KATEDRA INFORMATIKY

PŘÍRODOVĚDECKÁ FAKULTA

UNIVERZITA PALACKÉHO

PARADIGMATA PROGRAMOVÁNÍ 1B

JAN KONEČNÝ, VILÉM VYCHODIL

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

Olomouc 4. 10. 2010 (aktualizace)

Abstrakt

Série učebních textů z paradigmat programování seznamuje čtenáře s vybranými styly v programování. Tento text je zaměřen na problematiku funkcionálního programování. Text je koncipován jako cvičebnice, teoretické partie jsou prezentovány v redukované podobě a důraz je kladen na prezentaci příkladů a protipříkladů, na kterých jsou dané problémy demonstrovány. V textu se nacházejí klasické partie z funkcionálního programování (například procedury vyšších řádů, rekurze, indukce, práce s hierarchickými daty), ale i partie, které jsou v soudobé literatuře zastoupeny minimálně nebo vůbec (například detailní popis vyhodnocovacího procesu a konstrukce interpretu funkcionálního jazyka). Text je rozdělen do dvanácti na sebe navazujících lekcí. Pro správné pochopení a procvičení problematiky je vhodné číst lekce postupně bez přeskakování a snažit se řešit všechny úkoly. Text u čtenářů nepředpokládá žádné znalosti programování. Pro pochopení většiny příkladů stačí mít znalost středoškolské matematiky. Příklady vyžadující znalosti (úvodních kurzů) vysokoškolské matematiky jsou doplněny o odkazy na vhodnou literaturu.

Jan Konečný, <jan.konecny@upol.cz>
Vilém Vychodil, <vilem.vychodil@upol.cz>

Cílová skupina

Text je určen pro studenty oborů Aplikovaná informatika a Informatika uskutečňovaných v prezenční a kombinované formě na Přírodovědecké fakultě Univerzity Palackého v Olomouci. Může být užitečný i studentům jiných informatických, matematických a inženýrských oborů a všem, kteří se chtějí seznámit s různými styly programování.

Obsah

1 Program, jeho syntax a sémantika . . . 7
- 1.1 Programovací jazyk, program a výpočetní proces . . . 7
- 1.2 Syntax a sémantika programů . . . 13
- 1.3 Přehled paradigmat programování . . . 16
- 1.4 Symbolické výrazy a syntaxe jazyka Scheme . . . 19
- 1.5 Abstraktní interpret jazyka Scheme . . . 21
- 1.6 Rozšíření Scheme o speciální formy a vytváření abstrakcí pojmenováním hodnot . . . 31
- 1.7 Pravdivostní hodnoty a podmíněné výrazy . . . 35

2 Vytváření abstrakcí pomocí procedur . . . 42
- 2.1 Uživatelsky definovatelné procedury a λ-výrazy . . . 42
- 2.2 Vyhodnocování elementů v daném prostředí . . . 47
- 2.3 Vznik a aplikace uživatelsky definovatelných procedur . . . 50
- 2.4 Procedury vyšších řádů . . . 52
- 2.5 Procedury versus zobrazení . . . 57
- 2.6 Lexikální a dynamický rozsah platnosti . . . 62
- 2.7 Další podmíněné výrazy . . . 64

3 Lokální vazby a definice . . . 78
- 3.1 Vytváření lokálních vazeb . . . 78
- 3.2 Rozšíření λ-výrazů a lokální definice . . . 86
- 3.3 Příklady na použití lokálních vazeb a interních definic . . . 89
- 3.4 Abstrakční bariéry založené na procedurách . . . 92

4 Tečkové páry, symbolická data a kvotování . . . 97
- 4.1 Vytváření abstrakcí pomocí dat . . . 97
- 4.2 Tečkové páry . . . 99
- 4.3 Abstrakční bariéry založené na datech . . . 103
- 4.4 Implementace tečkových párů pomocí procedur vyšších řádů . . . 105
- 4.5 Symbolická data a kvotování . . . 107
- 4.6 Implementace racionální aritmetiky . . . 109

5 Seznamy . . . 115
- 5.1 Definice seznamu a příklady . . . 115
- 5.2 Program jako data . . . 118
- 5.3 Procedury pro manipulaci se seznamy . . . 121
- 5.4 Datové typy v jazyku Scheme . . . 126
- 5.5 Implementace párů uchovávajících délku seznamu . . . 129
- 5.6 Správa paměti během činnosti interpretu . . . 131
- 5.7 Odvozené procedury pro práci se seznamy . . . 132
- 5.8 Zpracování seznamů obsahujících další seznamy . . . 136

6 Explicitní aplikace a vyhodnocování . . . 142
- 6.1 Explicitní aplikace procedur . . . 142
- 6.2 Použití explicitní aplikace procedur při práci se seznamy . . . 144
- 6.3 Procedury s nepovinnými a libovolnými argumenty . . . 149
- 6.4 Vyhodnocování elementů a prostředí jako element prvního řádu . . . 152
- 6.5 Reprezentace množin a relací pomocí seznamů . . . 161

7 Akumulace . . . 171
7.1 Procedura FOLDR . . . 171
7.2 Použití FOLDR pro definici efektivních procedur . . . 174
7.3 Další příklady akumulace . . . 179
7.4 Procedura FOLDL . . . 182
7.5 Další příklady na FOLDR a FOLDL . . . 185
7.6 Výpočet faktoriálu a Fibonacciho čísel . . . 186
8 Rekurze a indukce . . . 191
8.1 Definice rekurzí a princip indukce . . . 191
8.2 Rekurze a indukce přes přirozená čísla . . . 199
8.3 Výpočetní procesy generované rekurzivními procedurami . . . 206
8.4 Jednorázové použití procedur . . . 217
8.5 Rekurze a indukce na seznamech . . . 220
8.6 Reprezentace polynomů . . . 228
9 Hloubková rekurze na seznamech . . . 241
9.1 Metody zastavení rekurze . . . 241
9.2 Rekurzivní procedury definované pomocí y-kombinátoru . . . 243
9.3 Lokální definice rekurzivních procedur . . . 246
9.4 Implementace vybraných rekurzivních procedur . . . 248
9.5 Hloubková rekurze na seznamech . . . 250
10 Kombinatorika na seznamech, reprezentace stromů a množin . . . 259
10.1 Reprezentace n-árních stromů . . . 259
10.2 Reprezentace množin pomocí uspořádaných seznamů . . . 261
10.3 Reprezentace množin pomocí binárních vyhledávacích stromů . . . 264
10.4 Kombinatorika na seznamech . . . 265
11 Kvazikvotování a manipulace se symbolickými výrazy . . . 270
11.1 Kvazikvotování . . . 270
11.2 Zjednodušování aritmetických výrazů . . . 272
11.3 Symbolická derivace . . . 277
11.4 Infixová, postfixová a bezzávorková notace . . . 279
11.5 Vyhodnocování výrazů v postfixové a v bezzávorkové notaci . . . 282
12 Čistě funkcionální interpret Scheme . . . 290
12.1 Automatické přetypování a generické procedury . . . 290
12.2 Systém manifestovaných typů . . . 293
12.3 Datová reprezentace elementů jazyka Scheme . . . 295
12.4 Vstup a výstup interpretu . . . 301
12.5 Implementace vyhodnocovacího procesu . . . 302
12.6 Počáteční prostředí a cyklus REPL . . . 307
12.7 Příklady použití interpretu . . . 313
A Seznam vybraných programů . . . 329
B Seznam obrázků . . . 332

Lekce 7: Akumulace

Obsah lekce: V této lekci se budeme zabývat akumulací, což je speciální postupná aplikace procedur. Pomocí akumulace ukážeme efektivnější řešení vybraných úkolů, které jsme řešili v předchozích lekcích, například mapování, filtrace a nahrazování prvků seznamu. Dále ukážeme, jak lze pomocí akumulace rozšířit některé procedury dvou argumentů tak, aby pracovaly s libovolným počtem argumentů.

Klíčová slova: akumulace, filtrace, procedura `foldl`, procedura `foldr`.

7.1 Procedura FOLDR

V této sekci se budeme zabývat *akumulací prvků seznamu*. Pod pojmem *akumulace* máme obvykle na mysli vytvoření jedné hodnoty pomocí více hodnot obsažených v seznamu (sečtení čísel v seznamu, nalezení maxima, vytvoření seznamu pouze z některých hodnot a podobně). Akumulace má blízko k explicitní aplikaci prováděné pomocí primitivní procedury `apply`, o které jsme se bavili v předchozí lekci. Použití `apply` při akumulaci má nevýhodu v tom, že při neznámé délce seznamu můžeme aplikovat pouze procedury s libovolnými argumenty. Jedině tak je totiž při použití `apply` zaručeno, že nedojde k chybě vlivem předání špatného počtu argumentů. Nyní budeme k problému akumulace přistupovat jinak. Seznam libovolné délky budeme akumulovat pomocí *procedur dvou argumentů*, které budou *aplikovány postupně* pro jednotlivé prvky seznamu a to buď zleva nebo zprava.

Nejprve si problém demonstrujeme na příkladu. Uvažujme seznam čísel

`(1 2 3 4)`.

Z předchozích lekcí víme, že tento seznam lze zkonstruovat pomocí konstruktoru párů `cons` a pomocí prázdného seznamu vyhodnocením následujícího výrazu:

```
(cons 1 (cons 2 (cons 3 (cons 4 '())))) ⟹ (1 2 3 4)
```

Ve výrazu si můžeme všimnout toho, že se na sebe postupně „nabalují“ výsledky aplikace procedury `cons`. Hodnota vzniklá vyhodnocením `(cons 4 '())`, což je jednoprvkový seznam obsahující čtyřku je dále použita jako druhý argument při další aplikaci `cons` spolu s argumentem `3`, výsledek této aplikace je opět použit při další aplikci `cons` jako druhý argument, a tak dále. Analogické postupné „nabalování“ výsledků aplikací bychom použili, kdybychom chtěli sečíst prvky seznamu za předpokladu, že procedura `+` by akceptovala pouze dva argumenty:

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

Aby podobnost obou výrazů více vynikla, přepíšeme předchozí s využitím součtu s nulou:

```
(+ 1 (+ 2 (+ 3 (+ 4 0))))
```

V podobném stylu bychom mohli vyjádřit součin prvků seznamu:

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

Nyní uvedeme ještě jeden příklad, ve kterém pro změnu nebudeme používat při postupném „nabalování“ výsledných hodnot primitivní procedury jako dosud (procedury `cons`, `+` a `*`), ale následující uživatelsky definovanou proceduru:

```
(define y+1
  (lambda (x y)
    (+ y 1)))
```

Všimněte si, že předchozí procedura zcela ignoruje svůj první argument a vrací hodnotu druhého argumentu zvětšenou o jedna (druhý argument tedy musí být číslo). V následující ukázce jsou zachyceny výsledky aplikací této procedury:

```
(y+1 'a 0)                          ⟹ 1
(y+1 'a (y+1 'b 0))                 ⟹ 2
(y+1 'a (y+1 'b (y+1 'c 0)))        ⟹ 3
(y+1 'a (y+1 'b (y+1 'c (y+1 'd 0)))) ⟹ 4
```

Jak je asi z ukázky a z definice procedury `y+1` patrné, proceduru můžeme tímto způsobem použít k počítání počtu prvků seznamu. Pro náš výchozí seznam bychom tedy měli:

```
(y+1 1 (y+1 2 (y+1 3 (y+1 4 0))))   ⟹   4
```

Výraz je opět ve tvaru postupného „nabalování" aplikací. V čem se lišily všechny předchozí ukázky postupně vnořených aplikací, které jsme uvedli? Pro zopakování, jednalo se o tyto výrazy:

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
(+ 1 (+ 2 (+ 3 (+ 4 0))))
(* 1 (* 2 (* 3 (* 4 1))))
(y+1 1 (y+1 2 (y+1 3 (y+1 4 0))))
```

Z hlediska jejich tvaru, ze lišily jen „nepatrně", protože je lze všechny chápat jako výrazy tvaru:

(⟨*procedura*⟩ 1 (⟨*procedura*⟩ 2 (⟨*procedura*⟩ 3 (⟨*procedura*⟩ 4 ⟨*terminátor*⟩)))),

kde ⟨*procedura*⟩ je procedura dvou argumentů a ⟨*terminátor*⟩ je element. Vskutku, v prvním případě byla ⟨*procedura*⟩ zastoupena `cons` a terminátor byl `()`. V druhém případě byla ⟨*procedura*⟩ zastoupena `+` a ⟨*terminátor*⟩ byl `0`. V posledním případě byla ⟨*procedura*⟩ zastoupena uživatelsky definovanou procedurou `y+1` a ⟨*terminátor*⟩ byl `0`. Z pohledu tvaru se tedy výrazy příliš neliší. Liší se ale výrazně z pohledu svého významu (výsledných hodnot): vytvoření seznamu, součet hodnot, součin hodnot, výpočet délky (což je v podstatě „počet zanoření " v daném výrazu).

Doposud jsme všechny úvahy prováděli nad seznamem pevné délky. Úvahy bychom ale mohli rozšířit na seznamy libovolných délek. Nabízí se mít k dispozici obecnou proceduru, která pro danou *proceduru dvou argumentů*, *terminátor* a *seznam* provede postupnou aplikaci dané procedury dvou argumentů přes všechny prvky seznamu tak, jak jsme nyní v několika případech ukázali. V jazyku Scheme budeme uvažovat proceduru `foldr` (z anglického *fold right*, neboli „zabal směrem doprava"), která je ve své základní podobě aplikována ve tvaru:

(`foldr` ⟨*procedura*⟩ ⟨*terminátor*⟩ ⟨*seznam*⟩).

Při své aplikaci procedura `foldr` provede postupnou aplikaci

(⟨*procedura*⟩ ⟨$prvek_1$⟩ (⟨*procedura*⟩ ⟨$prvek_2$⟩ (⟨*procedura*⟩ $\cdots$ (⟨*procedura*⟩ ⟨$prvek_n$⟩ ⟨*terminátor*⟩) $\cdots$))),

pro ⟨*seznam*⟩ ve tvaru (⟨$prvek_1$⟩ ⟨$prvek_2$⟩ $\cdots$ ⟨$prvek_n$⟩). To jest, použitím `foldr` na prázdný seznam je vrácen element označený jako ⟨*terminátor*⟩. Použitím `foldr` na jednoprvkový, dvouprvkový, tříprvkový, čtyřprvkový (a tak dále) seznam je vrácena hodnota aplikace:

(⟨*procedura*⟩ ⟨$prvek_1$⟩ ⟨*terminátor*⟩)
(⟨*procedura*⟩ ⟨$prvek_1$⟩ (⟨*procedura*⟩ ⟨$prvek_2$⟩ ⟨*terminátor*⟩))
(⟨*procedura*⟩ ⟨$prvek_1$⟩ (⟨*procedura*⟩ ⟨$prvek_2$⟩ (⟨*procedura*⟩ ⟨$prvek_3$⟩ ⟨*terminátor*⟩)))
(⟨*procedura*⟩ ⟨$prvek_1$⟩ (⟨*procedura*⟩ ⟨$prvek_2$⟩ (⟨*procedura*⟩ ⟨$prvek_3$⟩ (⟨*procedura*⟩ ⟨$prvek_4$⟩ ⟨*terminátor*⟩))))
$\vdots$

Předchozí příklady bychom tedy pomocí `foldr` mohli vyřešit takto:

```
(define s '(1 2 3 4))
(foldr cons '() s)      ⟹   (1 2 3 4)
(foldr + 0 s)           ⟹   10
(foldr * 1 s)           ⟹   24
(foldr y+1 0 s)         ⟹   4
```

První aplikací `foldr` jsme vytvořili duplikát výchozího seznamu, následuje součet a součin prvků seznamu a poslední příklad bychom mohli v souladu s našimi předchozími pozorováními charakterizovat jako vypočtení délky seznamu. Použití `foldr` má zjevnou výhodu v tom, že funguje pro libovolně dlouhý seznam o jehož transformaci na sérii postupných aplikací se jako programátoři nemusíme starat. O tom se ostatně můžete přesvědčit sami, zkuste několikrát změnit seznam nabázaný na `s` a proveďte výše uvedené aplikace `foldr`.

Poznámka 7.1. Všimněte si, jakou roli mají argumenty procedury, kterou při aplikaci předáváme proceduře `foldr`. První argument této procedury postupně *nabývá hodnot vyskytujících se v seznamu.* Na druhou stranu druhý argument zastupuje element, který *vznikl zabalením hodnot* v seznamu vyskytujících se *za průběžným prvkem.* Například při následující aplikaci `foldr`:

```
(foldr (lambda (x y)
         (list #f x y))
       'base
       '(a b c d))  ⟹  (#f a (#f b (#f c (#f d base))))
```

bude procedura vzniklá vyhodnocením λ-výrazu `(lambda (x y) (list #f x y))` nejprve aplikována s hodnotami `d` (poslední prvek seznamu) a `base` (terminátor). Výsledkem tedy bude seznam `(#f d base)`. V dalším kroku bude procedura aplikována s prvkem `c` (předposlední prvek seznamu) a druhým argumentem bude výsledek zabalení vytvořený v předchozím kroku, tedy seznam `(#f d base)`. Výsledkem aplikace procedury proto bude seznam `(#f c (#f d base))`. V dalším kroku bude procedura aplikována s prvkem `b` a seznamem `(#f c (#f d base))`. Jako výsledek vznikne `(#f b (#f c (#f d base)))`. Konečně v posledním kroku bude procedura aplikována s `a` (první prvek seznamu) a s posledním uvedeným seznamem. Výsledek této poslední aplikace bude vrácen jako výsledek aplikace `foldr`.

Následující příklady ukazují roli *terminátoru.*

```
(foldr cons 'base s)     ⟹  (1 2 3 4 . base)
(foldr cons '() s)       ⟹  (1 2 3 4)
(foldr list 'base s)     ⟹  (1 (2 (3 (4 base))))
(foldr list '() s)       ⟹  (1 (2 (3 (4 ()))))
(foldr list '() '())     ⟹  ()
(foldr list 'base '())   ⟹  base
(foldr list 666 '())     ⟹  666
```

Například na prvních dvou použitích `foldr` je vidět, že v prvním případě je aplikována procedura `cons` s argumenty `4` (poslední prvek seznamu) a `base`, což vede k vytvoření tečkového páru `(4 . base)`, který se ve výsledku vyskytuje na konci vytvořené hierarchické struktury. V druhém případě je aplikace `cons` provedena s hodnotami `4` a `()`, což vede na vytvoření jednoprvkového seznamu `(4)` – výsledná struktura vytvořená aplikací `foldr` je v tomto případě seznam. V posledních třech příkladech vidíme mezní případ: při pokusu o zabalení prázdného seznamu je vrácen terminátor.

> Pomocí procedury `foldr` lze efektivně implementovat řadu operací nad seznamy. Klíčem ke správnému použití `foldr` je pochopit roli procedury dvou argumentů, která je při aplikaci předávána jako první argument, a pomocí níž je provedeno samotné „zabalení hodnot“. Musíme si uvědomit, že tato procedura je postupně aplikována tak, že její první argument je *průběžný prvek seznamu* a druhý argument je *výsledek zabalení prvků nacházejících se v seznamu za průběžným prvkem*.

Proceduru `foldr` lze použít v obecnějším tvaru. Podobně jako tomu bylo u procedury `map`, procedura `foldr` připouští při aplikaci i více seznamů než jen jeden. Proceduru `foldr` lze tedy aplikovat ve tvaru

`(foldr` $\langle procedura\rangle$ $\langle terminátor\rangle$ $\langle seznam_1\rangle$ $\langle seznam_2\rangle \cdots \langle seznam_n\rangle$`)`,

kde $\langle seznam_1\rangle, \ldots, \langle seznam_n\rangle$ jsou seznamy ($n \geq 1$), a $\langle procedura\rangle$ je procedura $n+1$ argumentů. Při aplikaci `foldr` je provedeno analogické zabalení jako ve verzi s jedním seznamem, rozdíl je pouze v tom, že $\langle procedura\rangle$ je aplikována s $n+1$ argumenty, jimiž je n průběžných prvků z předaných seznamů a posledním argumentem je výsledek zabalení prvků následujících za průběžnými prvky.

Nejlépe činnost `foldr` pro víc argumentů vysvětlí následující příklady:

```
(foldr list 'base
       '(1 2 3) '(a b c) '(i j k))    ⟹  (1 a i (2 b j (3 c k base)))
```

