  (0 2 4 6 8)
(map (compose (lambda (x) (* 2 x)) (lambda (x) (* x x))) s)        ⟼  (0 4 16 36 64)
(map (compose (lambda (x) (* x x)) (lambda (x) (* 2 x))) s)        ⟼  (0 2 8 18 32)
```

9.5 Hloubková rekurze na seznamech

Doposud jsme více méně zpracovávali seznamy „prvek po prvku“. V této sekci se budeme zabývat implementací rekurzivních procedur, které při zpracování seznamu budou aplikovat samy sebe na ty prvky seznamů, které jsou opět seznamy. Typická úloha patřící do tohoto druhu úloh je například spočítání atomických prvků (to jest těch prvků celé struktury, které nejsou seznamy). Dále třeba linearizace seznamu, tedy vytvoření seznamu atomických prvků.

Nyní napíšeme a rozebereme možnou implementaci uvedených úloh (počet atomů a linearizace), poté se zaměříme na podobnost obou definic a navrhneme zobecnění. Definici procedury linearizace seznamu `linearize` provedeme takto:

```
(define linearize
  (lambda (l)
    (cond ((null? l) '())
          ((list? (car l)) (append (linearize (car l)) (linearize (cdr l))))
          (else (cons (car l) (linearize (cdr l)))))))
```

V této rekurzivní proceduře je limitní podmínkou rekurze prázdnost zpracovávaného seznamu. Je-li této podmínky dosaženo, vrací procedura prázdný seznam. V opačném případě (tedy pokud seznam není prázdný), zkoumáme jeho první prvek. Jestliže je tento prvek opět seznam, zlinearizujeme jej procedurou `linearize` a procedurou `append` ji připojíme ke zlinearizovanému zbytku seznamu. Je-li první prvek atomický, přidáme jej k linearizaci zbytku seznamu. Tímto postupem vytvoříme lineární seznam, který obsahuje všechny atomické prvky z původního seznamu. Viz ukázky použití:

```
(linearize '())                    ⟼  ()
(linearize '(1))                   ⟼  (1)
```

```
(linearize '((1)))                    ⟼  (1)
(linearize '(1 ((2)))                 ⟼  (1 2)
(linearize '(1 ((2)) () (3 (4) 5)))   ⟼  (1 2 3 4 5)
```

Teď se zaměřme na druhou uvedenou typickou úlohu, kterou je zjištění počtu atomických prvků v dané hierarchické struktuře. Následuje kód implementace této procedury.

```
(define atoms
  (lambda (l)
    (cond ((null? l) 0)
          ((list? (car l)) (+ (atoms (car l)) (atoms (cdr l))))
          (else (+ 1 (atoms (cdr l)))))))
```

Limitní podmínku tvoří, stejně jako v předchozí proceduře, test na prázdnost seznamu. Pokud je tedy seznam prázdný, je vrácena nula. Jinak se zajímáme o to, jestli je první prvek tohoto seznamu opětně seznam. Jestliže ano, sečteme jeho atomy a atomy ve zbytku seznamu. Jestliže první prvek není seznam, a tedy je to atomický prvek, přičteme za něj jedničku k počtu atomů ve zbytku seznamu. Viz příklady aplikace této procedury:

```
(atoms '())                           ⟼  0
(atoms '(1))                          ⟼  1
(atoms '((a)))                        ⟼  1
(atoms '(1 ((2)))                     ⟼  2
(atoms '(1 () 2 (a (() (b (c))) (d) e)))  ⟼  7
```

Pozorný čtenář si jistě povšiml nemalé podobnosti uvedených definic. V obou případech zastavujeme rekurzi v případě prázdného seznamu. Když byl seznam neprázdný, zajímalo nás, jestli byl jeho první prvek opět seznam nebo ne. V případě seznamu jsme rekurzivně volali proceduru pro tento první prvek a také pro zbytek seznamu. Výsledek této aplikace jsme pak zkombinovali pomocí další procedury (`append`, `+`). V případě atomického prvku jsme tento prvek zpracovali, a zkombinovali jej (vlastně stejnou procedurou jako v předchozí větvi) s výsledkem rekurzivního vyvolání procedury na zbytek seznamu. Konkrétně u procedury `atoms` jsme zpracovali atomický prvek tak, že jsme jej zaměnili za jedničku, tu jsme pak sečetli aplikací procedury `+`. U procedury `linearize` jsme uvedli kód:

```
(else (cons (car l) (linearize (cdr l))))
```

Stejný význam by měl kód, napsaný takto:

```
(else (append (list (car l)) (linearize (cdr l)))).
```

Zpracování prvku je tedy v tomto případě vytvoření jednoprvkového seznamu aplikací procedury `list` a procedurou kombinace je procedura `append`. Tato podobnost nám umožňuje vytvořit zobecnění těchto procedur – hloubkovou akumulační proceduru `depth-accum`.

```
(define depth-accum
  (lambda (combine nil modifier l)
    (cond ((null? l) nil)
          ((list? (car l)) (combine (depth-accum combine nil modifier (car l))
                                    (depth-accum combine nil modifier (cdr l))))
          (else (combine (modifier (car l))
                         (depth-accum combine nil modifier (cdr l)))))))
```

Sjednotili jsme tedy předchozí dvě procedury do jedné obecnější. Rozdíly v procedurách jsme zahrnuli do dalších argumentů této procedury: `combine` pro různé způsoby kombinace výsledků z rekurzivních volání (popř. ze zpracování atomických prvků), `nil` pro různé návratové hodnoty při splnění limitní podmínky a `modifier` pro různá zpracování atomických prvků. Například aplikace procedur `atoms` a `linearize` bychom mohli nahradit aplikací procedury `depth-accum` takto:

```
(depth-accum + 0 (lambda (x) 1) '(a (b c (d)) e))   ⟼  5
(depth-accum append '() list '(a (b c (d)) e))      ⟼  (a b c d e)
```

Pomocí procedury `depth-accum` samozřejmě můžeme definovat další hloubkově rekurzivní procedury. Teď uvedeme několik dalších procedur tohoto typu, k jejich vytvoření použijeme právě proceduru `depth-accum`. První z nich bude procedura `depth-count` zjišťující (hloubkově) počet výskytů daného atomického prvku v zadané struktuře. Viz její definici:

```
(define depth-count
  (lambda (atom l)
    (depth-accum + 0 (lambda (x)
                       (if (equal? atom x)
                           1 0))
                 l)))
```

Terminátorem je v tomto případě číslo nula a kombinační procedurou je procedura sčítání `+`. To je vlastně stejné jako u implementace procedury `atoms`. Na rozdíl od procedury `atoms` ale nezpracováváme každý atom tak, že jej zaměňujeme za číslo jedna, ale buďto za číslo jedna nebo za číslo nula v závislosti na tom, jestli je tento atom stejný jako zadaný element. Pro porovnání elementů jsme použili proceduru `equal?`. Následují ukázky použití takto vytvořené procedury.

```
(depth-count 1 '())                                  ⟹  0
(depth-count 1 '(1))                                 ⟹  1
(depth-count 1 '((1)))                               ⟹  1
(depth-count 1 '(1 (1)))                             ⟹  2
(depth-count 'a '(1 () 2 (a (() (b (a))) (d) e)))    ⟹  2
```

Dále vytvoříme predikát `depth-find`, který zjišťuje, zda je daný atom přítomen ve struktuře. K její implementaci opět použijeme proceduru hloubkové akumulace na seznamu `depth-accum`. Predikát `depth-find` můžeme implementovat náhledově:

```
(define depth-find
  (lambda (x l)
    (depth-accum (lambda (x y) (or x y))
                 #f
                 (lambda (y) (equal? x y))
                 l)))
```

Proceduru `depth-accum` jsme tedy použili těmito argumenty: procedura napodobující speciální formu `or` pro dva argumenty (taktéž bychom zde místo vytváření nové procedury mohli využít proceduru `or-proc`, se kterou jsme se již setkali v lekci 6). Terminátorem je pravdivostní hodnota nepravda a zpracováním atomického prvku zde rozumíme vrácení pravdivostní hodnoty podle toho, zda je shodný s hledaným elementem. Následují ukázky aplikace této procedury.