```
(foldr (lambda (x y z result)
         (cons (list x y z) result))
       '()
       '(1 2 3) '(a b c) '(i j k))    ⟹  ((1 a i) (2 b j) (3 c k))
```

V následující sekci uvedeme praktické příklady použití `foldr`.

7.2 Použití FOLDR pro definici efektivních procedur

Nyní ukážeme implementaci několika procedur pro práci se seznamy, které už jsme představili v předchozích lekcích. Uvidíme, že vytvoření těchto procedur pomocí `foldr` bude nejen kratší z hlediska jejich zápisu, ale procedury budou mnohem *efektivnější*. I když není výpočtová efektivita hlavním předmětem, na který se v tomto kurzu soustředíme, u každé vytvářené procedury je vždy vhodné zamýšlet se nad její efektivitou. Pro programování ve funkcionálních jazycích je typické vyvíjet program v několika etapách. V první fázi vývoj jde vlastně o obohacení jazyka o nové procedury, pomocí nichž budeme schopni vyřešit daný problém. Po vyřešení problému se můžeme opět vrátit k naprogramovaným procedurám a pokoušet se zvýšit jejich efektivitu tím, že je implementujeme znovu, samozřejmě při zachování dosavadní funkčnosti.

Tento pohled uplatníme v malém měřítku i nyní. Ukážeme si, jak lze efektivně vytvořit procedury, které jsme již měli (méně efektivně) vytvořené. První z nich bude procedura `length` vracející délku seznamu. Původní kód, který jsme vytvořili pomocí `map` a `apply` je uveden v programu 6.1 na straně 145. Před uvedením nové verze `length` se nejdřív zamysleme nad efektivitou původní verze z programu 6.1. Předně, jak můžeme snadno kvantitativně vyjádřit efektivitu procedury pracující se seznamy? Seznamy jsou sekvenční lineární dynamické datové struktury, k jejichž (dalším) prvkům přistupujeme pomocí `cdr`. Tato operace má vzhledem k dalším používaným operacím největší vliv na rychlost zpracování seznamu. Proto budeme vyjadřovat efektivitu *amortizovaně vzhledem k počtu operací* `cdr` provedených nad vstupními daty. Budeme přitom provádět běžná zjednodušení, která jsou studentům známá z kurzu *algoritmické matematiky*, nebudeme se zabývat samotnou strukturou seznamů, ale efektivitu (časovou složitost procedur) budeme vyjadřovat *vzhledem k délce vstupních seznamů.*

Procedura `length` v programu 6.1 počítá výslednou hodnotu tak, že nejprve provede mapování přes celý vstupní seznam. Pokud délku vstupního seznamu označíme n, pak tato fáze zabere n kroků (je potřeba n aplikací `cdr` na průchod seznamem[11]). V další fází je na vzniklý seznam aplikována operace sčítání ta potřebuje opět n kroků k tomu, aby prošla prvky seznamu (jedničky) a sečetla je. Dohromady tedy procedura pro seznam délky n provede $2n$ kroků. Časovou složitost budeme dále zapisovat v běžné O-notaci, v případě naší procedury tedy $O(2n)$. Intuitivně bychom očekávali, že délku n prvkového seznamu bychom měli stanovit právě v n krocích, procedura `length` z programu 6.1 tedy není příliš efektivní. Na druhou strany není těžké nahlédnout, že délku seznamu se nám nepodaří stanovit v „sublineárním čase“, protože pro výpočet délky seznamu musíme skutečně každý prvek seznamu navštívit aspoň jednou.

Podívejme se nyní na proceduru `length` z programu 7.1. Jde v podstatě jen o formalizaci myšlenky z před-

Program 7.1. Výpočet délky seznamu pomocí `foldr`.

```
(define length
  (lambda (l)
    (foldr (lambda (x y)
             (+ 1 y))
           0 l)))
```

[11]Někdo by v tuto chvíli mohl namítnout, že kroků je potřeba pouze $n-1$, protože u jednoprvkového seznamu už žádný další prvek nehledáme. Tato úvaha ale není správná, abychom zjistili, že se skutečně jedná o jednoprvkový seznam, musíme otestovat, jaký element se nachází na druhé pozici páru, tím pádem `cdr` musíme skutečně aplikovat n-krát. Uvědomte si, že interpret nevidí seznam „z vnějšku“ tak, jako jej vidíme my.

chozí sekce. Jelikož je v těle procedury aplikována procedura `foldr` se vstupním seznamem, její činnost zabere právě n kroků, během kterých jsou navštíveny všechny prvky seznamu. Hodnota terminátoru `0` znamená, že prázdný seznam má délku nula. Během akumulace je sečteno právě tolik jedniček, kolik je navštíveno prvků, výsledná hodnota je tedy číslo – délka seznamu. Celková časová složitost našeho řešení je tedy $O(n)$.

Z kurzu algoritmické matematiky možná víte, že složitost se stanovuje řádově. Z tohoto pohledu jsou složitosti $O(2n)$ a $O(n)$ obě stejného řádu (lineární složitost), protože multiplikativní konstanta je z hlediska řádové složitosti zanedbatelná[12]. Zde upozorněme na to, že orientovat se podle řádové složitosti, používané třeba ke klasifikaci řešitelných problémů podle jejich časové nebo prostorové složitosti, by bylo z našeho pohledu dost ošidné. Multiplikativní konstanta 2 je z pohledu procedury jako je `length` (která bude v programu zřejmě často používána), dost kritická a naše nová implementace mající složitost $O(n)$ je výrazně lepší než původní se složitostí $O(2n)$[13]. Další nově naprogramovanou procedurou bude spojení dvou seznamů. Původní kód procedury `append2` je k dispozici v programu 5.2 na straně 124. Tato implementace využívající `build-list` je extrémně neefektivní. Označíme-li délku prvního seznamu n a druhého m, pak je nejprve spotřebováno $n+m$ kroků na stanovení délek obou seznamů. Potom je konstruován nový seznam délky $n+m$. Proto, abychom zjistili složitost konstrukce, musíme rozebrat tělo procedury volané procedurou `build-list`. Při pohledu na tělo je jasné, že jsou postupně vraceny prvky z obou seznamů pomocí `list-ref`. Samotná procedura `list-ref` vrátí prvek na k-té pozici (nejdřív) během k kroků. Pro vrácení všech prvků z prvního seznamu tedy potřebujeme $1+2+\cdots+n$ kroků, což je (sečtením prvků aritmetické posloupnosti) dohromady $\frac{n(1+n)}{2}$ kroků. Pro druhý seznam potřebujeme analogicky $\frac{m(1+m)}{2}$ kroků. Celkovou složitost stanovíme součtem všech tří částí (výpočet délek a sekvenční přístup ke všem prvkům obou seznamů), to jest

$$O\big(n+m+\tfrac{n(1+n)}{2}+\tfrac{m(1+m)}{2}\big),$$

což je ekvivalentní

$$O\big(\tfrac{n(n+3)+m(m+3)}{2}\big).$$

Řádově je tedy časová složitost procedury z programu 5.2 dokonce *kvadratická*. To je přímo tristní, protože pro spojení seznamu délky m a seznamu délky n bychom intuitivně očekávali složitost $O(m+n)$. Program 7.2 dokonce ukazuje, že na tom můžeme být ještě o něco lépe. Nejdříve objasněme tělo nové

Program 7.2. Spojení dvou seznamů pomocí `foldr`.

```
(define append2
  (lambda (l1 l2)
    (foldr cons l2 l1)))
```

implementace procedury `append2`. Jedná se o akumulaci prvků prvního seznamu pomocí `cons`, která je terminována druhým předaným seznamem. Všechny prvky prvního seznamu jsou při této akumulaci postupně navštíveny (jeden po druhém), což zabere n kroků. Výsledná složitost je tedy $O(n)$. Pro někoho poněkud překvapivě, protože se do složitosti vůbec nepromítla délka druhého seznamu. Vskutku, druhý seznam je použit jako terminující element a není tedy vůbec procházen. Činnost `append2` napsané pomocí `foldr` si možná lépe uvědomíme, když si představíme, že `foldr` provede sérii aplikací typu:

(cons $\langle prvek_1\rangle$ (cons $\langle prvek_2\rangle$ (cons $\cdots$ (cons $\langle prvek_n\rangle$ $\langle seznam\rangle$) $\cdots$))),

která skutečně vede na spojení dvou seznamů: ($\langle prvek_1\rangle \cdots \langle prvek_n\rangle$) a $\langle seznam\rangle$.

Pomocí `foldr` a `append2` můžeme efektivně naprogramovat spojení libovolného množství seznamů tak, jak to ukazuje procedura `append` v programu 7.3. Proceduru jsme pochopitelně museli definovat jako

[12] Z prektického pohledu bychom se měli zajímat i o multiplikativní konstanty zvlášť v případě, pokud jsou velké.

[13] Zde opět připomeňme, že řádově jsou obě složitostní třídy stejné, to jest $O(n) = O(2n)$. Z praktického hlediska je však v tomto případě konstanta 2 výraznou přítěží. Pokud budeme chtít multiplikativní konstanty zdůrazňovat, budeme je v O-notacích uvádět, i když to není běžné.

Program 7.3. Spojení libovolného počtu seznamů pomocí foldr.

```
(define append
  (lambda lists
    (foldr append2 '() lists)))
```

proceduru s libovolnými argumenty, ty budou při její aplikaci navázané na symbol lists. V těle procedury je použito jedno volání foldr pomocí nějž provádíme akumulaci procedury append2, kterou jsme vytvořili v předchozím kroku. Tato procedura bude akumulovat prvky ze seznamu lists, což jsou seznamy předané proceduře append při její aplikaci. Terminátorem je prázdný seznam, protože spojením „žádných seznamů" vzniká prázdný seznam. Složitost této procedury je rovna $O(n)$, kde n je součet délek všech vstupních seznamů. Pokud použijeme append na spojení pouze dvou seznamů, bude mít složitost $O(n+m)$, což je zhoršení oproti append2. Důvodem je fakt, že append terminuje spojení prázdným seznamem a prochází prvky *všech* předaných seznamů (tedy i toho posledního, v našem případě druhého). To je jakási daň, kterou jsme zaplatili za obecnost řešení.

V programu 5.3 na straně 126 jsme ukázali implementaci procedury map1, což je varianta map pracující pouze s jedním seznamem. Tato implementace byla opět velmi neefektivní a používala build-list a sekvenční vyhledávání prvků pomocí opakované aplikaci list-ref. Časová složitost této procedury byla $O(\frac{n(n+3)}{2})$, protože n kroků bylo spotřebováno vypočtením délky seznamu a $1+2+\cdots+n$ kroků bylo potřeba na procházení jeho prvků. Složitost takto napsané map1 byla opět kvadratická. Proceduru lze ale vytvořit se složitostí $O(n)$ tak, jak je to ukázáno v programu 7.4. V nové implementaci map1 jsme prováděli akumulaci

Program 7.4. Mapovací procedura pracující s jedním seznamem pomocí foldr.

```
(define map1
  (lambda (f l)
    (foldr (lambda (x y)
             (cons (f x) y))
           '()
           l)))
```

hodnot pomocí uživatelsky definované procedury, která místo prostého použití cons tak, jak jsme jej použili v případě append2, provede napojení *modifikace prvního prvku* s již zpracovanou částí. Akumulace je terminována prázdným seznamem. Pomocí foldr můžeme nyní naprogramovat i obecný map pracující s libovolným (ale nenulovým) počtem seznamů. Obecná verze map se nachází v programu 7.5. Program 7.5 obsahuje pomocnou proceduru separate-last-argument, která má za účel pro daný neprázdný seznam vrátit tečkový pár, jehož prvním prvkem bude seznam prvků z původního seznamu kromě posledního a druhým prvkem tečkového páru bude poslední prvek seznamu. Viz následující příklady použití:

```
(separate-last-argument '())          ⟹  #f
(separate-last-argument '(a))         ⟹  (() . a)
(separate-last-argument '(a b))       ⟹  ((a) . b)
(separate-last-argument '(a b c))     ⟹  ((a b) . c)
(separate-last-argument '(a b c d))   ⟹  ((a b c) . d)
```

Procedura separate-last-argument nám tedy umožňuje přistoupit k prvkům seznamu (vyjma posledního) a k poslednímu prvku. Jedná se tedy o jakési „car a cdr naruby". Tuto pomocnou proceduru jsme v programu 7.5 dále použili na implementaci map. Samotný map jsme realizovali aplikací foldr. Jelikož však dopředu nevíme, kolik seznamů bude proceduře map předáno, museli jsme proceduru předanou foldr vytvořit jako proceduru s libovolným počtem argumentů. Pro n seznamů bude argumentů $n+1$, prvních n argumentů bude reprezentovat průběžné prvky seznamů a poslední argument bude zastupovat

Program 7.5. Obecná mapovací procedura pomocí `foldr`.

```
(define separate-last-argument
  (lambda (l)
    (foldr (lambda (x y)
             (if (not y)
                 (cons '() x)
                 (cons (cons x (car y)) (cdr y))))
           #f
           l)))

(define map
  (lambda (f . lists)
    (apply foldr
           (lambda args
             (let ((separation (separate-last-argument args)))
               (cons (apply f (car separation))
                     (cdr separation))))
           '()
           lists)))
```

akumulovanou hodnotu. Jelikož chceme k akumulované hodnotě přidat hodnotu vzniklou aplikací prvních n argumentů, potřebujeme od sebe nutně oddělit prvních n argumentů a poslední argument. K tomu jsme použili právě pomocnou proceduru `separate-last-argument`, která byla rovněž vytvořená pomocí `foldr`.

Všimněte si, že v proceduře `separate-last-argument` je `foldr` terminován elementem `#f`. V těle procedury předané `foldr` je vidět, že hned při první aplikaci, kdy je na `x` navázaný poslední prvek seznamu, je vytvořen pár ve tvaru `(() .` ⟨*poslední*⟩`)`. V každém dalším kroku již se druhý prvek tohoto páru nemění, a do prvního prvku se přidávají postupně procházené prvky. Tím vytvoříme požadovaný výstup, viz výše uvedené příklady.

Složitost obecného `map` můžeme stanovit zhruba takto. Procházíme m seznamů délky n postupně prvek po prvku, to zabere celkem mn kroků. K tomu musíme připočíst režii spojenou s násobnou aplikací pomocné procedury `separate-last-argument`. Tato procedura je aplikována právě tolikrát, jaká je délka seznamů, tedy n-krát. Při každé aplikaci potřebuje $m + 1$ kroků na separaci posledního argumentu. Celková složitost je tedy $O(mn + (m + 1) \cdot n)$, což je ekvivalentní $O((2m + 1) \cdot n)$.

Program 7.6 obsahuje efektivní implementaci filtrační procedury `filter`, kterou jsme představili v programu 6.2 na straně 147. Původní filtrační procedura měla časovou složitost $O(2n)$, zdůvodnění je analogické tomu, jaké jsme provedli u původní procedury `length`. Efektivní implementace z programu 7.6 ukazuje další použití `foldr`. V tomto případě je terminátorem opět prázdný seznam a uživatelsky definovaná procedura předaná `foldr` nejprve otestuje, zda-li průběžný prvek splňuje vlastnost danou procedurou navázanou na symbol `f`. Pokud ano, je prvek přidán k seznamu v němž se akumulují prvky splňující tyto vlastnost. V opačném případě není seznam akumulovaných prvků změněn. Časová složitost nového provedení `filter` je $O(n)$.

Další ukázkou je efektivní implementace predikátu `member?`. Tento predikát jsme představili v programu 6.3 na straně 147. Složitost původní implementace byla $O(2n)$, protože byla založena na původní implementaci `filter`. Kdybychom nyní uvažovali, že ponecháme původní kód `member?`, ale budeme v něm používat novou implementaci `filter` z programu 7.6, pak bude mít `member?` časovou složitost $O(n)$. Můžeme ale provést úplně novou implementaci `member?` přímo použitím `foldr` bez vazby na `filter`. Viz program 7.7.

Poslední procedurou, kterou v této sekci ukážeme je `replace`, která při své aplikaci vyžaduje tři argu-

Program 7.6. Filtrace prvků seznamu splňujících danou vlastnost pomocí `foldr`.

```
(define filter
  (lambda (f l)
    (foldr (lambda (x y)
             (if (f x)
                 (cons x y)
                 y))
           '()
           l)))
```

Program 7.7. Test přítomnosti prvku v seznamu pomocí `foldr`.

```
(define member?
  (lambda (elem l)
    (foldr (lambda (x y)
             (if (equal? x elem) #t y))
           #f
           l)))
```

menty: prvním je predikát jednoho argumentu reprezentující vlastnost prvku seznamu (analogická role jako u `filter`), druhým je procedura jednoho argumentu sloužící k modifikaci prvků seznamu (analogická role jako u `map`) a třetím argumentem je seznam. Výsledkem aplikace procedury `replace` je seznam elementů vzniklý ze vstupního seznamu tak, že každý prvek seznamu splňující vlastnost danou prvním argumentem je modifikován pomocí procedury dané druhým argumentem. Viz program 7.8.

Program 7.8. Nahrazení prvku dané vlastnosti modifikací prvku pomocí `foldr`.

```
(define replace
  (lambda (prop? modifier l)
    (foldr (lambda (x y)
             (if (prop? x)
                 (cons (modifier x) y)
                 (cons x y)))
           '()
           l)))
```

Následující příklady ukazují použití `replace`:

```
(define s '(1 2 3 4 5))
(replace even? (lambda (x) (- x)) s)          ⟹  (1 -2 3 -4 5)
(replace (lambda (x) #t) (lambda (x) 1) s)   ⟹  (1 1 1 1 1)
(replace (lambda (x) (<= x 3)) list s)        ⟹  ((1) (2) (3) 4 5)
(replace (lambda (x) (= 1 (modulo x 3)))
         (lambda (x) (+ x 10))
         s)                                    ⟹  (11 2 3 14 5)
```

7.3 Další příklady akumulace

V této sekci si ukážeme další příklady akumulace pomocí `foldr`. Nejprve se budeme zabývat problematikou rozšíření operace (procedury) dvou argumentů na proceduru *libovolných argumentů*. Konkrétně se budeme zabývat touto problematikou u monoidálních operací, viz sekci 2.5. Pokud je totiž operace $\circledcirc$ na dané množině asociativní a má neutrální prvek, pak můžeme bez újmy psát

$$a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n,$$

protože díky asociativitě nezáleží na uzávorkování předchozího výrazu. Díky neutralitě navíc platí, že

$$a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n = a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n \circledcirc e,$$

kde e je neutrální prvek vzhledem k operaci $\circledcirc$. Z hlediska akumulace pomocí `foldr` je pro nás zajímavé

$$a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n = (a_1 \circledcirc (a_2 \circledcirc \cdots (a_n \circledcirc e) \cdots)).$$

Pravá strana předchozí rovnosti je ve tvaru vhodném pro akumulaci pomocí `foldr`, protože monoidální operace $\circledcirc$ zde hraje analogickou roli jako procedura předávaná `foldr` a terminátor je neutrální prvek e. Kdybychom tedy ve Scheme neměli k dispozici `+`, `*` a podobné operace jako procedury libovolných argumentů, ale pouze dvou, pak bychom je pomocí `foldr` mohli snadno rozšířit na procedury libovolných argumentů.

V následujícím příkladu máme definovány procedury, které provádějí součet a součin dvou prvků:

```
(define add2 (lambda (x y) (+ x y)))
(define mul2 (lambda (x y) (* x y)))
```

Pomocí `foldr` je můžeme zobecnit na operace pro libovolný počet argumentů:

```
(define ++
  (lambda args
    (foldr add2 0 args)))

(define **
  (lambda args
    (foldr mul2 1 args)))
```

Samozřejmě, že předchozí příklad byl pouze „školský“, protože v interpretu máme k dispozici `+` a `*` pracující s libovolnými argumenty, takže není potřeba je „redukovat na dva argumenty“ pak „opět vyrábět“. Příklad měl sloužit pro demonstraci této obecné techniky. Analogicky jako v předchozím případě bychom mohli na libovolný počet argumentů zobecnit proceduru pro sčítání vektorů pevné délky:

```
(define vec+ (lambda (v1 v2) (map + v1 v2)))
```

Zde je ale malý problém s neutrálním prvkem. Neutrální prvek pro sčítání vektorů je pochopitelně nulový vektor. Pokud jsou vektory reprezentovány seznamem hodnot, pak by to měl být seznam skládající se ze samých nul. Potíž je ale v tom, že předchozí procedura byla schopná sčítat vektory *libovolné délky*, pro každou z délek máme jeden neutrální prvek. Situaci bychom mohli vyřešit tak, že bychom vytvořili proceduru vyššího řádu `make-vec+`, která by pro danou délku vrátila proceduru pro sčítání libovolně mnoha vektorů:

```
(define make-vec+
  (lambda (n)
    (let ((null-vector (build-list n (lambda (i) 0))))
      (lambda vectors
        (foldr vec+ null-vector vectors)))))
```

Použití procedury by pak bylo následující:

```
((make-vec+ 3))                                  ⟼  (0 0 0)
((make-vec+ 3) '(1 2 3))                         ⟼  (1 2 3)
((make-vec+ 3) '(1 2 3) '(10 20 30))             ⟼  (11 22 33)
((make-vec+ 3) '(1 2 3) '(10 20 30) '(2 4 6))    ⟼  (13 26 39)
```

Proceduru make-vec+ bychom místo foldr a vec+ mohli implementovat s použitím map pro libovolné argumenty. Následující příklad rovněž ukazuje použití apply s nepovinnými argumenty.

```
(define make-vec+
  (lambda (n)
    (let ((null-vector (build-list n (lambda (i) 0))))
      (lambda vectors
        (apply map + null-vector vectors)))))
```

Zamysleme se nyní nad možností rozšířit proceduru pro výpočet minima ze dvou prvků na proceduru zpracovávající libovolné argumenty:

```
(define min2
  (lambda (x y)
    (if (<= x y) x y)))
```

Problémem je, že „operace minimum“ se sice chová asociativně, to jest platí

$$\min(a, \min(b, c)) = \min(\min(a, b), c),$$

ale *nemá neutrální prvek*. Nyní máme několik možností, jak postupovat. Jednou z možností je implementovat procedury pro výpočet minima tak, jak je v současném standardu R^{5}RS jazyka Scheme, viz [R5RS]. To jest uvažujeme minimum z jednoho a více čísel:

```
(define min
  (lambda numbers
    (foldr min2 (car numbers) (cdr numbers))))
```

V předchozím kódu jsme jako terminátor zvolili první prvek seznamu čísel, se kterými je procedura min aplikována. Samotnou akumulaci pak provádíme přes seznam předaných čísel bez prvního. Zde jsme vlastně tiše využili i komutativitu sčítání čísel, protože pro seznam obsahující hodnoty $a_1, \ldots, a_n$ počítáme výsledek takto:

$$\min(a_2, \min(a_3, \ldots \min(a_n, a_1) \cdots)).$$

Z těla výše uvedené procedury je taky jasné, že aplikace min bez argumentu by skončila chybovým hlášením způsobeným použitím car a cdr na prázdný seznam.

Druhým způsobem řešení problému je neutrální prvek nějak „dodat“. Například bychom mohli uvažovat symbol +infty, který by nám zastupoval „plus nekonečno“. Tedy jakési nestandardní „největší číslo“. Tento krok by znamenal upravit predikát <= (a v důsledku i další aritmetické procedury) tak, aby pracoval i s touto novou hodnotou. Úprava by mohla vypadat takto, nejprve nadefinujeme +infty, který se bude vyhodnocovat na sebe sama:

```
(define +infty '+infty)
```

Dále vytvoříme novou verzi predikátu porovnávání čísel:

```
(define <=
  (let ((<= <=))
    (lambda (x y)
      (or (equal? y +infty)
          (and (not (equal? x +infty))
               (<= x y))))))
```

Nový <= se na číselných hodnotách chová stejně jako stará verze, pro +infty se chová tak, že +infty je „větší než všechno ostatní“. Viz následující příklady použití.

```
(<= 2 3)            ⟹  #t
(<= 3 2)            ⟹  #f
(<= 2 +infty)       ⟹  #t
(<= +infty 3)       ⟹  #f
(<= +infty +infty)  ⟹  #t
```

Nyní můžeme ponechat kód `min2` tak, jak jej máme, a pouze definujeme novou obecnou verzi `min`:

```
(define min
  (lambda numbers
    (foldr min2 +infty numbers)))
```

Takto definovanou proceduru je možné použít běžným způsobem, nyní i bez argumentů:

```
(min)            ⟹  +infty
(min 30)         ⟹  30
(min 30 10)      ⟹  10
(min 30 10 20)   ⟹  10
```

Nyní se vraťme k reprezentaci množin uvedené v sekci 6.5, kde jsme implementovali konstruktor množiny `list->set`. Ten ze seznamu vytvářel množinu tím, že podle tohoto seznamu odstranil duplicitní výskyty prvků. Pomocí procedury `foldr`, můžeme napsat elegantnější řešení:

```
(define list->set
  (lambda (l)
    (foldr (lambda (x y)
             (if (member? x y)
                 y
                 (cons x y)))
           '()
           l)))
```

Takto nadefinovaná procedura `list->set` provádí akumulaci pomocí `foldr` přes zadaný seznam. Jako terminátor je zvolen prázdný seznam. Vstupní procedura procedury `foldr` pak testuje přítomnost průběžného prvku v seznamu, který vznikl v předchozím kroku (používá se zde predikátu `member?`, který jsme napsali v programu 7.7). Pokud zjistí, že průběžný prvek v seznamu ještě není, přidá tento prvek do seznamu. V opačném případě vrací nezměněný seznam. Tak jsou odstraněny duplicitní výskyty, viz příklady použití.

```
(list->set '())                ⟹  ()
(list->set '(1 2 3))           ⟹  (1 2 3)
(list->set '(1 2 2 1 2 3 1))   ⟹  (2 3 1)
```

Teď se budeme zabývat možným zobecněním procedury `foldr`. Jedním z argumentů procedury `foldr` je ⟨*procedura*⟩. Tato procedura ⟨*procedura*⟩ nese vlastně dvě informace. Říká, jakým způsobem se modifikuje *průběžný prvek* a jakým způsobem se tato modifikace „nabalí“ na *zabalení modifikovaných hodnot za průběžným prvkem*. Vzhledem k tomu můžeme tuto proceduru rozdělit na dvě, tak aby každá z nich obsahovala jen jednu z těchto informací. Například:

- V programu 7.4 jsme definovali proceduru mapování přes jeden seznam `map1`. Proceduru `foldr` jsme aplikovali na proceduru, která je výsledkem vyhodnocení λ-výrazu

  `(lambda (x y) (cons (f x) y))`.

  Procedurou modifikující průběžný prvek je v tomto případě vstupní procedura procedury `map1` navázaná na symbol `f`. Výsledek aplikace této procedury na průběžný prvek pak kombinujeme se zbytkem pomocí konstruktoru `cons`.

- V programu 7.1, ve kterém jsme definovali proceduru `lenght`, předáváme proceduře `foldr` proceduru, která vznikne vyhodnocením výrazu `(lambda (x y) (+ 1 y))`. Argument `x` je v ní ignorován a je místo něj uvažováno číslo `1`. A toto číslo je přičítáno k zabalení zbytku. Rozdělit bychom ji mohli na konstantní proceduru vracející vždy `1` a na primitivní proceduru sčítání navázanou na symbol `+`.

Toto zobecnění napíšeme s pomocí procedury `foldr`. Procedura bude brát čtyři argumenty. Prvním z nich bude procedura `combinator` o dvou argumentech, která bude určovat způsob nabalovaní. Smysl těchto argumentů je v podstatě stejný jako u procedury, která je argumentem procedury `foldr`. Rozdíl je jen v tom, že jí jako první argument není předáván přímo průběžný prvek, ale jeho modifikace. Modifikací myslíme výsledek aplikace procedury `modifier`, která je druhým argumentem procedury `accum`, na průběžný prvek.

```
(define accum
  (lambda (combinator modifier nil l)
    (foldr (lambda (x y)
             (combinator (modifier x) y))
           nil l)))
```

Uvádíme několik příkladů volání této akumulační procedury:

```
(accum + (lambda (x) x) 0 '(1 2 3 4))          ⟹  10
(accum + (lambda (x) (* x x)) 0 '(1 2 3 4))    ⟹  30
(accum + (lambda (x) 1) 0 '(1 2 3 4))          ⟹  4
```

Už jsme naznačili, jak by se pomocí této obecné akumulační procedury daly napsat procedury `map1` a `lenght`. Na závěr ještě ukážeme, jak bychom ji mohli použít k filtrování seznamu. Kombinační procedurou bude spojování seznamu `append`, procedurou modifikující průběžné prvky bude procedura, která v závislosti na platnosti vstupního predikátu vrací buďto prázdný seznam `()`, nebo jednoprvkový seznam obsahující tento průběžný prvek. Jako terminátor použijeme prázdný seznam. Následuje celý kód:

```
(define filter
  (lambda (f l)
    (accum append
           (lambda (x)
             (if (f x)
                 (list x)
                 '()))
           '()
           l)))
```

7.4 Procedura FOLDL

V této sekci se zaměříme na variantu akumulační procedury `foldr`. Jak jsme si již mohli všimnout, `foldr` pracoval tím způsobem, že provedl sérii aplikací procedury dvou argumentů, čímž nám umožnil postupně na sebe „nabalovat" výsledky aplikací. Toto „nabalování" přitom postupovalo směrem doprava:

$(\langle \mathit{procedura} \rangle\ \langle \mathit{prvek}_1 \rangle\ (\langle \mathit{procedura} \rangle\ \langle \mathit{prvek}_2 \rangle\ (\langle \mathit{procedura} \rangle \cdots (\langle \mathit{procedura} \rangle\ \langle \mathit{prvek}_n \rangle\ \langle \mathit{terminátor} \rangle) \cdots)))$.

První aplikace, která je dokončena, je aplikace provedená s posledním prvkem seznamu. Následuje aplikace provedená nad předposledním prvkem seznamu a tak se postupuje až k prvnímu prvku. Tento proces bychom také mohli obrátit. Mohli bychom uvažovat zabalení v tomto směru:

$(\langle \mathit{procedura} \rangle\ (\langle \mathit{procedura} \rangle \cdots (\langle \mathit{procedura} \rangle\ (\langle \mathit{procedura} \rangle\ \langle \mathit{terminátor} \rangle\ \langle \mathit{prvek}_1 \rangle)\ \langle \mathit{prvek}_2 \rangle) \cdots)\ \langle \mathit{prvek}_n \rangle)$,

kdy je jako první aplikována procedura na terminátor a první prvek seznamu, výsledek je použit při aplikaci s druhým prvkem seznamu a tak dále. Jako poslední je provedena aplikace s posledním prvkem seznamu. U procedury `foldr` tedy probíhaly aplikace směrem *zprava* (odtud název *fold right*). U nově uvedeného typu „zabalení" probíhá aplikace směrem *zleva*. Nabízí se tedy uvažovat proceduru vyššího řádku, která by byla duální k `foldr` a prováděla zabalení druhým z uvedených způsobů (zleva). Tuto proceduru nazveme `foldl` (z anglického *fold left*).

Procedura `foldl` bude mít argumenty stejného typu a významu jako měla procedura `foldr`, nebudeme je tedy opakovat. Při své aplikaci provede postupnou sérii aplikací dané procedury na prvky seznamu (směrem zleva), která je ukončena terminátorem. V literatuře [BW88] se lze setkat s různými variantami `foldl`, které se liší tím, jaký význam má první a druhý argument procedury, která je předaná `foldl` jako první argument. Podle [BW88] je výsledkem aplikace

(`foldl` ⟨*procedura*⟩ ⟨*terminátor*⟩ ⟨*seznam*⟩)

série aplikací tvaru

$(\langle\textit{procedura}\rangle\ (\langle\textit{procedura}\rangle \cdots (\langle\textit{procedura}\rangle\ (\langle\textit{procedura}\rangle\ \langle\textit{terminátor}\rangle\ \langle\textit{prvek}_1\rangle)\ \langle\textit{prvek}_2\rangle) \cdots)\ \langle\textit{prvek}_n\rangle)$,

to jest přesně tak, jak jsme naznačili v úvodu sekce. Lze také uvažovat sérii aplikací vypadající takto:

$(\langle\textit{procedura}\rangle\ \langle\textit{prvek}_n\rangle\ (\langle\textit{procedura}\rangle\ \langle\textit{prvek}_{n-1}\rangle\ (\langle\textit{procedura}\rangle \cdots (\langle\textit{procedura}\rangle\ \langle\textit{prvek}_1\rangle\ \langle\textit{terminátor}\rangle) \cdots)))$.

Mezi oběma předchozími sériemi aplikací je zcela zřejmě *jediný rozdíl*. V prvním případě je ⟨*procedura*⟩ aplikována tak, že jejím prvním argumentem je výsledek předchozí akumulace a druhým argumentem je průběžný prvek seznamu. V druhém případě je tomu obráceně: prvním argumentem je průběžný prvek a druhým argumentem je výsledek předchozí akumulace. V obou případech je ale akumulace zahájena od prvního prvku seznamu.

Z toho, co jsme teď uvedli, by mělo být zřejmé, že procedury provádějící výše uvedené „zabalení zleva“ budeme schopni naprogramovat pomocí `foldr` a to v případě obou typů sérií aplikací. Pro druhý typ je to jednodušší, protože stačí použít `foldr` na převrácený seznam. V případě prvního typu pak už jen stačí obrátit argumenty při aplikaci procedury. Procedury provádějící obě zabalení jsou prezentovány v programu 7.9. Procedura pojmenovaná `genuine-foldl` reprezentuje „zabalení zleva“ podle [BW88],

Program 7.9. Procedury `genuine-foldl` a `foldl` vytvořené pomocí `foldr` a reverze seznamu.

```
(define genuine-foldl
  (lambda (f term l)
    (foldr (lambda (x y)
             (f y x))
           term
           (reverse l))))

(define foldl
  (lambda (f term l)
    (foldr f term (reverse l))))
```

tedy první z uvedených typů. Procedura `foldl` reprezentuje druhý z typů. Rozdíly mezi oběma typy zabalení a rozdíl oproti `foldr` si nejlépe uvědomíme na následujícím příkladu. Nejprve nadefinujeme pomocnou proceduru:

```
(define proc
  (lambda (x y)
    (list #f x y))),
```

kterou pak použijeme s týmž seznamem při aplikaci `foldr`, `foldl` a `genuine-foldl`:

```
(define s '(a b c d))
(foldr proc 'base s)          ⟹  (#f a (#f b (#f c (#f d base))))
(foldl proc 'base s)          ⟹  (#f d (#f c (#f b (#f a base))))
(genuine-foldl proc 'base s)  ⟹  (#f (#f (#f (#f base a) b) c) d)
```

Jak vidíme, výsledky aplikaci odpovídají oběma typům, které jsme uvedli v této sekci.

Poznámka 7.2. Jedním ze základních vztahů, který platí mezi `foldr` a `genuine-foldl` je ten, že pokud je při akumulaci použita *monoidální procedura* a jako terminátor je použit její *neutrální prvek*, pak je *výsledek použití* `foldr` *a* `genuine-foldl` *stejný*. Toto pozorování lze jednoduše dokázat.

Jako příklad použití `foldl` si můžeme uvést proceduru `reverse` provádějící otočení seznamu:

```
(define reverse
  (lambda (l)
    (foldl cons '() l)))
```

Tento příklad je poněkud „umělý“, protože v programu 7.9 jsme samotný `foldl` zavedli pomocí `reverse`. Kdybychom to učinili a poté definovali `reverse` předchozím způsobem, při pokusu o jeho aplikaci bychom se dostali do nekončící série aplikací (protože `foldl` aplikuje `reverse` a obráceně). Ukažme tedy o něco přirozenější příklad. V sekci 2.5 jsme představili proceduru vyššího řádu `compose2` vracející, pro dvě vstupní procedury jednoho argumentu, proceduru reprezentující jejich složení, viz příklad 2.6 na straně 61. Nyní bychom mohli pomocí `foldl` naprogramovat složení libovolného množství procedur tak, jak to ukazuje program 7.10. Procedura `compose` je pomocí `foldl` vytvořena přímočaře. Terminátorem je identita,

Program 7.10. Složení libovolného množství procedur pomocí `foldl`.

```
(define compose
  (lambda functions
    (foldl (lambda (f g)
              (lambda (x) (f (g x))))
            (lambda (x) x)
            functions)))
```

což je neutrální prvek vzhledem ke skládání. Procedura dvou argumentů, která je při aplikaci předána `foldl`, provádí složení akumulované hodnoty (procedury vzniklé předchozími složeními) s průběžnou procedurou ze seznamu procedur `functions`. Proč jsme při skládání nepoužili `foldr` jako u všech ostatních procedur v této lekci? Protože jsme chtěli pro seznam procedur reprezentující funkce $f_1, \ldots, f_n$ (v tomto pořadí) vrátit proceduru reprezentující jejich kompozici $f_1 \circ f_2 \circ \cdots \circ f_{n-1} \circ f_n$, která je daná

$$(f_1 \circ f_2 \circ \cdots \circ f_{n-1} \circ f_n)(x) = (f_n(f_{n-1}(\cdots(f_2(f_1(x)))\cdots))),$$

což vede k použití „zabalení zleva“ – skládání je potřeba aplikovat směrem „zepředu“ seznamu (tedy používáme „zabalení zleva“). Vzhledem k tomu, že skládání funkcí na množině je monoidální operace, použití `genuine-foldl` by nám nepomohlo (vedlo by to na stejný výsledek jako použití `foldr`), protože bychom tím provedli složení v opačném pořadí, což je vzhledem k nekomutativitě skládání funkcí problém.

Následující příklady ukazují použití `compose`. Nejprve použijeme pomocné definice

```
(define s '(0 1 2 3 4))
(define f1 (lambda (x) (* 2 x)))
(define f2 (lambda (x) (* x x)))
(define f3 (lambda (x) (+ x 1))),
```

které dále použijeme při skládání pomocí `compose`:

```
(map (compose) s)          ⟾  (0 1 2 3 4)
(map (compose f1) s)       ⟾  (0 2 4 6 8)
(map (compose f1 f2) s)    ⟾  (0 4 16 36 64)
(map (compose f2 f1) s)    ⟾  (0 2 8 18 32)
(map (compose f1 f2 f3) s) ⟾  (1 5 17 37 65)
(map (compose f3 f2 f1) s) ⟾  (2 8 18 32 50)
  ⋮
```

Poznamenejme, že k procedury `genuine-foldl` a `foldl` budeme rovněž chápat jako procedury pracující nad libovolným počtem seznamů (vždy alespoň nad jedním) stejně tak, jako tomu bylo i u procedur `foldr`, `map` a podobně. V programu 7.11 je uvedeno rozšíření těchto procedur tak, aby nepracovaly pouze s jedním seznamem, ale obecně s více seznamy. V obou případech jsme rozšířili seznam argumentů o volitelnou část a v těle jsme provedli explicitní aplikaci `foldr` pomocí `apply`. Následující příklady ukazují činnost obou procedur pro více seznamů:

```
(foldl list 'base '(a b) '(1 2) '(#f #t))          ⟾  (b 2 #t (a 1 #f base))
(genuine-foldl list 'base '(a b) '(1 2) '(#f #t))  ⟾  ((base #f 1 a) #t 2 b)
```

Program 7.11. Procedury genuine-foldl a foldl pracující s libovolným počtem seznamů.

```
(define genuine-foldl
  (lambda (f term . lists)
    (apply foldr
           (lambda args
             (apply f (reverse args)))
           term
           (map reverse lists))))

(define foldl
  (lambda (f term . lists)
    (apply foldr f term (map reverse lists))))
```

Poznámka 7.3. Z posledního příkladu a z implementace genuine-list jsme si mohli všimnout, že proceduru předávanou genuine-list jsme aplikovali s převráceným seznamem argumentů. U procedur více než dvou argumentů ale není jasné, zda-li by se toto „převrácení" mělo týkat všech argumentů nebo jestli bychom pouze neměli dát poslední argument (zastupující akumulovanou hodnotu) na začátek seznamu. Při definici genuine-foldl pracující pouze s jedním seznamem jsme tento problém nemuseli vůbec uvažovat. Také si všimněte, že procedury foldl se tento problém netýká, protože má argumenty pořád ve stejném pořadí. Z tohoto důvodu budeme dále preferovat používání procedury foldl nad procedurou genuine-foldl (nemluvě o tom, že ve Scheme má foldl praktičtější uplatnění).

Procedury foldr a foldl nejsou přítomny ve standardu R^{5}RS jazyka Scheme, ačkoliv některé interprety jazyka Scheme jimi disponují. Tyto procedury jsou přítomny v mnoha funkcionálních programovacích jazycích. Ve Scheme si foldr i foldl můžeme naprogramovat, což ukážeme v dalších lekcích.

7.5 Další příklady na FOLDR a FOLDL

V této sekci uvedeme praktické použití akumulační procedury foldr. Jako první se budeme zabývat procedurou, která bere jako argument libovolný seznam a vrací seznam všech jeho suffixů – včetně prázdného. Použijeme proceduru foldr tímto způsobem: Procedura, která je jejím prvním argumentem, vybere ze seznamu doposud nalezených suffixů první prvek, přidá do něj průběžný prvek seznamu a výsledný seznam přibalí k seznamu nalezených suffixů. Tento seznam obsahuje z počátku jen prázdný seznam, protože prázdný seznam je suffixem jakéhokoli seznamu. Tím je dán druhý argument procedury foldr. Posledním (třetím) argumentem předaným foldr je samotný seznam, jehož suffixy hledáme. Implementace by pak vypadala takto:

```
(define suffixes
  (lambda (l)
    (foldr (lambda (x y)
             (cons (cons x (car y)) y))
           '(())
           l)))
```

Aplikací takto nadefinované procedury dostáváme seznam suffixů seznamu seřazené od nejdelšího po nejkratší (to jest po prázdný seznam):

```
(suffixes '())        ⟹  (())
(suffixes '(1))       ⟹  ((1) ())
(suffixes '(1 2))     ⟹  ((1 2) (2) ())
(suffixes '(1 2 3))   ⟹  ((1 2 3) (2 3) (3) ())
```

Pokud bychom chtěli jen neprázdné suffixy, mohli bychom to udělat mnoha způsoby s použitím procedury `suffixes`, kterou jsme právě nadefinovali. Ze seznamu suffixů, který je výsledkem aplikace této procedury, pak můžeme odstranit prázdný seznam vyfiltrováním neprázdných seznamů, odstraněním posledního prvku, a tak dále. Též bychom mohli použít následující elegantní řešení:

```
(define safe-car
  (lambda (x)
    (if (null? x)
        '()
        (car x))))

 (define suffixes
   (lambda (l)
     (foldr (lambda (x y)
              (cons (cons x (safe-car y)) y))
            '()
            l)))
```

Uvedený program obsahuje definici bezpečné verze selektoru `car`. Tuto bezpečnější verzi `safe-car` jsme popsal i v sekci 5.4. Jinak se nová procedura `suffixes` liší jen použití procedury `safe-car` namísto `car`, a v použití prázdného seznamu jako terminátoru. Použití `safe-car` je důležité při první aplikaci procedury, která je argumentem `foldr`, kdy je aplikována na prázdný seznam.

Použití této procedury dostáváme podobné výsledky jako dříve. Liší se jen v absenci prázdného seznamu v seznamu nalezených sufixů.