```
(depth-find 'a '(1 () 2 (a (() (b (a))) (d) e)))  ⟹  #t
(depth-find 'b '(1 () 2 (a (() (b (a))) (d) e)))  ⟹  #t
(depth-find 'x '(1 () 2 (a (() (b (a))) (d) e)))  ⟹  #f
```

Posledním užitím procedury `depth-accum` (ve stávající podobě), kterou si v této sekci ukážeme bude procedura hloubkové filtrace atomů daných vlastnosti, které zachovává hloubkovou strukturu seznamu. Pro jeho definici viz program 9.5. Terminátorem je zde prázdný seznam, atomické prvky zpracováváme tak, že pokud splňují zadaný predikát `x`, aplikujeme na ně modifikátor `modifier`, jinak je ponecháváme stejné. Procedurou pro kombinaci výsledků je pak konstruktor páru `cons`. Viz příklad aplikace:

```
(define s '(1 () 2 (a (() (b (a))) (d) e)))
(depth-replace number? - s)                ⟹  (-1 () -2 (a (() (b (a))) (d) e))
(depth-replace symbol? (lambda (x) #f) s)  ⟹  (1 () 2 (#f (() (#f (#f))) (#f) #f))
```

Procedury `linearize` a `atoms`, jejichž definice jsme uvedli na začátku této sekce, bychom samozřejmě mohli napsat i jinak. Například použití procedur `apply` a `map` nám umožňuje znatelně zkrátit kód těchto procedur. Definice procedury `linearize` by vypadalo takto:

Program 9.5. Implementace procedury hloubkového nahrazovaní atomů `depth-replace`

```
(define depth-replace
  (lambda (prop? modifier l)
    (depth-accum cons
                 '()
                 (lambda (z)
                   (if (prop? z)
                       (modifier z)
                       z))
                 l)))
```

```
(define linearize
  (lambda (l)
    (if (list? l)
        (apply append (map linearize l))
        (list l))))
```

Procedura `linearize` nejdříve zkontroluje, zda je její argument seznam. Jestliže ne, vytvoříme jednoprvkový seznam obsahující tento prvek. V případě, že ano, pomocí procedury `map` aplikuje sama sebe na každý jeho prvek – dostáváme tak seznam, jehož prvky jsou lineární seznamy (buďto zlinearizované seznamy z původního seznamu nebo jednoprvkové seznamy vytvořené z atomických prvků). Tyto seznamy spojíme v jeden aplikací procedury `append`.

Obdobným způsobem napíšeme proceduru `atoms`:

```
(define atoms
  (lambda (l)
    (if (list? l)
        (apply + (map atoms l))
        1)))
```

Stejně jako v definici procedury `linearize` jsme nejdříve zjistili, zda je argumentem procedury seznam. Pokud ne, vracíme číslo `1`. Jinak aplikujeme na každý prvek tohoto seznamu pomocí procedury `map`. Výsledný seznam čísel pak sečteme aplikací procedury sčítání.

Nyní se můžeme zaměřit na zobecnění inspirované těmito dvěma definicemi. Podruhé naprogramujeme proceduru `depth-accum` zobecňující procedury `atoms` a `linearize` tímto způsobem:

```
(define depth-accum
  (lambda (combine modifier l)
    (if (list? l)
        (apply combine (map (lambda (x) (depth-accum combine modifier x)) l))
        (modifier l))))
```

Prvky, ve kterých je kódy procedur `atoms` a `linearize` lišily jsme shrnuli do argumentů `combine` a `modifier`. V porovnání s předchozím řešením nám tedy odpadl terminátor `nil`.

Funkci předchozích procedur `atoms` a `linearize` bychom mohli vyjádřit pomocí procedury `depth-accum` tak jak ukazují následující příklady:

```
(depth-accum append list '(a (b c (d)) e))
(depth-accum + (lambda (x) 1) '(a (b c (d)) e))
```

Takto definovaná procedura `depth-accum` má ale dva nedostatky. Procedura pro kombinaci prvků musí být procedurou libovolného počtu argumentů. To proto, že ji aplikujeme pomocí procedury `apply` na seznam jehož délku předem neznáme. Druhým nedostatkem je skutečnost že procedura `depth-accum` funguje i když jejím posledním argumentem nebude seznam.

Zatímco s druhým nedostatkem bychom se mohli docela klidně smířit, první nedostatek nám nové řešení jaksi degraduje v porovnání s předchozím. Například nemůžeme přímo přepsat proceduru `depth-replace` z programu 9.5, protože tam jsme ke kombinaci používali proceduru `cons`, která přijímá přesně dva argumenty. Tento nedostatek bychom mohli odstranit například tak, že bychom z libovolné monoidální operace vytvářeli příslušnou operaci libovolného počtu argumentů. To bychom mohli provádět třeba pomocí takto nadefinované procedury:

```
(define arbitrary
  (lambda (f nil)
    (lambda l
      (foldr f nil l))))
```

Takto můžeme pomocí `depth-accum` přepsat proceduru `depth-replace`, kterou jsme definovali v programu 9.5. V kódu tak z procedury `cons` vytvoříme proceduru libovolného množství argumentů. K tomu ale potřebujeme doplnit neutrální prvek. Ta hraje v podstatě stejnou úlohu jako terminátor v předchozí verzi. Takže se tak připravujeme o výhodu menšího počtu argumentů.

```
(define depth-replace
  (lambda (prop? modifier l)
    (depth-accum (arbitrary cons '())
                 (lambda (z) (if (prop? z) (modifier z) z))
                 l)))
```

V závěru této sekce se podíváme na praktické využití hloubkové rekurze. Představme si, máme tabulku naplněnou číselnými údaji a že máme zpracovávat výrazy zadané uživatelem. Tyto uživatelem zadané výrazy obsahují *odkazy do datové tabulky*, což jsou páry ve tvaru `(`*řádek* `.` *sloupec*`)`. Pro přesnější představu uveďme příklad takové tabulky a příklad takového výrazu:

```
(define tab
  '((1 0 1 0 0 1 1 0 3 0)
    (0 2 2 1 0 1 0 4 0 5)
    (2 1 0 1 5 0 2 1 3 1)
    (3 0 2 1 5 0 4 1 1 1)
    (2 1 2 0 5 1 3 1 1 2)
    (0 1 3 0 0 0 0 1 1 2)))

(+ 1 (3 . 2) (* 2 (2 . 4)))
```

Vyhodnocení uvedeného výrazu přirozeně skončí chybou. Naším prvním záměrem tedy bude proceduru `query->sexpr` nahrazující tyto páry ve tvaru `(`*řádek* `.` *sloupec*`)` seznamy `(table-ref` *řádek* *sloupec*`)`. K tomu použijeme proceduru `depth-replace`, kterou jsme definovali v programu 9.5.