```
(suffixes '())          ⟼  ()
(suffixes '(a))         ⟼  ((a))
(suffixes '(a b c))     ⟼  ((a b c) (b c) (c))
(suffixes '(a b c d))   ⟼  ((a b c d) (b c d) (c d) (d))
```

Kdybychom namísto procedury `foldr` použili proceduru `foldl`, nebyl by výsledkem seznam sufixů, ale naopak seznam prefixů. To je samozřejmě způsobeno změnou směru, kterým je akumulace prováděna.

```
(define prefixes
  (lambda (l)
    (foldl (lambda (x y)
             (cons (append (safe-car y) (list x)) y))
           '()
           l)))
```

Takto nadefinovanou procedurou můžeme hledat všechny neprázdné prefixy zadaného seznamu.

```
(prefixes '())          ⟼  ()
(prefixes '(a))         ⟼  (a)
(prefixes '(a b c))     ⟼  ((a b c) (a b) (a))
(prefixes '(a b c d))   ⟼  ((a b c d) (a b c) (a b) (a))
```

7.6 Výpočet faktoriálu a Fibonacciho čísel

Procedury `foldr` a `foldl` lze použít i pro výpočet hodnot matematických funkcí. V této sekci si ukážeme dvě ukázky použití `foldr` při výpočtu *faktoriálu* a *Fibonacciho čísel.* Hned na počátku však řekněme, že příklady reprezentované v této sekci mají spíš „odstrašující charakter“. Jejich smyslem je poukázat na fakt, že i když se nám podařilo pomocí `foldr` vytvořit řadu užitečných a efektivních procedur (s krátkým a přehledným tělem), ne vždy je použití `foldr` na místě.

Připomeňme, že faktoriál $n!$ nezáporného čísla n je definován jako součin přirozených čísel od 1 do n. Neformálně jej tedy lze chápat jako číslo dané

$$n! = \underbrace{1 \cdot 2 \cdot \cdots \cdot n}_{n \text{ činitelů}}.$$

V další sekci ukážeme zavedení faktoriálu, které je z matematického hlediska přesnější. V tuto chvíli si ale vystačíme s touto poněkud neformální definicí. Pro $n = 0, 1, 2, \ldots$ nabývá faktoriál $n!$ následujících hodnot:

$$1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600, \ldots$$

Naším úkolem nyní je naprogramovat proceduru `fac`, která pro daný argument jímž bude nezáporné číslo n, vrátí hodnotu faktoriálu $n!$. Z předchozího je tedy jasné, že naše procedura musí provést součin $1 \cdot 2 \cdot \cdots \cdot n$. Nabízí se tedy pomocí `build-list` nejprve vytvořit seznam činitelů a potom jej vynásobit pomocí `apply` nebo `foldr`. Obě verze jsou uvedeny v programu 7.12. Procedury skutečně počítají to, co

Program 7.12. Výpočet faktoriálu pomocí procedur vyšších řádů.

```
(define fac
  (lambda (n)
    (apply * (build-list n (lambda (x) (+ x 1))))))

(define fac
  (lambda (n)
    (foldr * 1 (build-list n (lambda (x) (+ x 1))))))
```

mají, o čemž se můžeme přesvědčit například vyhodnocením následujícího výrazu:

`(map fac '(0 1 2 3 4 5 6 7 8 9))` $\Longmapsto$ `(1 1 2 6 24 120 720 5040 40320 362880)`

Mnoha čtenářům by se ale mohlo zdát, že procedura `fac` pracuje až příliš složitě na to, že počítá tak jednoduchou funkci jako je faktoriál. To je pravda. Během výpočtu jsme zkonstruovali pomocný seznam, což trvalo n kroků a jeho prvky jsme posléze vynásobili, to trvalo dalších n kroků. Časová složitost výpočtu tedy byla $O(2n)$, při výpočtu jsme navíc konstruovali seznam, který jsme použili pouze jednorázově. Ani kód procedury nebyl tak čitelný, jak bychom (u jednoduché funkce jakou je faktoriál) očekávali. Lepší variantu procedury `fac` ukážeme v další lekci.

Druhým příkladem v této sekci bude procedura pro výpočet prvků Fibonacciho posloupnosti. Fibonacciho posloupnost je tvořena počátečními dvěma prvky $F_0 = 0$, $F_1 = 1$ a každý další prvek posloupnost vzniká součtem předchozích dvou, posloupnost tedy vypadá následovně:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, \ldots$$

Procedura počítající prvky Fibonacciho posloupnosti je uvedená v programu 7.13. Rozeberme si nyní, jak

Program 7.13. Výpočet prvků Fibinacciho posloupnosti pomocí procedur vyšších řádů.

```
(define fib
  (lambda (n)
    (cadr
     (foldr (lambda (x last)
              (list (apply + last) (car last)))
            '(1 0)
            (build-list n (lambda (x) #f))))))
```

je procedura `fib` naprogramovaná. Procedura ve svém těle provádí akumulaci pomocí `foldr`. Při této akumulaci jsou postupně sčítány dvě předchozí Fibonacciho čísla, čímž se získá další prvek posloupnosti. Ten je dál použit s předchozím prvkem k získání dalšího prvku a tak dále. Akumulace probíhá přes seznam vytvořený aplikací `build-list`. Tento seznam má délku n. Samotné prvky seznamu pro nás nebudou mít žádný význam, seznam je použit pouze jako „čítač kroků“ určující, kolikátý člen Fibonacciho posloupnost má být nalezen. Při samotné akumulaci je vždy vytvářen dvouprvkový seznam ve tvaru (F_{i+1} F_i). Tedy druhý prvek seznamu je průběžné Fibonacciho číslo a první prvek seznamu je jeho následník. Pokud máme k dispozici F_i a F_{i+1} můžeme hodnotu F_{i+2} stanovit součtem $F_i + F_{i+1}$, což provádíme při akumulaci, viz explicitní aplikaci procedury sčítání. Výsledkem akumulace v i-tém kroku je tedy vytvoření seznamu tvaru (F_{i+2} F_{i+1}) na základě seznamu (F_{i+1} F_i), který je navázaný na symbol `last` (všimněte si, že argument `x` je ignorován). Terminátorem akumulace je seznam (F_1 F_0), to jest seznam `(1 0)`. Pro dané n je výsledkem akumulace seznam (F_{n+1} F_n), stačí tedy vrátit jeho druhý prvek, což je požadovaný výsledek, to je provedeno aplikací `cadr` na výsledek akumulace. Viz příklad použití procedury:

```
(map fib '(0 1 2 3 4 5 6 7 8 9))  ⟹  (0 1 1 2 3 5 8 13 21 34)
```

Při akumulaci se postupně vytváří dvouprvkové seznamy posledních dvou uvažovaných Fibonacciho čísel. Například při výpočtu `(fib 9)` bude argument `last` při aplikaci procedury předané `foldr` postupně nabývat hodnot `(1 0)`, `(1 1)`, `(2 1)`, `(3 2)`, `(5 3)`, `(8 5)`, `(13 8)`, `(21 13)` a konečně `(34 21)`. Výsledkem akumulace bude tím pádem seznam `(55 34)`, jehož druhý prvek je vrácen jako výsledek výchozí aplikace `(fib 9)`. Netřeba asi zdůrazňovat, že tento způsob výpočtu Fibonacciho čísel je opět dost neefektivní. V první řadě, kód procedury `fib` je dost nestravitelný a jeho úplné pochopení již vyžaduje nějakou chvíli. Během výpočtu je dále konstruována celá řada „odpadních seznamů“, například n-prvkový seznam `(#f #f` $\cdots$, přes který se akumuluje, a jehož hodnoty (paradoxně) nemají žádný význam. Dále v každém kroku konstruujeme dvouprvkový seznam udržující informaci o posledních dvou stanovených Fibonacciho číslech, což také výrazně ubírá na efektivitě.

Čistota kódu (jeho jednoduchost a čitelnost) a jeho efektivita (výkon) jdou v některých případech proti sobě. Výše uvedené příklady však nejsou ani čistě provedené ani efektivní. V další sekci se mimo jiné zaměříme na zefektivnění a čistější naprogramování procedur `fac` a `fib`.

Shrnutí

V této lekci jsme ze zabývali akumulací, což je speciální postupná aplikace procedur. V jazyku Scheme jsme uvažovali dodatečné procedury `foldr` a `foldl`, pomocí nichž lze akumulaci provádět. Ukázali jsme efektivní implementace vybraných procedur pracujících se seznamy, například mapování, filtrace a nahrazování prvků seznamu. Provedli jsme diskusi ohledně jejich časové složitosti i ohledně složitosti jejich původních verzí. Dále jsme se zabývali problematikou rozšiřování procedur dvou argumentů tak, aby pracovaly s libovolným počtem argumentů. Lekci jsme zakončili ukázkou metody výpočtu faktoriálu a Fibonacciho čísel pomocí akumulace.

Pojmy k zapamatování

- akumulace, explicitní aplikace, filtrace,
- zabalení směrem doprava, zabalení směrem doleva,
- rozšíření procedur dvou argumentů na libovolné argumenty,
- efektivní implementace procedur,
- výpočet faktoriálu a Fibonacciho čísel

Nově představené prvky jazyka Scheme

- procedury `foldr`, `foldl` a `genuine-foldl`.

Kontrolní otázky

1. *Čím se od sebe liší* `foldr` *a* `foldr`*?*
2. *Jak probíhá aplikace* `foldr`*?*
3. *K čemu slouží terminátory?*
4. *Jak lze využít* `foldr` *k rozšíření procedury dvou argumentů na libovolný počet argumentů?*
5. *Co pro nás hrálo klíčovou roli při stanovování časové náročnosti procedur?*

6. *Jaký je rozdíl mezi* `foldl` *a* `genuine-foldl`*?*

Cvičení

1. V sekci 7.3 jsme rozšířili proceduru `min2` na proceduru libovolného počtu argumentů. Stejným způsobem rozšiřte proceduru na výběr čísla s extrémní absolutní hodnotou `abs-min`.
2. Napište predikát, který pro posloupnost zjistí, zda je neklesající. Posloupnost bude reprezentovaná seznamem, jehož prvky jsou čísla. Viz příklady aplikace:
```
(nondecreasing? '())          ⟹  #t
(nondecreasing? '(1 2 3 4))   ⟹  #t
(nondecreasing? '(1 2 4 3))   ⟹  #f
(nondecreasing? '(1 4 2 3))   ⟹  #f
(nondecreasing? '(4 1 2 3))   ⟹  #f
```
3. Napište procedury `after` a `before`, jejichž argumenty budou element ⟨*elem*⟩ a seznam ⟨*l*⟩. Procedura `after` bude vracet seznam prvků za posledním výskytem prvku ⟨*elem*⟩ (včetně) v seznamu ⟨*l*⟩. Procedura `before` zase seznam prvků před prvním výskytem prvku ⟨*elem*⟩ (včetně) v seznamu ⟨*l*⟩. Viz příklady použití:
```
(after 10 '(1 2 3 4 3 5 6))    ⟹  ()
(after 3 '(1 2 3 4 3 5 6))     ⟹  (3 5 6)
(after 6 '(1 2 3 4 3 5 6))     ⟹  (6)
(before 10 '(1 2 3 4 3 5 6))   ⟹  (1 2 3 4 5 6)
(before 1 '(1 2 3 4 3 5 6))    ⟹  (1)
(before 3 '(1 2 3 4 3 5 6))    ⟹  (1 2 3)
```

Řešení ke cvičením

1.
```
(define abs-min2
  (lambda (x y)
    (if (<= (abs x) (abs y)) x y)))

(define abs-min
  (lambda args
    (foldr abs-min2 (car args) (cdr args))))
```
2.
```
(define nondecreasing?
  (lambda (l)
    (foldr (lambda (x y)
             (if y
                 (if (or (equal? y #t) (<= x y))
                     x
                     #f)
                 #f))
           #t
           l)))
```
3.
```
(define conseq
  (lambda (elem x y)
    (if (car y)
        y
        (cons (equal? x elem)
              (cons x (cdr y))))))
(define after
  (lambda (elem l)
    (let ((found (foldr (lambda (x y) (conseq elem x y)) '(#f . ()) l)))
      (if (car found)
          (cdr found)
          #f))))
```

```
(define before
  (lambda (elem l)
    (let ((found (foldl (lambda (x y) (conseq elem x y)) '(#f . ()) l)))
      (if (car found)
          (reverse (cdr found))
          #f))))
```

Lekce 8: Rekurze a indukce

Obsah lekce: Tato lekce se věnuje rekurzivním procedurám a výpočetním procesům generovaným rekurzivními procedurami. Dále ukážeme, jak je možné použít princip indukce pro dokázání správnosti definice rekurzivní procedury. Nejprve se zaměříme na rekurzi a princip indukce přes přirozená čísla. Dále ukážeme rekurzi a indukci na seznamech, které hrají (nejen) ve funkcionálním programování klíčovou roli. Během výkladu se budeme zabývat vlastnostmi výpočetních procesů generovaných rekurzivními procedurami, jejich složitostí, průběhem výpočtu a jeho náročností.

Klíčová slova: koncová rekurze, lineární rekurze, princip indukce, rekurze, stromová rekurze, střadače.

8.1 Definice rekurzí a princip indukce

Pojmy *rekurze* a *indukce* jsou jedny z nejdůležitějších pojmů v informatice a to jak teoretické tak aplikované. V této úvodní sekci se budeme oběma pojmy zabývat spíše z matematického pohledu, ale ukážeme jejich silnou vazbu k funkcionálnímu programování a programování obecně. Aby nedošlo hned na počátku k nějakému nedorozumění, upozorněme na fakt, že slovo „rekurze“ má v informatice, ale i v jiných disciplínách (například v logice), *mnoho různých významů.* My se budeme zabývat rekurzí jako *metodou definice funkcí* (ve smyslu matematických funkcí, čili zobrazení) a *procedur.* Indukci budeme používat jako obecný dokazovací princip jímž budeme schopni dokazovat vlastnosti rekurzivně definovaných funkcí a procedur.

V programátorské terminologii je pod pojmem *rekurzivní procedura* obvykle myšlena procedura, která ve svém těle *provádí aplikaci sebe sama.* Tak se na rekurzivní procedury budeme dále v textu dívat i my. Ukážeme, že rekurzí (to jest „aplikací sebe sama“) lze vyřešit mnoho problémů z předchozích lekcí elegantně (co se týče jejich naprogramování a čitelnosti kódu) a efektivně (co se týče jejich výpočetní složitosti). Na úvod také podotkněme, že rekurzivní procedury (procedury aplikující sebe sama) jsou z technického hlediska obyčejné procedury. Při úvahách o rekurzivních procedurách tedy nebudeme muset nijak rozšiřovat modely vyhodnocování elementů ani aplikace procedur. Důvodem, proč se těmto „vlastně obyčejným procedurám“ budeme věnovat několik lekcí je, že programátoři potřebují obvykle delší dobu na úplné pochopení rekurze jako *principu* – od programátora to vyžaduje jistou dávku představivosti a také trpělivosti (zvláště při počátečním seznamování se s problematikou a s řešením prvních příkladů).

Nyní si ukážeme několik definic pomocí rekurze. Abychom si ukázali, že rekurze jako princip není omezená jen na „programování procedur“, budeme si v této sekci ukazovat rekurzivní definice různých zobrazení, se kterými jsme se již setkali v předchozích lekcích. Nebudeme přitom zatím ukazovat přímou vazbu na jazyk Scheme. V dalších sekcích si pak ukážeme souvislost s vytvářením rekurzivních procedur (ve Scheme).

Uveďme si nejprve několik motivačních příkladů. V sekci 7.6 jsme uvedli proceduru pro výpočet faktoriálu daného čísla. Faktoriál $n!$ čísla n jsme popsali poněkud neformálně jako „součin čísel od 1 do n“. Mohli bychom jej však definovat daleko přesněji. Faktoriál lze chápat jako funkci (zobrazení), které každému nezápornému celému číslu n přiřazuje hodnotu $n!$ danou následujícím vztahem:

$$n! = \begin{cases} 1 & \text{pokud } n \leq 1, \\ n \cdot (n-1)! & \text{jinak.} \end{cases}$$

Tento vztah říká, že faktoriál nuly a jedničky je roven jedné (první řádek definičního vztahu). Pro $n \geq 2$ je faktoriál $n!$ definován na druhém řádku jako číslo dané $n \cdot (n-1)!$. Slovně řečeno, faktoriál pro $n \geq 2$ získáme jako „součin n s faktoriálem $n-1$“. V předchozí definici jsme vlastně *zavedli faktoriál čísla n pomocí faktoriálu menšího čísla.* Nejedná se ale o „definici kruhem“, protože faktoriál n je vyjádřen pomocí faktoriálu $n-1$ (tedy pomocí hodnoty jiného faktoriálu, *nikoliv* pomocí *své vlastní hodnoty,* což by vedlo „do kruhu“). Výše uvedená definice $n!$ je prvním příkladem *rekurzivní definice.* V tomto případě tedy rekurzivní definice zobrazení z $\mathbb{N}_0$ (nezáporná celá čísla) do $\mathbb{N}$ (přirozená čísla).

Rozepíšeme-li hodnoty $0!, 1!, 2!, \ldots$ podle předchozí definice, získáme:

$0! = 1,$
$1! = 1,$
$2! = 2 \cdot 1! = 2 \cdot 1 = 2,$

$$
\begin{aligned}
&3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1! = 3 \cdot 2 \cdot 1 = 6,\\
&4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 = 24,\\
&\quad\vdots
\end{aligned}
$$

Všimněte si, že počínaje třetím řádkem můžeme hodnotu faktoriálu jednoznačně spočítat na základě znalosti výsledku z předchozího řádku. To přesně koresponduje s definičním předpisem $n!$ pro $n \geq 2$. Stručněji zapsáno tedy máme:

$$
\begin{aligned}
&0! = 1,\\
&1! = 1,\\
&2! = 2 \cdot 1! = 2 \cdot 1 = 2,\\
&3! = 3 \cdot 2! = 3 \cdot 2 = 6,\\
&4! = 4 \cdot 3! = 4 \cdot 6 = 24,\\
&\quad\vdots
\end{aligned}
$$

Ačkoliv jsme pro n nabývající konkrétních hodnot $n = 0, \ldots, 4$ ukázali, že výsledky $n!$ jsou jednoznačně definované (a mají očekávané hodnoty), nijak jsme zatím neprokázali tento fakt pro *libovolné nezáporné celé* n. Je přirozeně jasné, že nelze udělat „ruční výpis $n!$“ pro každé n, protože nezáporných celých čísel je nekonečně mnoho. Pro ověření správnosti tedy musíme sáhnout po nějakém formálním dokazovacím principu[14]. Správnost našeho zavedení si za chvíli dokážeme pomocí matematické indukce. Ještě předtím však uveďme druhý příklad.

Posloupnost $F_0, F_1, F_2, \ldots$ Fibonacciho čísel, kterou jsme neformálně zavedli v sekci 7.6 jako „posloupnost začínající 0 a 1 a jejíž každý další prvek je součtem předchozích dvou“, bychom nyní mohli přesně zavést pomocí definičního vztahu

$$
F_n = \begin{cases} 0 & \text{pokud } n = 0,\\ 1 & \text{pokud } n = 1,\\ F_{n-1} + F_{n-2} & \text{jinak.} \end{cases}
$$

Nebo pomocí jeho stručnější ekvivalentní varianty

$$
F_n = \begin{cases} n & \text{pokud } n \leq 1,\\ F_{n-1} + F_{n-2} & \text{jinak.} \end{cases}
$$

Jedná se opět o příklad rekurzivní definice, protože jsme hodnotu F_n (to jest n-té Fibonacciho číslo) definovali pomocí hodnot F_{n-2} a F_{n-1}. Je v celku evidentní, že každé F_i je jednoznačně definované a že přiřazení $n \mapsto F_n$ (pro každé nezáporné celé n) je zobrazení z množiny $\mathbf{N}_0$ do množiny $\mathbf{N}_0$, přesto i tento fakt dále dokážeme indukcí. Dosazením do výše uvedeného definičního vztahu můžeme vyjádřit Fibonacciho čísla $F_0, \ldots, F_4, \ldots$ následovně:

$$
\begin{aligned}
&F_0 = 0,\\
&F_1 = 1,\\
&F_2 = F_1 + F_0 = 0 + 1 = 1,\\
&F_3 = F_2 + F_1 = (F_1 + F_0) + F_1 = (1 + 0) + 1 = 2,\\
&F_4 = F_3 + F_2 = (F_2 + F_1) + (F_1 + F_0) = ((F_1 + F_0) + F_1) + (F_1 + F_0) = ((1 + 0) + 1) + (1 + 0) = 3,\\
&\quad\vdots
\end{aligned}
$$

Obdobně jako i u faktoriálu můžeme vidět, že počínaje třetím řádkem můžeme každé Fibonacciho číslo stanovit z hodnot předchozích dvou řádků, to jest:

$$
\begin{aligned}
&F_0 = 0,\\
&F_1 = 1,\\
&F_2 = F_1 + F_0 = 0 + 1 = 1,\\
&F_3 = F_2 + F_1 = 1 + 1 = 2,\\
&F_4 = F_3 + F_2 = 2 + 1 = 3,
\end{aligned}
$$

[14]Někdo by v tomto okamžiku mohl namítat, že „správnost je přece jasná.“ V případě faktoriálu bychom mohli připustit, že správnost jeho rekurzivního zavedení je zřejmá. V praxi je však potřeba definovat rekurzivně daleko složitější zobrazení, u kterých již o správnosti naší vlastní definice nemusíme být „jen tak“ přesvědčeni (správněji: *neměli bychom* být přesvědčeni – pokud ovšem netrpíme obzvláště vyvinutým syndromem „programátorské arogance“), a měli bychom mít tedy k dispozici formální aparát, kterým správnost *prokážeme.*

$$F_5 = F_4 + F_3 = 3 + 2 = 5,$$
$$F_6 = F_5 + F_4 = 5 + 3 = 8,$$
$$\vdots$$

V tuto chvíli bychom mohli říct, že pro princip definice rekurzí je charakteristické, že n-tá hodnota zobrazení (kromě nulté) může být (obecně ale nemusí) definována pomocí hodnoty příslušné některému předchůdci čísla n. Jelikož 0 nemá předchůdce, funkční hodnota pro nulu musí být explicitně definována bez rekurze. V další části této sekce navíc zjistíme, že princip rekurze je možné uplatnit nejen pro definice zobrazení z množin nezáporných celých čísel (do nějakých jiných množin), nýbrž i pro definice obecných zobrazení z množin *hodnot s vhodnou strukturou* (například *seznamů*).

Slovo *rekurze* pochází z latiny a jeho původním významem je „jít zpět“, což dobře koresponduje s tím, jak chápeme rekurzi jako princip definice matematických funkcí (zobrazení): funkční hodnoty pro některá n jsou definovány pomocí funkčních hodnot pro čísla předcházející n. Při výpočtu funkční hodnoty pro n se tedy „jde zpět“ k funkčním hodnotám pro čísla ostře menší než n, jejich funkční hodnoty mohou být opět definovány pomocí funkčních hodnot předchozích čísel, a tak dále. Při výpočtu se tedy postupuje od funkční hodnoty pro n směrem k funkčním hodnotám pro čísla menší než n.

Nyní si ukážeme princip *indukce přes přirozená čísla* (princip *matematické indukce*), který bude dostačovat při dokazování vlastností rekurzivně definovaných funkcí (zobrazení) jakými byly výše uvedené $n!$ (faktoriál) a F_n (Fibonacciho čísel). Z praktických důvodů budeme v dalším výkladu přidávat k množině přirozených čísel i nulu a nebudeme to již všude zdůrazňovat.

V následujícím výkladu budeme pracovat s pojmem „vlastnost přirozeného čísla“. Samotný pojem „vlastnost“ nebudeme příliš formalizovat. Vlastnost budeme značit P. Budeme-li mít dánu vlastnost P, pak budeme vždy předpokládat, že pro každé přirozené číslo n platí: buď (i) *n má/splňuje vlastnost P* (vlastnost P platí pro n), což budeme dále značit $P(n)$, nebo (ii) *n nemá/nesplňuje vlastnost P* (vlastnost P neplatí pro n)[15].

Poznámka 8.1. Jako příklady vlastností přirozených čísel můžeme uvést vlastnost P, která říká: „n je sudé“, dále vlastnost P, která říká „n je prvočíslo“ a podobně. Tyto dvě vlastnosti platí pro některá přirozená čísla a pro jiná neplatí. Třeba vlastnost P: „n je dělitelné jedničkou“ platí pro každé přirozené číslo. V dalším textu pro nás budou důležité právě vlastnosti platné pro každé přirozené číslo (bude se však jednat o vlastnosti netriviálního charakteru dané rekurzivními předpisy funkcí, výrazně složitější než „n je dělitelné jedničkou“, a jejich platnost pro každé n budeme muset prokázat). Pokud bude někdy ve slovním popisu figurovat více čísel, pak by mohlo z vágního popisu (v přirozeném jazyku) dojít k nedorozumění. Například v popisu vlastnosti „číslo n je menší nebo rovno číslu m“ figurují označení „dvou čísel“, není tedy jasné, ke kterému se vlastnost vztahuje (přitom je díky použité nerovnosti čísel podstatný rozdíl v tom, jestli vlastnost vztáhneme k n či k m). V takovém případě budeme vlastnost symbolicky zapisovat ve tvaru

$P(n)$: „$\ldots n \ldots$“,

abychom explicitně vyjádřili, že se v popisu vyjadřuje vlastnost čísla označeného n.

V následující větě je prokázána platnost principu indukce přes přirozená čísla.

Věta 8.2 (princip indukce přes přirozená čísla)**.** *K tomu abychom ověřili, že vlastnost P platí pro každé $n \in \mathbb{N}_0$, stačí prokázat platnost následujících dvou bodů:*

(i) *platí $P(0)$,*

(ii) *pokud platí $P(i)$, pak platí $P(i+1)$.*

[15] Toto neformální chápání vlastností budeme však muset používat velice obezřetně. Snadno bychom totiž mohli vytvořit vlastnost, která by vedla k logickým paradoxům. V tomto kursu však budeme vždy používat vlastnosti, které k nim nevedou. Více informací o problematice paradoxů bude studentům předneseno v rámci kursu *matematické logiky*.

Důkaz. Tvrzení dokážeme sporem. Nechť jsou splněny body (i) a (ii) a zároveň existuje číslo $n \in \mathbf{N}_0$, které nemá vlastnost P. Ze všech čísel, která nemají P můžeme vybrat nejmenší z nich (nejmenší takové číslo vždy existuje, protože množina přirozených čísel je dobře uspořádaná a z předpokladu plyne, že existuje aspoň jedno takové číslo). Označme toto číslo n_0. Pak n_0 nemůže být rovno 0, protože bychom porušili platnost (i). Tím pádem $n_0 \geq 1$. Jelikož jsme n_0 zvolili jako nejmenší z čísel nemajících P, pak $n_0 - 1$ musí mít P. Potom ale dle bodu (ii) i n_0 musí mít P, což je spor. □

Následující příklady ukazují použití principu indukce k prokázání jednoznačnosti a správnost rekurzivních definic faktoriálu a posloupnosti Fibonacciho čísel.

Příklad 8.3. Všimněte si, že principem indukce, viz větu 8.2, nyní můžeme snadno prokázat, že výše uvedená rekurzivní definice $n!$ je skutečně korektní, to jest že každému n přiřazuje jednoznačně hodnotu $n!$, která je součinem čísel od jedné do n. Uvažujme vlastnost P, která říká: „hodnota $n!$ je jednoznačně definovaná“. Pro $n = 0$ platí P zcela jasně, protože je to dáno prvním řádkem definičního vztahu, bod (i) věty 8.2 je tedy pro P splněn triviálně. Ověříme bod (ii). Předpokládejme, že n má vlastnost P, tedy hodnota $n!$ je jednoznačně daná. Pokud je $n = 0$, pak pro $n + 1$ je situace opět pokryta prvním řádkem (a indukční předpoklad v tomto případě na nic nepotřebujeme). Pokud $n > 1$, pak z toho, že $n!$ je jednoznačně daná a z druhého řádku odvodíme

$$(n + 1)! = (n + 1) \cdot (n + 1 - 1)! = (n + 1) \cdot n!,$$

což je jednoznačně daná číselná hodnota vzniklá vynásobením $n!$ (jednoznačně dané) s číslem $n + 1$. Bod (ii) taky platí. Použitím principu indukce jsme tedy prokázali jednoznačnost předchozí rekurzivní definice $n!$. Stejně tak bychom mohli principem indukce dokázat platnost vlastnosti P říkající „$n!$ je násobkem přirozených čísel od 1 do n“.

Příklad 8.4. Analogicky můžeme principem indukce prokázat, že definice F_n je korektní pro každé n. Uvažujme pro tento účel vlastnost $P(n)$: „F_i je jednoznačně definované číslo pro každé nezáporné $i \leq n$“. Evidentně $n = 0$ splňuje P, tedy (i) z věty 8.2 pro P platí. Stejně tak pro $n = 1$ platí P. Z předpokladu, že P platí pro n nyní odvodíme platnost P pro $n + 1$. To, že P platí pro n znamená, že hodnoty všech F_i, kde $i \leq n$, jsou jednoznačně dané. Máme ukázat, že i F_{n+1} je jednoznačně daná. Podle druhého řádku rekurzivní definice Fibonacciho čísel platí:

$$F_{n+1} = F_{n+1-1} + F_{n+1-2} = F_n + F_{n-1}.$$

To jest, F_{n+1} je hodnota vzniklá součtem dvou jednoznačně daných hodnot F_n a F_{n-1} (jejich jednoznačnost je daná induktivním předpokladem), tedy i F_{n+1} je jednoznačně daná hodnota. Pro P tedy platí i bod (ii) věty 8.2, to jest každé F_n je jednoznačně dané.

Všimněte si, že v rekurzivních definicích faktoriálu a Fibonacciho čísel nemůžeme vypustit mezní případy. To jest pro faktoriál případy $0! = 1! = 1$ a pro Fibonacciho čísla $F_0 = 0$ a $F_1 = 1$. Bez nich by totiž definice nebyla úplná (a tím pádem ani jednoznačná). Dále si všimněte, že při definici rekurzí „jdeme zpět“, to jest vyjadřujeme funkční hodnoty pomocí funkčních hodnot definovaných pro předcházející čísla. U principu indukce je tomu naopak. Při prokazování indukcí „jdeme dopředu“.

Rekurze a indukce nemusejí „jít jen přes přirozená čísla“. V předchozích ukázkách jsme použili principy rekurze a indukce díky tomu, že pro každé uvažované nezáporné celé číslo n jsme vždy byli schopni rozlišit dva základní případy:

(i) buď platí $n = 0$,

(ii) nebo je n ve tvaru $m + 1$, kde $m \in \mathbb{N}_0$.

To jest buď bylo dané nezáporné celé číslo nula, nebo bylo následníkem jiného nezáporného celého čísla. Principy rekurze a indukce byly možné právě díky tomu, že množina uvažovaných čísel $\mathbb{N}_0$ (případně vybavená operacemi jako je sčítání a podobně) měla tuto *vhodnou strukturální vlastnost*. Existují však i jiné množiny, u kterých lze přirozeně najít strukturální vlastnosti, které umožňují mít principy rekurze a indukce (v modifikované podobě). Pro nás jako informatiky bude důležitá *množina všech seznamů* (vybavená dalšími operacemi) a principy rekurze a indukce, které půjdou „přes seznamy.“

Abychom se nyní mohli bavit o rekurzi a indukci na seznamech, ale pořád si (zatím) zachovali jistý odstup od programovacího jazyka Scheme, zvolíme následující notaci. Množinu všech *neprázdných seznamů* (chápaných jako elementy) budeme označovat $\mathcal{L}$; množinu všech *seznamů* (chápaných jako elementy) budeme označovat $\mathcal{L}_{()}$; množinu všech elementů budeme označovat $\mathcal{E}$. Máme tedy $\mathcal{L} \subset \mathcal{L}_{()} \subset \mathcal{E}$, přitom $\mathcal{L}_{()} = \mathcal{L} \cup \{\texttt{()}\}$. Samotné seznamy, to jest prvky množiny $\mathcal{L}_{()}$, budeme označovat jako obvykle, i když v tuto chvíli je pro nás mnohem důležitější množina všech seznamů $\mathcal{L}_{()}$ než konkrétní seznamy. Plně v souladu s pojetím procedur jako matematických funkcí (viz sekci 2.5 na straně 57) můžeme nyní chápat konstruktor `cons` a selektory `car` a `cdr` jako následující zobrazení:

$$\begin{aligned}
\mathrm{cons}\colon \mathcal{E} \times \mathcal{L}_{()} &\to \mathcal{L},\\
\mathrm{car}\colon \mathcal{L} &\to \mathcal{E},\\
\mathrm{cdr}\colon \mathcal{L} &\to \mathcal{L}_{()}.
\end{aligned}$$

To jest, cons zobrazuje dvojice $\langle$element, seznam$\rangle$ na neprázdné seznamy (funkční hodnota $\mathrm{cons}(e, l)$ představuje seznam vzniklý připojením elementu e na začátek seznamu l); car zobrazuje neprázdné seznamy na elementy (funkční hodnota $\mathrm{car}(l)$ představuje první prvek seznamu l); cdr zobrazuje neprázdné seznamy na seznamy (funkční hodnota $\mathrm{cdr}(l)$ představuje seznam l bez prvního prvku). Snadno nahlédneme, že procedury `cons`, `car` a `cdr` lze skutečně chápat jako reprezentace zobrazení cons, car a cdr. Pomocí funkce cons můžeme vyjádřit strukturální vlastnost seznamů, kterou budou používat dále uvedené principy rekurze a indukce: pro každý seznam $l \in \mathcal{L}_{()}$ platí, že

(i) buď je l prázdný seznam,

(ii) nebo existuje seznam $k \in \mathcal{L}_{()}$ a element $e \in \mathcal{E}$ tak, že platí $l = \mathrm{cons}(e, k)$.

Nyní bychom mohli uvažovat další procedury, třeba `length` a `append2`, a jim odpovídající zobrazení:

$$\begin{aligned}
\mathrm{append2}\colon \mathcal{L}_{()} \times \mathcal{L}_{()} &\to \mathcal{L}_{()},\\
\mathrm{length}\colon \mathcal{L}_{()} &\to \mathbb{N}_0.
\end{aligned}$$

Zobrazení append2 zobrazuje dvojice seznamů na seznam vzniklý jejich spojením, zobrazení length zobrazuje seznamy na jejich délky.

> Nyní se nám nabízí alternativní pohled na procedury versus zobrazení (matematické funkce). Většinu procedur, se kterými se při programování v Jazyku Scheme setkáváme, lze považovat za *konečné reprezentace* zobrazení, to jest konečné reprezentace *obecně nekonečných matematických objektů*. Funkcionální jazyk považujeme za *čistý*, pokud lze každou primitivní proceduru tohoto jazyka chápat jako reprezentaci zobrazení a v jazyku nelze vytvořit uživatelsky definovanou proceduru, která by nějaké zobrazení nereprezentovala. Jazyk Scheme z tohoto pohledu *čistý není*. To je důsledek mimo jiné toho, že s prostředím manipulujeme jako s elementem prvního řádu a umožňujeme explicitní vyhodnocení elementů relativně vzhledem k prostředí, viz lekci 6. Mezi čisté funkcionální jazyky patří třeba Haskell a Clean.

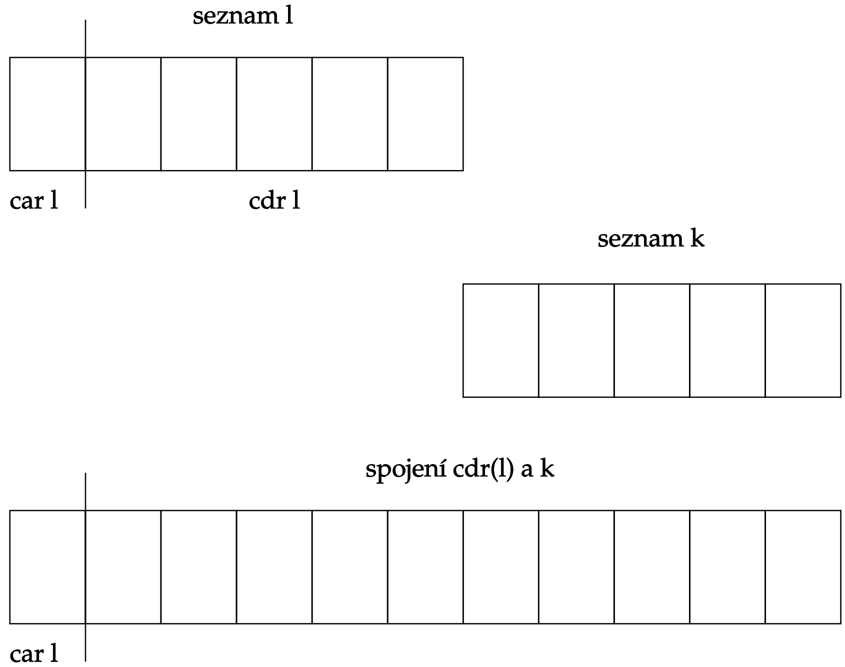
Zobrazení length a append2 můžeme velmi snadno nadefinovat rekurzí přes seznamy. Při tomto typu rekurze jsou funkční hodnoty $f(\ldots, l, \ldots)$, kde l je seznam, vyjádřeny pomocí funkčních hodnot $f(\ldots, k, \ldots)$, kde k je seznam *vyjádřitelný z l pomocí (aspoň jednoho) použití* car *a* cdr. Abychom zpřesnili pojem „být vyjádřitelný pomocí car a cdr", zavedeme následující pojem.

Definice 8.5 (strukturální složitost seznamů)**.** Seznam k se nazývá *strukturálně jednodušší než seznam l*, pokud existují zobrazení $f_1, \ldots, f_k$ ($k \geq 1$) tak, že $\{f_1, \ldots, f_k\} \subseteq \{\mathrm{car}, \mathrm{cdr}\}$ a $k = f_1(f_2(\cdots(f_k(l))\cdots))$. ■

Princip definice rekurzí přes seznamy je tedy založen na tom, že definujeme (funkční) hodnotu pro prázdný seznam, který je ze všech seznamů strukturálně nejjednodušší (neexistuje seznam, který by byl strukturálně jednodušší než prázdný seznam), a dále v případě neprázdných seznamů vytvoříme předpis, který vyjádří (funkční) hodnoty pro tyto seznamy pomocí (funkčních) hodnot seznamů, které jsou strukturálně jednodušší.

Například u délky seznamu můžeme uvažovat takto:

Obrázek 8.1. Schématické zachycení úvahy o spojení dvou seznamů



„Délka prázdného seznamu je nula. Pokud daný seznam není prázdný, pak je ve tvaru $\mathrm{cons}(e, k)$, kde k je opět seznam. Navíc k je vyjádřitelný pomocí cdr ze seznamu l, protože

$$k = \mathrm{cdr}(\mathrm{cons}(e, k)) = \mathrm{cdr}(l).$$

Jelikož je délka seznamu k o jedno menší než délka seznamu l, délku l lze vyjádřit pomocí délky k jako hodnotu $1 + \mathrm{length}(k) = 1 + \mathrm{length}(\mathrm{cdr}(l))$."

Tato úvaha vede na následující rekurzivní definici:

$$\mathrm{length}(l) = \begin{cases} 0 & \text{pokud } l \text{ je prázdný seznam,} \\ 1 + \mathrm{length}(\mathrm{cdr}(l)) & \text{jinak.} \end{cases}$$

Předchozí definice vyjadřuje přesně to, jak jsme slovně length popsali. Zcela v souladu s tím, jak jsme v úvodu naznačili, jsme length definovali pro dva případy: nejprve jsme řekli, co je hodnotu $\mathrm{length}(\texttt{()})$ (co je délkou prázdného seznamu) a pak jsme využili faktu, že druhý prvek každého neprázdného seznamu je opět seznam a můžeme pro něj uvažovat funkční hodnotu length a operovat s ní. Ukažme si funkční hodnoty length v případě některých seznamů:

$$\begin{aligned}
&\mathrm{length}(\texttt{()}) = 0,\\
&\mathrm{length}(\texttt{(a)}) = 1 + \mathrm{length}(\mathrm{cdr}(\texttt{(a)})) = 1 + \mathrm{length}(\texttt{()}) = 1 + 0 = 1,\\
&\mathrm{length}(\texttt{(a b)}) = 1 + \mathrm{length}(\mathrm{cdr}(\texttt{(a b)})) = 1 + \mathrm{length}(\texttt{(b)}) = 1 + (1 + \mathrm{length}(\mathrm{cdr}(\texttt{(b)}))) =\\
&\qquad = 1 + (1 + \mathrm{length}(\texttt{()})) = 1 + (1 + 0) = 2,\\
&\mathrm{length}(\texttt{(a b c)}) = 1 + \mathrm{length}(\mathrm{cdr}(\texttt{(a b c)})) = 1 + \mathrm{length}(\texttt{(b c)}) = 1 + (1 + \mathrm{length}(\mathrm{cdr}(\texttt{(b c)}))) =\\
&\qquad = 1 + (1 + \mathrm{length}(\texttt{(c)})) = 1 + (1 + (1 + \mathrm{length}(\mathrm{cdr}(\texttt{(c)})))) =\\
&\qquad = 1 + (1 + (1 + \mathrm{length}(\texttt{()}))) = 1 + (1 + (1 + 0)) = 3,\\
&\qquad \vdots
\end{aligned}$$

Pro prázdný, jednoprvkový, dvouprvkový a tříprvkový seznamy jsou tedy funkční hodnoty výše definované length očekávané. Jak tomu bude v případě pro libovolný seznam? Stejně jako v případě faktoriálu a Fibonacciho čísel se o tom nemůžeme přesvědčit ručně tím, že „vypíšeme hodnoty" length pro každý seznam (je jich nekonečně mnoho). Opět si ale poradíme tak, že představíme vhodný dokazovací princip a správnost definice pomocí něj prokážeme. Předtím si ale uvedeme ještě rekurzivní definici $\mathrm{append2}$.

V případě $\mathrm{append2}$ můžeme uvažovat takto:

„Pokud spojíme prázdný seznam s libovolným seznamem k, pak je výsledkem spojení seznam k. Pokud je l neprázdný seznam, můžeme uvažovat seznam $j = \mathrm{cdr}(l)$. Pokud spojíme j a k, vznikne nám seznam, který obsahuje všechny prvky z l kromě prvního, následované prvky ze seznamu k. Pokud k tomuto seznamu (to jest, k seznamu rovnajícímu se spojení j a k) připojíme na začátek první prvek z l (pomocí cons), pak jsme získali spojení l a k. Schématicky je úvaha zobrazena na obrázku 8.1."

Ačkoliv je předchozí úvaha možná o něco málo složitější, vede na následující rekurzivní definici:

$$\mathrm{append2}(l,k) = \begin{cases} k & \text{pokud } l \text{ je prázdný seznam,} \\ \mathrm{cons}(\mathrm{car}(l), \mathrm{append2}(\mathrm{cdr}(l),k)) & \text{jinak.} \end{cases}$$

Předchozí definice říká právě to, že (i) spojením prázdného seznamu s druhým seznamem získáme právě druhý seznam (neutralita prázdného seznamu vůči spojení seznamů); (ii) druhý bod definice append2 říká, že v případě, kdy je první seznam neprázdný, stačí spojit první seznam bez prvního prvku s druhým seznamem a k tomuto výsledku připojit na začátek první prvek prvního seznamu. Všimněte si, že v definici funkční hodnoty $\mathrm{append2}(l,k)$ jde rekurze pouze přes l. To jest $\mathrm{append2}(l,k)$ je vyjádřena pomocí $\mathrm{append2}(\mathrm{cdr}(l),k)$, seznam k se nemění. Rekurze přes k (druhý ze spojovaných seznamů) by nám při řešení této konkrétní úlohy k ničemu nebyla. Na tomto příkladu je již možná trochu vidět, že definice rekurzí vyžaduje určitý vhled do problému a cvik. Uveďme si nyní příklad vyjádření spojení dvou seznamů `(a b)` a `(1 2 3)` pomocí výše definovaného append2:

$$\begin{aligned} \mathrm{append2}(\texttt{(a b)}, \texttt{(1 2 3)}) &= \mathrm{cons}(\mathrm{car}(\texttt{(a b)}), \mathrm{append2}(\mathrm{cdr}(\texttt{(a b)}), \texttt{(1 2 3)})) = \\ &= \mathrm{cons}(\texttt{a}, \mathrm{append2}(\texttt{(b)}, \texttt{(1 2 3)})) = \\ &= \mathrm{cons}(\texttt{a}, \mathrm{cons}(\mathrm{car}(\texttt{(b)}), \mathrm{append2}(\mathrm{cdr}(\texttt{(b)}), \texttt{(1 2 3)}))) = \\ &= \mathrm{cons}(\texttt{a}, \mathrm{cons}(\texttt{b}, \mathrm{append2}(\texttt{()}, \texttt{(1 2 3)}))) = \mathrm{cons}(\texttt{a}, \mathrm{cons}(\texttt{b}, \texttt{(1 2 3)})) = \\ &= \mathrm{cons}(\texttt{a}, \texttt{(b 1 2 3)}) = \texttt{(a b 1 2 3)}. \end{aligned}$$

Nyní představíme princip indukce, který je použitelný pro dokazování vlastností zobrazení definovaných rekurzí přes seznamy. Nyní již nemůžeme použít klasickou matematickou indukci, protože ta „jde přes čísla". V případě seznamů budeme používat indukci, který jde přes jejich „strukturu", proto jí budeme říkat *strukturální indukce*. Analogicky rekurzi přes seznamy budeme říkat *strukturální rekurze.*

Věta 8.6 (princip strukturální indukce přes seznamy)**.** *K tomu abychom ověřili, že vlastnost P platí pro každý seznam $l \in \mathcal{L}_{()}$, stačí prokázat platnost následujících dvou bodů:*

(i) *platí $P(\texttt{()})$, to jest P platí pro prázdný seznam,*

(ii) *pokud platí $P(l)$, pak pro každý element e platí $P(\mathrm{cons}(e,l))$.*

Důkaz. Tvrzení dokážeme opět sporem. Nechťjsou splněny body (i) a (ii) a zároveň existuje seznam l, který nemá vlastnost P. Ze všech seznamů, které nemají P vybereme seznam s minimální délkou a označíme jej l_0. Upozorněme na to, že obecně může existovat více seznamů se stejnou minimální délkou, které nemají vlastnost P. Vybraný seznam l_0 tedy splňuje vlastnost, že každý (ostře) kratší seznam má vlastnost P. Zcela evidentně l_0 musí být neprázdný seznam, jinak by byl porušen bod (i). Jelikož je l_0 neprázdný, je to seznam, který je výsledkem $\mathrm{cons}(e,l')$, pro nějaký element e a seznam l'. Seznam l' je o jeden element kratší než seznam l_0, má tedy (ostře) menší délku. Tím pádem l' musí mít vlastnost P. Potom dle bodu (ii) i $l_0 = \mathrm{cons}(e,l')$ musí mít vlastnost P, což je spor. □

Příklad 8.7. Zobrazení $\mathrm{length}\colon \mathcal{L}_{()} \to \mathbb{N}_0$ je jednoznačně definované a jeho hodnoty jsou délky seznamů. To jest pro každý seznam l je $\mathrm{length}(l)$ rovno délce seznamu l, tak jak jsme ji doposud chápali. Toto tvrzení můžeme dokázat principem strukturální indukce z věty 8.6. Vskutku, uvažujme vlastnost $P(l)$: „Délka seznamu l je rovna $\mathrm{length}(l)$." Pro prázdný seznam `()` máme dle prvního bodu definice $\mathrm{length}(l) = 0$, tedy délka prázdného seznamu je nula. To jest, prázdný seznam má vlastnost P, bod (i) věty 8.6 je pro P splněn. Předpokládejme, že l má vlastnost P. To znamená, že $\mathrm{length}(l)$ je délka seznamu l. Pro to, abychom ověřili (ii) musíme pro každý element e ukázat, že seznam $\mathrm{cons}(e,l)$ má vlastnost P. Jelikož je seznam $\mathrm{cons}(e,l)$ neprázdný, z druhého bodu definice length a ze zřejmého faktu $\mathrm{cdr}(\mathrm{cons}(e,l)) = l$ dostáváme:

$$\mathrm{length}(\mathrm{cons}(e,l)) = 1 + \mathrm{length}(\mathrm{cdr}(\mathrm{cons}(e,l))) = 1 + \mathrm{length}(l).$$

To jest $\mathrm{length}(\mathrm{cons}(e,l))$ je rovno délce seznamu l zvětšené o 1. Jelikož je $\mathrm{cons}(e,l)$ seznam vzniklý z l

připojením elementu e na začátek l, dostáváme, že $\mathrm{length}(\mathrm{cons}(e,l))$ je délkou seznamu $\mathrm{cons}(e,l)$, což znamená, že $\mathrm{cons}(e,l)$ má vlastnost P. Tím jsme dokončili důkaz bodu (ii) pro P. Z věty 8.6 tedy okamžitě dostáváme důkaz správnosti definice length.

Příklad 8.8. Správnost definice $\mathrm{append2}$ můžeme prokázat podobně jako jsme prokázali správnost definice length. Nejprve si ale musíme uvědomit, přes který seznam budeme indukci provádět, protože $\mathrm{append2}$ je zobrazení typu $\mathrm{append2}\colon \mathcal{L}_{()} \times \mathcal{L}_{()} \to \mathcal{L}_{()}$. Pokud se podíváme na definici, pak vidíme, že $\mathrm{append2}(l,k)$ je v netriviálním případě vyjádřen pomocí konstrukce obsahující $\mathrm{append2}(\mathrm{cdr}(l),k)$. Se strukturou druhého seznamu (to jest seznamu k) se v definici nijak neoperuje. To nám napovídá, že indukci bychom měli vést přes seznam l. Uvažujme tedy vlastnost $P(l)$: „Pro každý seznam k platí: spojení seznamů l a k (v tomto pořadí) je rovno $\mathrm{append2}(l,k)$.“ Prokážeme, že každý seznam l má vlastnost P. Pokud je l prázdný, pak zřejmě $\mathrm{append2}(l,k)=k$, takže bod (i) věty 8.6 pro vlastnost P je triviálně splněn. Předpokládejme, že l má vlastnost P. To znamená, že pro každý seznam k platí, že $\mathrm{append2}(l,k)$ je seznam vzniklý spojením l a k. Nyní prokážeme, že každý seznam $\mathrm{cons}(e,l)$, kde e je libovolný element, má vlastnost P. Seznam $\mathrm{cons}(e,l)$ je neprázdný, tedy dle druhého bodu definice $\mathrm{append2}$ a s využitím

$$\mathrm{car}(\mathrm{cons}(e,l)) = e,$$
$$\mathrm{cdr}(\mathrm{cons}(e,l)) = l,$$

můžeme vyjádřit

$$\mathrm{append2}(\mathrm{cons}(e,l),k) = \mathrm{cons}(\mathrm{car}(\mathrm{cons}(e,l)),\mathrm{append2}(\mathrm{cdr}(\mathrm{cons}(e,l)),k)) = \mathrm{cons}(e,\mathrm{append2}(l,k)).$$

Předchozí rovnost říká, že spojení seznamu $\mathrm{cons}(e,l)$ se seznamem k je rovno připojení elementu e na začátek seznamu $\mathrm{append2}(l,k)$. Dle indukčního předpokladu je $\mathrm{append2}(l,k)$ výsledek spojení seznamu l a k. To jest $\mathrm{append2}(\mathrm{cons}(e,l),k)$ je rovno výsledku připojení elementu e na začátek spojení seznamů l a k. Jinými slovy, $\mathrm{append2}(\mathrm{cons}(e,l),k)$ je spojení seznamu $\mathrm{cons}(e,l)$ se seznamem k. Máme hotov důkaz bodu (ii) pro vlastnost P. Z věty 8.6 dostáváme, že výše uvedená definice $\mathrm{append2}$ je korektní definice spojení dvou seznamů.

Poznámka 8.9. Na předchozích důkazech je zajímavé, že jsme prokázali vlastnosti nově definovaných zobrazení pracujících se seznamy, aniž bychom se zabývali tím, jak jsou seznamy reprezentovány (jak vypadají prvky množiny $\mathcal{L}_{()}$). Vše co nám stačilo, byl fakt, že seznam je buď prázdný, nebo jej lze chápat jako funkční hodnotu $\mathrm{cons}(e,l)$. Z pohledu strukturální indukce je tedy stěžejní právě tato vlastnost, nikoliv to, zda-li chápeme seznamu jako „elementy jazyka konstruované z párů“ (viz lekci 4 a lekci 5) nebo třeba nějak úplně jinak. Se seznamy jsme v této sekci pracovali jako s prvky množiny $\mathcal{L}_{()}$, které jsme vyjadřovali pomocí funkčních hodnot zobrazení cons, car a cdr.

Mezní podmínky v rekurzivních definicích length a $\mathrm{append2}$ opět nelze vynechat, protože takové definici by nebyly kompletní. Všimněte si, že analogicky jako v případě rekurze a indukce přes čísla, jde princip strukturální rekurze směrem „zpět“, protože funkční hodnoty definovaných funkcí jsou vyjádřeny pomocí funkčních hodnot pro strukturálně jednodušší seznamy. Naopak, princip indukce postupuje od strukturálně nejjednoduššího seznamu – prázdného seznamu, směrem „dopředu“, to jest ke složitějším seznamům.

Úkolem této sekce bylo představit principy rekurze a indukce přes čísla a přes seznamy. Ukázali jsme, že rekurzí lze definovat důležitá zobrazení. S programováním to souvisí tak, že analogický princip, jako jsme použili při definování zobrazení, můžeme použít při vytváření procedur, které tato zobrazení reprezentují. Takovým procedurám budeme říkal rekurzivní procedury a blíže se jim budeme věnovat v dalších sekcích. Princip indukce je pro nás důležitým dokazovacím principem, kterým můžeme dokazovat vlastnosti rekurzivně definovaných zobrazení (a procedur, které je reprezentují).

Abychom ještě na závěr sekce demonstrovali sílu definic rekurzí, ukážeme rekurzivní definici zobrazení korespondujícím s procedurou `foldr` představenou v předchozí lekci. Vzpomeňme, že pomocí `foldr` jsme byli schopni vytvořit řadu procedur počínaje `length`, `append2`, přes filtrační proceduru `filter` a tak dále. Doposud jsme ale neřekli, zda-li je možné proceduru `foldr` v jazyky Scheme uživatelsky definovat, nebo jestli musí být přítomna v jazyku jako primitivní procedura. Odpověď je, že procedura *je* definovatelná plně prostředky jazyka, které již máme k dispozici. Definicí procedury se teď zabývat nebudeme, tu ukážeme v dalších částech textu, ale ukážeme rekurzivní definici zobrazení foldr, které je reprezentované procedurou `foldr`. Zobrazení foldr lze chápat jako zobrazení

$$\mathrm{foldr}\colon \mathcal{F} \times \mathcal{E} \times \mathcal{L}_{()} \to \mathcal{E},$$

kde $\mathcal{F}$ je množina všech zobrazení $f\colon \mathcal{E} \times \mathcal{E} \to \mathcal{E}$. Nyní můžeme definovat:

$$\mathrm{foldr}(f,t,l) = \begin{cases} t & \text{pokud } l \text{ je prázdný seznam,} \\ f(\mathrm{car}(l), \mathrm{foldr}(f,t,\mathrm{cdr}(l))) & \text{jinak.} \end{cases}$$

Strukturální indukcí se můžete přesvědčit, že definice je jednoznačná, a že pro seznam l délky n bude funkční hodnota $\mathrm{foldr}(f,t,l)$ získána pomocí „postupného zabalení“ funkcí f tak, jak jsme popsali v předchozí lekci. Snadno potom můžeme vidět, že length a append2 můžeme definovat pomocí foldr jako

$$\begin{aligned} \mathrm{length}(l) &= \mathrm{foldr}(g, 0, l), \text{ kde } g(x,y) = 1 + y, \\ \mathrm{append2}(l,k) &= \mathrm{foldr}(\mathrm{cons}, k, l). \end{aligned}$$

Všimněte si korespondence předchozích zavedení length a append2 s definicemi procedur `length` a `append2` v programech 7.1 a 7.2 na stranách 174 a 175. Tyto ukázky by nám měly dát jakousi představu o tom, že strukturální rekurze je skutečně silným nástrojem (který je potřeba umět správně používat).

Rekurze a indukce nejsou jen nějaké „matematické kuriozity v programování“, jedná se o široce využívané principy bez nichž by byla naše schopnost řešit problémy výrazně snížena. Některé typy úloh lze bez rekurze řešit jen velmi obtížně. Ve funkcionálních jazycích je tradičně rekurze vedle procedur vyšších řádů jednou z nejpoužívanějších metod vytváření procedur (a v důsledku výpočetních procesů). Ve funkcionálních jazycích rekurze de facto nahrazuje *cykly*, které se používají hlavně v procedurálních jazycích. Rekurze je však na rozdíl od prostých cyklů mnohem mocnější, jak záhy uvidíme.

V této sekci jsme ukázali řadu důkazů správnosti definic a vlastností definovaných zobrazení. Budeme v tom pokračovat v omezené míře i v dalších sekcích. V praxi zpravidla není potřeba dokazovat správnost tímto detailním způsobem, protože zkušený programátor má již některé typické konstrukce tak říkajíc „v oku“ a jejich správnost je schopen velmi rychle „vidět“. Zdůrazněme však, že tento nadhled přichází až s určitou programátorskou zkušeností, jejíž nabytí chvíli trvá. Ani potom bychom však formální aparát, který jsme představili v této sekci, neměli považovat za „cosi zbytečného“, ale spíš za užitečnou pomůcku.

8.2 Rekurze a indukce přes přirozená čísla

V této sekci se budeme zabývat rekurzí přes přirozená čísla (ke kterým z technických důvodů přidáváme i nulu). Ukážeme, jak souvisí rekurzivní definice zobrazení, které jsme představili v předchozí sekci, s procedurami, které tato zobrazení reprezentují. Jako první příklad budeme uvažovat rekurzivní definici n-té mocniny čísla. Pro jednoduchost budeme uvažovat pouze případy, kdy n nabývá celočíselné nezáporné hodnoty. V tomto případě můžeme pro libovolné $x \in \mathbb{R}$ definovat x^n následujícím předpisem:

$$x^n = \begin{cases} 1 & \text{pokud } n = 0, \\ x \cdot x^{n-1} & \text{jinak.} \end{cases}$$

Předchozí předpis říká, že $x^0 = 1$ a pokud je $n \geq 1$, pak je $x^n = x \cdot x^{n-1}$. Principem prezentovaným ve větě 8.2 bychom opět mohli snadno dokázat platnost vlastností „pro n je hodnota x^n jednoznačně definovaná“ a navíc je zřejmé, že se jedná skutečně o n-tou mocninu x. Upozorněme na fakt, že předchozí rekurzivní definice definuje pro $x = 0$ hodnotu $x^0 = 1$, což je v rozporu s matematickým chápáním nutné mocniny (z matematického pohledu není hodnota 0^0 definovaná). Z praktického (programátorského) pohledu je však vhodné zavést 0^0 jako 1.

Proceduru `expt` počítající hodnoty x^n podle výše uvedeného rekurzivního předpisu bychom v jazyku Scheme mohli naprogramovat tak, jak je to uvedeno v programu 8.1. Procedura `expt` v programu 8.1 je formalizací předchozího rekurzivního předpisu v jazyku Scheme. V těle procedury je pomocí speciální formy `if` vyjádřen podmíněný výraz: „Pokud je n rovno nule, pak je výsledek umocnění jedna. V opačném případě je výsledek umocnění roven součinu hodnoty x s hodnotou x umocněnou na $n - 1$.“

Program 8.1. Rekurzivní procedura počítající x^n.