```
(define query->sexpr
  (lambda (expr)
    (depth-replace pair?
                   (lambda (x)
                     (let ((row (car x))
                           (col (cdr x)))
                       (list 'table-ref row col)))
                   expr)))
```

Viz příklad použití:

```
(query->sexpr '(+ 1 (3 . 2) (* 2 (2 . 4))))
  ⟹  (+ 1 (table-ref 3 2) (* 2 (table-ref 2 4)))
```

Proceduru `query->sexpr` teď využijeme k vytvoření procedury vyššího řádu `query->proc`, která přijímá jeden argument, kterým je procedura, která je použita jako procedura dereference do datové tabulky. Provedeme to tak, že výraz, který je vrácen procedurou `query->sexpr` „obalíme“ kontextem

```
(lambda (table-ref) ···)
```

a takto vzniklý λ-výraz vyhodnotíme procedurou `eval`. Viz následující kód:

```
(define query->proc
  (lambda (expr)
    (eval (list 'lambda '(table-ref) (query->sexpr expr)))))
```

Zde uvádíme příklad aplikace:

```
((query->proc '(+ 1 (3 . 2) (* 2 (2 . 4))))
 (lambda (row col) (list-ref (list-ref tab row) col)))  ⟼  13
```

Pomocí této procedury `query->proc` můžeme definovat proceduru pro vyhodnocení výrazu s odkazy do tabulky vzhledem ke konkrétní tabulce. Procedura `eval-in-table` bude brát dva argumenty: výraz a tabulku. Ve svém těle pak aplikací procedury `query->proc` vytvoří proceduru (viz výše), kterou aplikuje na proceduru pro přístup k údaji v tabulce. Procedura pro přístup je přitom jen dvojnásobná aplikace procedury `list-ref`. Viz definici procedury `eval-in-table` a po ní následující příklady použití:

```
(define eval-in-table
  (lambda (expr table)
    ((query->proc expr)
     (lambda (row col)
       (list-ref (list-ref table row) col)))))

(eval-in-table '(+ 1 2) tab)                    ⟼  3
(eval-in-table '(+ 1 (1 . 7)) tab)              ⟼  5
(eval-in-table '(+ 1 (3 . 2) (* 2 (2 . 4))) tab)  ⟼  13
```

Proceduru `table-ref` přitom můžeme realizovat různými způsoby. Dokonce můžeme opustit původně zamýšlený formát tabulky, jako seznam seznamů, pracovat například s lineárním seznamem. Tuto variantu zachycuje následující program:

```
(define eval-in-linear-list
  (lambda (expr l cols)
    ((query->proc expr)
     (lambda (row col)
       (list-ref l (+ (* cols row) col))))))