```
(define expt
  (lambda (x n)
    (if (= n 0)
        1
        (* x (expt x (- n 1))))))
```

Definice 8.10 (rekurzivní procedura, rekurzivní aplikace)**.** Proceduře budeme říkat *rekurzivní procedura,* pokud při vyhodnocení jejího těla dochází (v některých případech) k aplikaci sebe sama. Aplikaci „sebe sama“ budeme dále nazývat *rekurzivní aplikace procedury.* ■

Procedura `expt` je tedy prvním příkladem rekurzivní procedury, protože v posledním argumentu předaném speciální formě `if`, který je vyhodnocen v případě, že podmínka (první argument předaný `if`) bude nepravdivá, dojde k aplikaci `expt`. Jak již ale bylo řečeno, rekurzivní procedury jsou procedury v klasickém smyslu tak, jak jsme je popsali v úvodních lekcích tohoto textu (není tedy na nich nic „speciálního“).

Poznámka 8.11. Otázkou, kterou bychom se měli zabývat je, proč vlastně aplikace rekurzivních procedur „funguje“. Jinými slovy, je z pohledu vyhodnocovacího procesu skutečně v pořádku, že procedura aplikuje sebe sama? Podíváme-li se na kód procedury `expt` v programu 8.1, pak je zřejmé, že procedura vzniklá vyhodnocením λ-výrazu vznikla v *globálním prostředí.* V tomto prostředí je i provedena vazba této procedury na symbol `expt`. Při aplikaci procedury `expt` je vytvořeno lokální prostředí, jehož předkem je prostředí vzniku procedury, tedy globální prostředí. V lokálním prostředí jsou při aplikaci procedury definovány vazby symbolů `x` a `n`. Pokud při vyhodnocování těla během aplikaci dojde k vyhodnocení symbolu `expt`, pak tento symbol není nalezen v lokálním prostředí. Hledáním vazby se proto postupuje v nadřazeném prostředí, což je globální prostředí – v něm má symbol `expt` vazbu a tou je právě aplikovaná procedura. Procedura vzniklá vyhodnocením λ-výrazu v programu 8.1 má tedy skutečně k dispozici „sebe sama“ prostřednictvím vazby symbolu `expt`.

Na rekurzivní proceduře `expt` si můžeme všimnout dvou částí:

limitní podmínka rekurze je podmínka, po jejímž splnění je vyhodnocen výraz jež nezpůsobí další aplikaci samotné rekurzivní procedury. Na limitní podmínku rekurze se lze dívat jako na podmínku vymezující *triviální případy aplikace procedury,* v jejichž případě není potřeba pro stanovení výsledku aplikace provádět rekurzivní aplikaci. V případě procedury `expt` je limitní podmínkou rekurze fakt, že `n` je rovno nule, v tomto případe je výsledek vyhodnocení číslo `1`, což koresponduje s faktem $x^0 = 1$. Každá rekurzivní procedura by měla mít alespoň jednu limitní podmínku, obecně jich může mít i víc.

předpis rekurze je část těla procedury, při jejímž vyhodnocení dochází k rekurzivní aplikaci procedury. Součástí předpisu rekurze je vyjádření, jak bude stanoven výsledek aplikace rekurzivní procedury s jistými argumenty pomocí rekurzivní aplikace této procedury s jinými argumenty (obvykle s argumenty, které jsou ve smyslu konkrétního použití procedury „jednodušší“). V případě procedury `expt` je rekurzivní předpis `(* x (expt x (- n 1)))`, což koresponduje s faktem, že $x^n = x \cdot x^{n-1}$ pro $n \geq 1$ Rekurzivních předpisů může být v proceduře opět víc, ale z principu by měl být aspoň jeden, abychom se mohli „o rekurzi vůbec bavit“.

Limitní podmínku i předpis rekurze lze snadno vyčíst přímo z rekurzivní definice x^n tak, jak jsme ji uvedli na začátku sekce. Při vytváření rekurzivních procedur je vždy potřeba si limitní podmínky a předpisy rekurze jasně uvědomit a správně formalizovat (v programu). Absence některé z důležitých částí v rekurzivní proceduře, třeba absence limitní podmínky, obvykle vede na *nekončící sérii aplikací* nebo na *vznik chyby.*

Nyní si uvedeme příklad aplikace procedury `expt` s číselnými argumenty `8` a `4` (v tomto pořadí). Aplikace procedury s těmito argumenty je znázorněna na obrázku 8.2 (vlevo). Nejprve si všimněte metody, kterou jsme při znázornění zvolili. Každou novou aplikaci procedury jsme vyjádřili tak, že do těla výrazu jsme

Obrázek 8.2. Schématické zachycení aplikace procedury `expt`.

`(expt 8 4)`	vyvolání 1. aplikace procedury
`(* 8 (expt 8 3))`	*navíjení*: vyvolání 2. aplikace
`(* 8 (* 8 (expt 8 2)))`	*navíjení*: vyvolání 3. aplikace
`(* 8 (* 8 (* 8 (expt 8 1))))`	*navíjení*: vyvolání 4. aplikace
`(* 8 (* 8 (* 8 (* 8 (expt 8 0)))))`	*navíjení*: vyvolání 5. aplikace
`(* 8 (* 8 (* 8 (* 8 1))))`	*dosažení limitní podmínky* v průběhu 5. aplikace
`(* 8 (* 8 (* 8 8)))`	stav po *odvinutí* 4. aplikace
`(* 8 (* 8 64))`	stav po *odvinutí* 3. aplikace
`(* 8 512)`	stav po *odvinutí* 2. aplikace
`4096`	*výsledná hodnota* vzniklá *odvinutím* 1. aplikace

místo seznamu, jehož vyhodnocení aplikaci způsobilo, zapsali *tělo aplikované procedury*, v němž jsme zaměnili *formální argumenty* za *hodnoty předaných argumentů*. V prvním kroku jsme tedy výraz `(expt 8 4)` nahradili výrazem `(* 8 (expt 8 3))`, v dalším kroku jsme výraz `(expt 8 3)` v `(* 8 (expt 8 3))` nahradili výrazem `(* 8 (expt 8 2))` čímž vznikl výraz `(* 8 (* 8 (expt 8 2)))` a tak dále. Jak je z obrázku patrné, při výpočtu je v první fází prováděna rekurzivní aplikace, protože pro hodnoty `4`, . . . , `1` navázané na symbolu `n` se neuplatňuje limitní podmínka rekurze. Této fázi výpočtu, při které dochází k postupné aplikaci rekurzivního předpisu, se říká „navíjení“. Navíjení je ukončeno dosažením limitní podmínky, viz obrázek 8.2 (vpravo). V našem případě to je když je na `n` navázaná hodnota `0`. Od tohoto okamžiku se již procedura `expt` neaplikuje, ale dochází ke „zpětnému dosazování“ vypočtených hodnot a dokončování jednotlivých aplikací `expt`, které byly zahájeny během fáze navíjení. Této druhé fázi říkáme příznačně fáze „odvíjení“. Výsledná hodnota vzniká odvinutím první aplikace `expt` (první aplikace je pochopitelně odvinuta jako poslední, protože se ve fázi odvíjení pohybujeme zpětně).

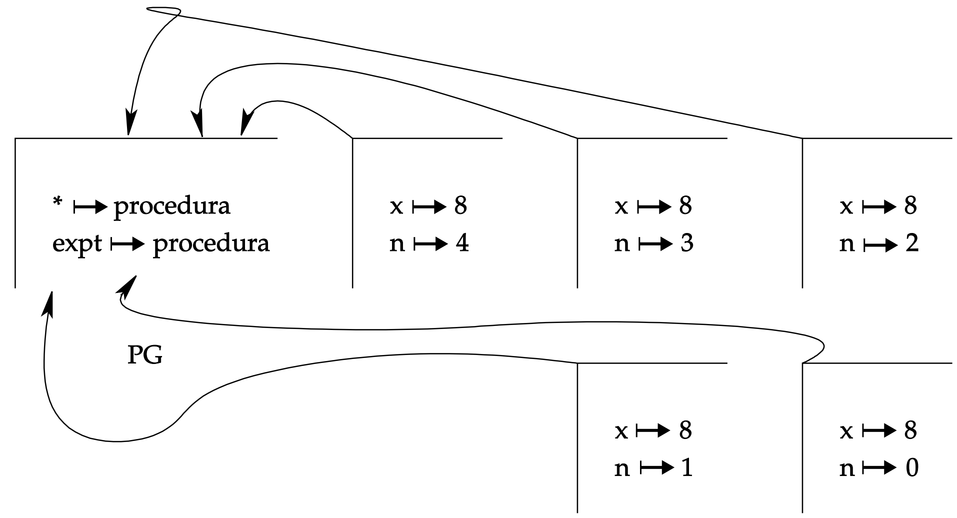
Při aplikaci `expt` s argumenty `8` a `4` došlo de facto k pěti aplikacím této procedury (všechny proběhly ve fází navíjení). Při těchto aplikacích vzniklo tím pádem pět prostředí. Otázkou by možná mohlo být, jestli to není nějaký „nadbytečný luxus“. V žádném případě tomu tak není. Během fáze navíjení je, neformálně řečeno, budována *série odložených výpočtů*. Výsledek `(expt 8 4)` je definován jako součin čísla `8` s hodnotou vzniknou aplikací `(expt 8 3)`. Tento součin však nemůže být dokončen dřív, než je dokončena aplikace `expt` s argumenty `8` a `3`. Do té doby musí být někde uloženy hodnoty, které je potřeba mít k dispozici pro dokončení výpočtu, až bude výsledek vyhodnocení `(expt 8 3)` znám. Tímto úložištěm jsou právě *prostředí*. Na obrázku 8.3 jsou zachycena prostředí vzniklá při vyhodnocení `(expt 8 4)`, předkem všech těchto prostředí je globální prostředí, v němž je na symbol `expt` navázána rekurzivně aplikovaná procedura.

Jak jsme již na příkladu vysvětlili, průběh výpočetního procesu generovaného aplikací rekurzivní procedury můžeme shrnout do dvou důležitých fází:

fáze navíjení je fáze, ve které dochází k postupné rekurzivní aplikaci. V průběhu této fáze jsou vytvářena nová prostředí v nichž jsou uloženy informace o vazbách formálních argumentů rekurzivně aplikované procedury. Tato prostředí v sobě udržují informaci o pomyslném *odloženém výpočtu* (anglicky *deferred computation*). Fáze navíjení končí dosažením limitní podmínky rekurze, v jejímž případě je vrácena hodnota vzniklá bez dalších rekurzivních aplikací procedury.

fáze odvíjení Nastává po dosažení limitní podmínky rekurze. Během této fáze dochází k dokončení vyhodnocení těla procedury v prostředích, která vznikla v předchozí fázi navíjení. Postupuje se přitom zpětně, jako první je dokončeno vyhodnocení těla v prostředí poslední aplikace procedury, následuje vyhodnocení těla v prostředí předposlední aplikace procedury a tak se postupuje až k vyhodnocení těla v prostředí první aplikace procedury a výsledná hodnota je výsledkem původní aplikace. Během fáze odvíjení může v některých případech *znovu nastat fáze navíjení* (uvidíme až na dalších příkladech).

Obrázek 8.3. Prostředí vzniklá během vyhodnocení `(expt 8 4)`



Rekurzivní procedury jsou procedury, které ve svém těle vyvolávají aplikaci sebe sama. Každá rekurzivní procedura má svou limitní podmínku a předpis rekurze. Aplikaci rekurzivní procedury si můžeme představit jako proces probíhající ve dvou základních fázích, které se mohou vzájemně střídat: fáze navíjení a fáze odvíjení. Na rozdíl od jazyka Scheme, některé funkcionální jazyky disponují tak zvaným *líným vyhodnocováním*. Jde o princip vyhodnocování výrazů, který mimo jiné umožňuje definovat rekurzivní procedury bez limitní podmínky, při jejichž aplikaci výpočetní proces neuvízne v sérii nekončících aplikací (právě díky línému vyhodnocování). Jazyk Scheme *není líný*, a každá rekurzivní procedura by tedy limitní podmínku mít měla (pokud nechceme záměrně výpočetní proces „vytuhnout“). V další části učebního textu však ukážeme, že líné vyhodnocování můžeme ve Scheme implementovat, takže předchozí tvrzení tak úplně neplatí.

V tuto chvíli by nám mělo být jasné, jak vypadají rekurzivní procedury, jak je psát, a také „proč vlastně fungují“. Nyní si ukážeme několik dalších rekurzivních procedur pracujících s čísly. Jako první se zamyslíme nad zefektivněním stávající procedury `expt`. Procedura `expt` uvedená v programu 8.1 je naprogramována tak, že při své aplikaci redukuje problém nalezení x^n na problém nalezení x^{n-1}. Je tedy jasné, že pro výpočet x^n vznikne $n+1$ prostředí, protože limitní podmínka je dána pro $n = 0$. Časová složitost výpočtu x^n (touto procedurou) je tedy $O(n)$, hodnota x nehraje (teoreticky) roli[16]. Vzhledem k tomu, že při výpočtu vznikne $n+1$ nových prostředí, můžeme říct, že prostorová složitost výpočtu x^n (touto procedurou) je také $O(n)$ (počet prostředí roste lineárně s n). Otázkou je, zda-li bychom proceduru nemohli naprogramovat efektivněji (z hlediska časového, prostorového nebo z hlediska obou složitostí).

Při navrhování rekurzivních procedur se často uplatňuje princip, který je v informatice označován jako *divide et impera* neboli „rozděl a panuj“. Tento princip spočívá v tom, že řešení problému je rozloženo na řešení (obecně několika) podproblémů stejného typu, ale výrazně menší složitosti. Rozložení problému na menší podproblémy se v programátorské terminologii nazývá *dekompozice.* Až jsou tyto podproblémy vyřešeny, je z jejich řešení (v rozumném čase) složeno řešení výchozího problému. Tento způsob nahlížení na řešení problémů vede většinou na vytváření rekurzivních procedur. Příklady použití principu *divide et impera* uvidíme v dalších sekcích. Už při výpočtu x^n si ale ukážeme, že při dekompozici problému můžeme postupovat výrazně efektivněji než v programu 8.1.

[16]Samozřejmě, že prakticky hraje roli i hodnota x, protože mocnění velkých celých čísel v jejich přesné reprezentaci bude jistě náročnější než mocnění malých čísel. Tento technický rys však můžeme nyní při stanovování rámcové složitosti přehlédnout, protože při ní jde o to stanovit složitost vzhledem k počtu aplikací násobení. Samotnou délku násobení chápeme zjednodušeně jako konstantní.

Výpočet x^n bychom mohli urychlit tím, že si uvědomíme, že $x^{2n} = x^n \cdot x^n = (x^n)^2$. To jest, pokud je exponent sudé číslo, můžeme problém výpočtu mocniny redukovat na problém výpočtu mocniny s *polovičním exponentem*. Intuitivně lze asi vycítit, že to je výrazný posun oproti pouhému zmenšení exponentu o jedna, jak tomu bylo doposud. Hodnotu x^n (pro n nezáporné celé číslo) bychom tedy mohli definovat takto:

$$x^n = \begin{cases} 1 & \text{pokud } n = 0, \\ (x^{\frac{n}{2}})^2 & \text{pokud je } n \text{ sudé,} \\ x \cdot x^{n-1} & \text{pokud je } n \text{ liché.} \end{cases}$$

Výše uvedený předpis povede na rekurzivní proceduru s jednou limitní podmínkou a se dvěma separátními předpisy rekurze – jeden pro případ, kdy je exponent sudý a jeden pro případ, kdy je exponent lichý. Před uvedením samotné rekurzivní procedury si však dokažme, že naše úprava algoritmu je korektní a vede ke správným výsledkům. Na základě původní rekurzivní definice x^n můžeme indukcí prokázat následující tvrzení.

Věta 8.12. *Pro libovolné $x \in \mathbb{R}$ a nezáporná celá čísla m, n platí, že $x^{m+n} = x^m \cdot x^n$.*

Důkaz. Principem indukce prokážeme platnost vlastnosti $P(i)$: „Pro každá dvě nezáporná celá čísla m, n taková, že $m + n = i$, platí $x^{m+n} = x^m \cdot x^n$“. Pro $i = 0$ máme pouze $m = 0$ a $n = 0$. Navíc zřejmě v tomto případě $x^{m+n} = x^{0+0} = x^0 = 1 = 1 \cdot 1 = x^0 \cdot x^0 = x^m \cdot x^n$, takže bod (i) platí. Nyní prokážeme platnost bodu (ii). Předpokládejme, že P platí pro i. Máme ukázat, že P platí pro $i + 1$, to jest máme ukázat, že pro každá dvě nezáporná celá čísla m, n taková, že $m+n = i+1$, platí $x^{m+n} = x^m \cdot x^n$. Z rekurzivní definice x^n dostáváme, že $x^{m+n} = x \cdot x^{m+n-1}$. Jelikož $m + n = i + 1$, pak musí být aspoň jedno z čísel m a n přirozené. Předpokládejme, že je to n (případ pro m by se odvodil duálně). Tím pádem $n - 1$ je celé nezáporné číslo. Máme tedy dvě čísla, m a $n - 1$, která jsou celá a nezáporná a platí pro ně $m + n - 1 = i$. Nyní můžeme aplikovat indukční předpoklad na m a $n - 1$, to jest dostáváme $x^{m+n-1} = x^m \cdot x^{n-1}$. S pomocí poslední rovnosti tedy vyjádříme x^{m+n} takto:

$$x^{m+n} = x \cdot x^{m+n-1} = x \cdot x^m \cdot x^{n-1} = x^m \cdot x \cdot x^{n-1} = x^m \cdot x^n.$$

Poslední rovnost opět plyne z rekurzivní definice x^n. Dohromady jsme tedy prokázali platnost bodu (ii) pro vlastnost P. Užitím principu indukce z věty 8.2 jsme prokázali větu 8.12. □

Nyní je zřejmé, že i upravená rekurzivní definice x^n je korektní, protože dvě věty 8.12 máme

$$(x^{\frac{n}{2}})^2 = x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} = x^{\frac{n}{2}+\frac{n}{2}} = x^n$$

a mohli bychom použít tento fakt při důkazu indukcí. Při prokázání bychom však museli u bodu (ii) věty 8.2 rozlišit víc případů (pro n sudé a n liché). Důkaz ponecháme na laskavém čtenáři. Procedura, která počítá hodnotu x^n podle výše uvedeného upraveného vztahu je zobrazena v programu 8.2. Jelikož

Program 8.2. Rychlá rekurzivní procedura počítající x^n.

```
(define na2
  (lambda (x) (* x x)))

(define expt
  (lambda (x n)
    (cond ((= n 0) 1)
          ((even? n) (na2 (expt x (/ n 2))))
          (else (* x (expt x (- n 1)))))))
```

jsme v programu potřebovali odlišit dva různé předpisy rekurze, vyplatilo se nám použít speciální formu `cond` místo dvou vnořených speciálním forem `if`. Jak jsme na tom se složitostí nové verze `expt`? Nejprve si uveďme příklad její aplikaci s argumenty `2` a `25`. Aplikace je schématicky zachycena na obrázku 8.4. Z příkladu je vidět, že vypočtení hodnoty mocniny pro exponent rovný `25` bylo potřeba provést pouze osm rekurzivních aplikací procedury `expt`. Při použití původní verze by jich bylo potřeba rovných dvacet šest.

Obrázek 8.4. Schématické zachycení aplikace rychlé procedury `expt`.

`(expt 2 25)`	vyvolání 1. aplikace procedury
`(* 2 (expt 2 24))`	*navíjení*: vyvolání 2. aplikace
`(* 2 (na2 (expt 2 12)))`	*navíjení*: vyvolání 3. aplikace
`(* 2 (na2 (na2 (expt 2 6))))`	*navíjení*: vyvolání 4. aplikace
`(* 2 (na2 (na2 (na2 (expt 2 3)))))`	*navíjení*: vyvolání 5. aplikace
`(* 2 (na2 (na2 (na2 (* 2 (expt 2 2))))))`	*navíjení*: vyvolání 6. aplikace
`(* 2 (na2 (na2 (na2 (* 2 (na2 (expt 2 1)))))))`	*navíjení*: vyvolání 7. aplikace
`(* 2 (na2 (na2 (na2 (* 2 (na2 (* 2 (expt 2 0))))))))`	*navíjení*: vyvolání 8. aplikace
`(* 2 (na2 (na2 (na2 (* 2 (na2 (* 2 1)))))))`	*dosažení limitní podmínky*
`(* 2 (na2 (na2 (na2 (* 2 (na2 2))))))`	stav po *odvinutí* 7. aplikace
`(* 2 (na2 (na2 (na2 (* 2 4)))))`	stav po *odvinutí* 6. aplikace
`(* 2 (na2 (na2 (na2 8))))`	stav po *odvinutí* 5. aplikace
`(* 2 (na2 (na2 64)))`	stav po *odvinutí* 4. aplikace
`(* 2 (na2 4096))`	stav po *odvinutí* 3. aplikace
`(* 2 16777216)`	stav po *odvinutí* 2. aplikace
`33554432`	*výsledná hodnota* vzniklá *odvinutím* 1. aplikace

Jak můžeme stanovit složitost nové verze procedury `expt`? Zamysleme se nyní nad ideálním případem, kdy při výpočtu x^n je použit pouze rekurzivní předpis pracující se sudým exponentem až na jediný případ a to pro $n = 1$. V tomto případě je při první, druhé, třetí, ... aplikaci vypočtena hodnota x^0, x^1, x^2, x^4, x^8, x^{16}, x^{32}, ... Obecně řečeno, při k-té aplikaci ($k \geq 2$) je vypočtena hodnota $x^{2^{k-2}}$. Odpověď na otázku, při kolikáté aplikaci je vypočtena hodnota x^n tedy vede na řešení rovnice:

$$x^n = x^{2^{k-2}},$$

jelikož je v našem případě x parametr (zde se dopouštíme mírného zjednodušení), stačí vyřešit

$$n = 2^{k-2},$$

což je po zlogaritmování rovno

$$\log n = (k-2) \cdot \log 2,$$

z čehož vyjádříme k následovně:

$$k = 2 + \frac{\log n}{\log 2} = 2 + \log_2 n.$$

Počet kroků po výpočet x^n je tedy v případě jediného použití rekurzivního předpisu pracujícího s lichým exponentem $2 + \log_2 n$. Jak se počet kroků promítne v případě, kdy dojde k užití více jak jednoho rekurzivního předpisu pro lichý exponent? Uvědomme si nejprve, kolik nejvýš může těchto „patologických situací“ nastat. Při aplikaci rekurzivního předpisu pro lichý exponent je výpočet x^n redukován na výpočet x^{n-1}. V tom případě je ale $n-1$ sudé číslo, takže nikdy nemohou nastat dvě aplikace tohoto rekurzivního předpisu po sobě. To znamená, že v nejhorším případě bude počet kroků nutných k vypočtení x^n dvojnásobkem našeho odhadu $2 + \log_2 n$ (aplikace obou rekurzivních předpisů pro lichý a sudý exponent se budou vzájemně střídat). Dostáváme tím tedy, že počet prostředí, která vznikají během aplikace nové verze procedury `expt` roste logaritmicky (při základu 2) vzhledem k n. Snadno již můžeme vidět, že časová i prostorová složitost jsou shodně $O(\log_2 n)$, což je výrazně lepší výsledek než původní $O(n)$, který se projeví zvlášť pro velká n.

Při programování „rychlé verze `expt`“ bychom si však měli dát pozor na efektivní formalizaci rekurzivního vztahu. Kdybychom místo kódu v programu 8.2 použili následující kód