(eval-in-linear-list '(list (0 . 0) 'blah (1 . 7) 'halb (2 . 5))
                     '(10 20 30 40 50 60 70 80 90 100
                       11 21 31 41 51 61 71 81 91 101
                       12 22 32 42 52 62 72 82 92 102)
                     10)  ⟼  (10 blah 81 halb 62)
```

Shrnutí

V této lekci jsme pokračovali v problematice rekurzivních procedur. Ukázali různé způsoby specifikace limitní podmínky v rekurzivních procedurách a vysvětlili jsme, proč by „`if` jako procedura“ nebyla použitelná pro zastavování rekurze. Dále jsme se zabývali otázkou, zda je speciální forma `define` nutná pro vytváření rekurzívních procedur, a ukázali jsme že rekurzi můžeme zajistit i pomocí konstruktu nazývaného y-kombinátor. V další části lekce jsme představili speciální formu `letrec`, která umožňuje definovat lokální rekurzivní procedury. V závěru lekce jsme se pak věnovali hloubkové rekurzi na seznamech a hloubkovým akumulačním procedurám.

Pojmy k zapamatování

- metody zastavení rekurze

- y-kombinátor
- hloubková rekurze na seznamech
- atom
- linearizace seznamu

Nově představené prvky jazyka Scheme

- element *nedefinovaná hodnota*
- speciální forma `letrec`

Kontrolní otázky

1. *Jakými způsoby lze zastavit rekurzi?*
2. *Proč* `if` *jako procedura nezastaví rekurzi?*
3. *Jakou má roli speciální forma* `define` *při psaní rekurzivních procedur.*
4. *Co je to y-kombinátor?*
5. *Jak se používá speciální forma* `letrec`*?*
6. *Na jaké výrazy se přepisují* `letrec`*-bloky?*
7. *Co je to nedefinovaná hodnota?*
8. *Co se myslí hloubkovou rekurzí na seznamech?*

Cvičení

1. pomocí y-kombinátoru naprogramujte následující procedury:
   - proceduru pro výpočet n-tého Fibonacciho čísla `fib`
   - mapování procedury přes jeden seznam `map1`
   - linearizace seznamu `linearize`
2. Implementujte následující procedury bez použití procedury `depth-accum`:
   - `depth-filter` – hloubková filtrace na seznamu
   - `depth-map` – hloubkové mapování procedury přes seznam
   - `atom?` – predikát zjišťující, jestli je element atomem seznamu
3. Implementujte procedury z předchozího úkolu pomocí první verze `depth-accum`.
4. Implementujte procedury z předchozího úkolu pomocí druhé verze `depth-accum`.

Úkoly k textu

1. Popište rozdíl mezi speciální formou `letrec` a speciální formou `let+` z úkolů k textu lekce 2. Napište kód, jehož výsledek vyhodnocení se bude lišit při použití těchto speciálních forem.
2.

Řešení ke cvičením

1. - ```
     ;; fib
     (lambda (n)
       ((lambda (y)
          (y y n))
        (lambda (fib n)
          (if (<= n 1) n
              (+ (fib fib (- n 1))
                 (fib fib (- n 2)))))))
     ```
     ```

- ;; map1
```
(lambda (f l)
  ((lambda (y)
     (y y f l))
   (lambda (map1 f l)
     (if (null? l)
  '()
  (cons (f (car l))
        (map1 map1 f (cdr l)))))))
```
- ;; linearize
```
(lambda (l)
  ((lambda (y)
    (y y l))
   (lambda (lin l)
     (cond ((null? l) '())
           ((list? (car l)) (append (lin lin (car l)) (lin lin (cdr l))))
           (else (cons (car l) (lin lin (cdr l))))))))
```

2.
```
(define depth-filter
  (lambda (p? l)
    (cond ((null? l) '())
          ((list? (car l)) (cons (depth-filter p? (car l))
                                 (depth-filter p? (cdr l))))
          ((p? (car l)) (cons (car l) (depth-filter p? (cdr l))))
          (#t (depth-filter p? (cdr l))))))
```

-
```
(define depth-map
  (lambda (f l)
    (cond ((null? l) '())
          ((list? (car l)) (cons (depth-map f (car l))
                                 (depth-map f (cdr l))))
          (#t (cons (f (car l)) (depth-map f (cdr l)))))))
```

-
```
(define atom?
  (lambda (a l)
    (if (null? l) #f
        (or (if (list? (car l))
                (atom? a (car l))
                (equal? (car l) a))
            (atom? a (cdr l))))))
```

3.
-
```
(define depth-filter
  (lambda (pred? l)
    (depth-accum (lambda (x y)
     (if (null? x)
         y
                (cons x y)))
      '()
       (lambda (x) (if (pred? x) x '()))
       l)))
```

-
```
(define depth-map
  (lambda (f l)
    (depth-accum cons '() f l)))
```

-
```
(define atom?
  (lambda (a l)
    (depth-accum (lambda (x y) (or x y)) #f (lambda(x) (equal? a x)) l)))
```

4.

- ```
  (define depth-filter
    (lambda (pred? l)
      (depth-accum (arbitrary (lambda (x y)
                                (if (null? x)
                                    y
                                    (cons x y))) '())
                   (lambda (x) (if (pred? x) x '()))
                   l)))
  ```
- ```
  (define depth-map
    (lambda (f l)
      (depth-accum list f l)))
  ```
- ```
  (define atom?
    (lambda (a l)
      (depth-accum (arbitrary (lambda (x y) (or x y)) #f)
                   (lambda(x) (equal? a x)) l)))
  ```
  ```