```
(define expt
  (lambda (x n)
    (cond ((= n 0) 1)
          ((even? n) (* (expt x (/ n 2)) (expt x (/ n 2))))
```

```
        (else (* x (expt x (- n 1)))))),
```

pak by v rekurzivním předpisu pro sudé n docházelo k výpočtu hodnoty $x^{\frac{n}{2}}$ dvakrát. Zbytečně by tedy docházelo k rekurzivní aplikaci. Ve skutečnosti bychom tedy proceduru nijak neurychlili, ale *drasticky* bychom jí *zpomalili*. V programu 8.2 byla hodnota $x^{\frac{n}{2}}$ počítána pouze jednou a její druhá mocnina byla vypočítána procedurou na2. V předchozím kódu však dochází v předpisu rekurze pro sudou větev ke dvěma rekurzivním aplikacím. To jest s každou aplikací původní rychlé procedury se počet aplikací v nové verzi zdvojnásobí. Mohli bychom tedy provést hrubý odhad počtu aplikací pro výpočet x^n v případě, že je použit rekurzivní předpis pro lichá n pouze jednou, jako $2^{1+\log_2 n}$ (hodnota 1 v exponentu reprezentuje jediné použití rekurzivního předpisu pro n liché, použití pro $n = 0$ jsme pro jednoduchost ignorovali), což je větší hodnota než $2^{\log_2 n} = n$. Procedura provádějící dvě rekurzivní aplikace ve svém těle by tedy ve skutečnosti byla dokonce pomalejší než výchozí procedura expt v programu 8.1 (se zvyšujícím se n by bylo zpomalení stále více cítit, například pro $n = 4096$ je potřeba pro výpočet celkem 12288 kroků – v této hodnotě jsou započítány i všechny aplikace pro $n = 0$).

Předchozí problém bychom bez použití na2 mohli vyřešit třeba tak, že hodnotu $x^{\frac{n}{2}}$ vypočteme pouze jednou, navážeme ji v lokálním prostředí na nějaký symbol a poté ji použijeme dvakrát při výpočtu součinu. Technicky se ale vlastně jedná o stejné řešení jako při použití uživatelsky definované procedury na2 (v obou případech vzniká nové prostředí). Upravený kód by tedy vypadal následovně:

```
(define expt
  (lambda (x n)
    (cond ((= n 0) 1)
          ((even? n) (let ((result (expt x (/ n 2))))
                       (* result result)))
          (else (* x (expt x (- n 1)))))))
```

Z hlediska řádové složitosti bychom program výrazně neurychlili, kdybychom přidali další dvě limitní podmínky řešící případy mocnění jedničkou a dvojkou:

```
(define expt
  (lambda (x n)
    (cond ((= n 0) 1)
          ((= n 1) x)
          ((= n 2) (na2 x))
          ((even? n) (na2 (expt x (/ n 2))))
          (else (* x (expt x (- n 1)))))))
```

Předchozí kód by z technického hlediska možná vedl i ke zpomalení výpočtu, protože při každé rekurzivní aplikaci procedury je testováno větší množství podmínek.

Někoho by možná mohlo napadnout udělat další urychlení algoritmu na bází vyřešení případu pro exponent ve tvaru $3n$. Dle věty 8.12 totiž víme, že $x^{3n} = x^{n+n+n} = x^n \cdot x^n \cdot x^n$. Nabízelo by se tedy naprogramovat proceduru na3 počítající třetí mocninu a upravit program 8.2 následovně:

```
(define expt
  (lambda (x n)
    (cond ((= n 0) 1)
          ((= n 1) x)
          ((= n 2) (na2 x))
          ((= (modulo n 3) 0) (na3 (expt x (/ n 3))))
          ((even? n) (na2 (expt x (/ n 2))))
          (else (* x (expt x (- n 1)))))))
```

Přidaná větev říká, že pokud je exponent dělitelný třemi beze zbytku, pak redukujeme problém výpočtu hodnoty x^n na problém výpočtu hodnoty $(x^{\frac{n}{3}})^3$. Analogicky bychom mohli postupovat pro další přirozená čísla $4, 5, \ldots$ Z hlediska efektivity jak teoretické tak praktické to však již *téměř nic nepřináší.* V odborných kruzích panuje názor, že „vylepšování“ programů tímto způsobem „nestojí za to.“ Uveďme si důvod proč. Předně, po zařazení nové větve do programu by počet kroků nutných k výpočtu x^n nerostl logaritmicky při základu dvě, ale logaritmicky *při základu tři* (toto je hodně *optimistický odhad*, což nám ale nyní nevadí, protože chceme prokázat, že složitost se výrazně nezlepší). Co to ale znamená? Z vlastností logaritmů víme:

$$\log_3 n = \frac{\log_2 n}{\log_2 3} = \frac{1}{\log_2 3} \cdot \log_2 n \approx 0.6309 \log_2 n.$$

Což jinými slovy znamená, že z hlediska asymptotické složitosti platí $O(\log_3 n) = O(\log_2 n)$, protože $\log_3 n$ se od $\log_2 n$ liší pouze o multiplikativní konstantu (přibližně) 0.6309. Z tohoto důvodu běžně značíme třídu složitosti bez uvedení základu logaritmu $O(\log n)$, protože tato třída je ekvivalentní všem $O(\log_k n)$, kde $k \in (1, \infty)$. Řádově má předchozí vylepšení programu stejnou složitost jako původní verze v programu 8.2. Ani z čistě praktického hlediska není vhodné takové rozšíření programu dělat, protože čas, který je ušetřený přidanou větví je z velké části zkonzumován testováním většího počtu složitějších podmínek.

Na závěr této sekce si uveďme procedury `fac` a `fib` pro výpočet faktoriálu a Fibonacciho čísel naprogramované pomocí rekurzivních vztahů uvedených v sekci 8.1 na začátku této lekce. Procedury jsou napsány v programech 8.3 a 8.4. Na první pohled je vidět, že rekurzivní procedury `fac` a `fib` jsou výrazně čitelnější

Program 8.3. Rekurzivní výpočet faktoriálu.

```
(define fac
  (lambda (n)
    (if (<= n 1)
        1
        (* n (fac (- n 1))))))
```

Program 8.4. Rekurzivní výpočet prvků Fibonacciho posloupnosti.

```
(define fib
  (lambda (n)
    (if (<= n 1)
        n
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

než procedury uvedené v programech 7.12 a 7.13 vytvořených pomocí `foldr`. Z hlediska výpočtové složitosti by někoho z nás možná mohlo překvapit, že zatímco procedura `fib` z programu 7.13 na straně 187 vrací výsledky „okamžitě“ i pro větší vstupní hodnoty (třeba pro $n = 1000$), rekurzivní procedura 8.4 při hodnotě $n = 1000$ „nemá odezvu“. Tento jev bychom si nyní neměli ukvapeně vykládat, jako že rekurzivní procedury „nejsou efektivní“ (velmi efektivní proceduru jsme konec konců viděli už v programu 8.2 na straně 203), spíš bychom si jej měli vyložit jako důležitou motivaci pro studium vlastností výpočetních procesů generovaných rekurzivními procedurami. Touto problematikou se budeme věnovat v další sekci.

8.3 Výpočetní procesy generované rekurzivními procedurami

V této sekci se budeme zabývat výpočetními procesy generovanými rekurzivními procedurami. Z kvalitativního hlediska mohou totiž rekurzivní procedury generovat výpočetní procesy s různou výpočetní náročností. Vše si budeme postupně demonstrovat na příkladech. Začneme příklady s rekurzivními verzemi procedur `fac` a `fib`, které jsme ukázali v programech 8.3 a 8.4.

Uvažujme nyní rekurzivní verzi procedury `fac` na výpočet faktoriálu. Pokud tuto proceduru aplikujeme s argumentem `5`, pak bude mít výpočetní proces generovaný touto rekurzivní procedurou průběh, který je zakreslený v obrázku 8.5. Výpočetní proces probíhá analogicky jako u původní verze procedury `expt`,

Obrázek 8.5. Schématické zachycení aplikace rekurzivní verze `fac`.

`(fac 5)`	*navíjení*: vyvolání 1. aplikace
`(* 5 (fac 4))`	*navíjení*: vyvolání 2. aplikace
`(* 5 (* 4 (fac 3)))`	*navíjení*: vyvolání 3. aplikace
`(* 5 (* 4 (* 3 (fac 2))))`	*navíjení*: vyvolání 4. aplikace
`(* 5 (* 4 (* 3 (* 2 (fac 1)))))`	*navíjení*: vyvolání 5. aplikace
`(* 5 (* 4 (* 3 (* 2 1))))`	*dosažení limitní podmínky*
`(* 5 (* 4 (* 3 2)))`	stav po *odvinutí* 4. aplikace
`(* 5 (* 4 6))`	stav po *odvinutí* 3. aplikace
`(* 5 24)`	stav po *odvinutí* 2. aplikace
`120`	*výsledná hodnota* vzniklá *odvinutím* 1. aplikace

kterou jsme představili v programu 8.1, srovnejte s obrázkem 8.2. Rekurzivní procedura pro výpočet faktoriálu má zřejmě lineární časovou složitost, protože $n!$ je vypočítána v n krocích. Prostorová složitost je rovněž lineární, protože s každou novou aplikací této procedury vznikne nové prostředí. Počet vzniklých prostředí během aplikace procedury je tedy lineárně závislý na n a žádné další prostorové nároky procedura nemá. Časová i prostorová složitost výpočetního procesu generovaného rekurzivní verzí `fac` jsou tedy ve třídě $O(n)$.

Předchozí řešení výpočtu faktoriálu je založeno na vyjádření $n!$ pomocí $n \cdot (n-1)!$. Hodnotu faktoriálu bychom ale mohli vyjádřit i jiným rekurzivním předpisem. Konkrétně můžeme rekurzivně zavést zobrazení $f\colon \mathbf{N}_0 \times \mathbf{N} \to \mathbf{N}$ tak, že $f(n,k)$ bude mít hodnotu faktoriálu n násobenou hodnotou k. Na první pohled se toto zobrazení může jevit jako „nějaká komplikace“ výchozí definice faktoriálu, ale jak dál uvidíme, pro výpočet faktoriálu bude mnohem efektivnější. Uvažujme tedy předpis definující toto zobrazení:

$$f(n,k) = \begin{cases} k & \text{pokud } n \leq 1, \\ f(n-1, k\cdot n) & \text{jinak.} \end{cases}$$

Předpis je opět rekurzivní a opět bychom o něm mohli prokázat, že je jednoznačný (bude plynout z následujícího tvrzení, takže to nyní prokazovat nebudeme). Na rozdíl od všech ostatních předpisů je rekurzivní předpis f zajímavý tím, že na druhém řádku je hodnota $f(n,k)$ vyjádřena pouze pomocí funkční hodnoty $f(\cdots)$ (bez provádění dalších operací, které by byly zapsány „vně výrazu $f(\cdots)$“). To výrazně zjednodušuje možnost funkční hodnotu takto definovaného zobrazení počítat. Uveďme si nyní několik funkčních hodnot $f(n,1)$ pro $n = 0,1,2,\ldots$:

$$\begin{aligned}
&f(0,1) = 1,\\
&f(1,1) = 1,\\
&f(2,1) = f(1,2) = 2,\\
&f(3,1) = f(2,3) = f(1,6) = 6,\\
&f(4,1) = f(3,4) = f(2,12) = f(1,24) = 24,\\
&f(5,1) = f(4,5) = f(3,20) = f(2,60) = f(1,120) = 120,\\
&\qquad\vdots
\end{aligned}$$

Z předchozích příkladů by měla být jasná intuice skrytá za definicí f. První argument slouží jako jakýsi „čítač“, který jde postupně směrem dolů až dojde k jedničce. Přitom se v druhém argumentu postupně „akumulují“ hodnoty násobků všech hodnot čítače. Tedy násobků $k \cdot n \cdot (n-1) \cdot (n-2) \cdots 2 = n!$, akumulace přitom začíná hodnotou k. V předchozích ukázkách tedy $k = 1$. Toto neformální tvrzení nyní zpřesníme následující větou.

Věta 8.13. *Pro každé $n \in \mathbb{N}_0$, $k \in \mathbb{N}$, $c \in \mathbb{R}$ a $f\colon \mathbf{N}_0 \times \mathbf{N} \to \mathbf{N}$ zavedené předchozím rekurzivním předpisem platí:*

(i) $c \cdot f(n,k) = f(n, c\cdot k)$,

(ii) $f(n,k) = k \cdot n!$,

(iii) $f(n,1) = n!$.

Důkaz. (i): Tvrzení prokážeme indukcí. Z pohledu věty 8.2 nám vlastně jde o prokázání vlastnosti $P(n)$: „pro každé $k \in \mathbb{N}$ a $c \in \mathbb{R}$ platí $c \cdot f(n,k) = f(n, c \cdot k)$". Pro $n \leq 1$ máme dle definici $c \cdot f(n,k) = c \cdot k = f(n, c \cdot k)$, tedy tvrzení platí. Předpokládejme, že n má vlastnost P. Prokážeme, že i $n+1$ má tuto vlastnost. Dle definičního vztahu máme $c \cdot f(n+1,k) = c \cdot f((n+1)-1, k \cdot (n+1)) = c \cdot f(n, k \cdot (n+1))$. S užitím indukčního předpokladu a asociativity násobení dostáváme

$$c \cdot f(n+1,k) = c \cdot f(n, k \cdot (n+1)) = f(n, c \cdot (k \cdot (n+1))) = f(n, (c \cdot k) \cdot (n+1)) = f(n+1, c \cdot k),$$

to jest vlastnost P platí pro každé n. Tím jsme prokázali bod (i).

(ii): Tvrzení prokážeme indukcí a s využitím předchozího bodu. Z pohledu věty 8.2 jde teď o prokázání vlastnosti $P(n)$: „pro každé $k \in \mathbb{N}$ platí $f(n,k) = k \cdot n!$". Pro $n \leq 1$ tvrzení evidentně platí pro každé k. Nechť tvrzení platí pro n. Prokážeme jeho platnost pro $n+1$. S využitím indukčního předpokladu, rekurzivního vztahu pro $n!$ a bodu (i) tedy máme

$$k \cdot (n+1)! = k \cdot ((n+1) \cdot n!) = (k \cdot (n+1)) \cdot n! = f(n, k \cdot (n+1)) = f(n+1, k).$$

To jest P platí pro každé n, tím je prokázán bod (ii).

(iii) je speciálním případem (ii) pro $k = 1$. □

Rekurzivní přepis pro f nám umožňuje naprogramovat novou verzi procedury `fak`. Tato procedura je napsána v programu 8.5. Přísně vzato, s rekurzivně definovaným zobrazením f koresponduje pouze

Program 8.5. Iterativní procedura pro výpočet faktoriálu.

```
(define fac-iter
  (lambda (i accum)
    (if (<= i 1)
        accum
        (fac-iter (- i 1) (* accum i)))))

(define fac
  (lambda (n)
    (fac-iter n 1)))
```

pomocná procedura `fac-iter`. Procedura `fac` pouze provede aplikaci `fac-iter` s druhým argumentem nastaveným na `1`. Proceduru `fac` jsme zavedli proto, že její uživatelé by se k výpočtu faktoriálu měli stavět jako k černé skříňce. I když pro jeho výpočet potřebujeme definovat proceduru dvou argumentů, tento fakt by pro programátory používající pouze `fac` měl být skryt, což se nám tímto rozdělením na dvě procedury (`fac` a pomocnou `fac-iter`) podařilo. V obrázku 8.6 máme zachycen průběh výpočtu nové verze `fac` pro hodnotu `5`. Na tomto příkladu si můžeme všimnout několika zajímavých věcí. Nejprve konstatujme, že

Obrázek 8.6. Schématické zachycení aplikace iterativní verze `fac`.

`(fac 5)`	
`(fac-iter 5 1)`	*navíjení*: vyvolání 1. aplikace
`(fac-iter 4 5)`	*navíjení*: vyvolání 2. aplikace
`(fac-iter 3 20)`	*navíjení*: vyvolání 3. aplikace
`(fac-iter 2 60)`	*navíjení*: vyvolání 4. aplikace
`(fac-iter 1 120)`	*navíjení*: vyvolání 5. aplikace
`120`	*dosažení limitní podmínky a vrácení hodnoty*

je zřejmě vidět, že časovou složitost výpočtu jsme oproti předcházející verzi nijak nezlepšili, pro výpočet faktoriálu je i nadále potřeba provést n součinů, což vede na časovou složitost v třídě $O(n)$. V příkladu je ale zajímavé, že fáze *odvíjení* prakticky *chybí*. Lépe řečeno, *fáze odvíjení je degenerovaná* na pouhé *postupné vracení téže hodnoty* (*výsledku*). To je důsledkem faktu, že rekurzivní předpis v proceduře `fac-iter` obsahuje *pouze* sebe sama s novými hodnotami. Tato aplikace není použita jako součást žádného odloženého výpočtu. Při dosažení limitní podmínky je tedy pouze postupně vracena výsledná hodnota, se kterou se již nijak nemanipuluje.

Otázkou je, zda-li bychom předchozí pozorování o trivialitě fáze odvíjení dokázali využít při efektivnější aplikaci procedury. Víme již, že časovou stránku jsme nevylepšili. Otázkou je, jestli bychom mohli vylepšit prostorovou náročnost výpočetního procesu. I v programu 8.6 totiž dochází pro vstupní hodnotu n k n aplikacím `fac-iter`. Podle principu aplikace uživatelsky definovaných procedur by tedy mělo vzniknout n nových prostředí a prostorová složitost výpočetního procesu by byla opět $O(n)$. Připomeňme, že v rekurzivní verzi `fac` sloužila prostředí k uchování informací o odloženém výpočtu. Jelikož `fac-iter` odložený výpočet nevytváří, nabízí se možnost provádět její aplikace neustále v *jednom prostředí*, pouze *s měnícími se vazbami symbolů*. Tím bychom prostorovou složitost výpočetního procesu srazili dolů na $O(1)$ (procedura by pro libovolně velké vstupní n pracovala v konstantním prostoru – což je z programátorského hlediska „ideální složitost“). Optimalizaci aplikací některých rekurzivních procedur v tomto smyslu (nevytváření nových prostředí při každé aplikaci, ale pouze přepisování vazeb a vyhodnocování v jednom prostředí) nazýváme *optimalizace na koncovou rekurzi* (anglicky *tail recursion optimization*, často zkracováno TRO).

Interprety jazyka Scheme dělají optimalizaci na koncovou rekurzi automaticky ve všech situacích, kdy je to možné. Abychom si tuto optimalizaci blíže popsali, zavedeme si několik nových pojmů. Jedním z hlavních pojmů je tak zvaná *koncová pozice* (speciální pozice výrazu v těle procedury), ze které lze provést rekurzivní aplikaci procedury bez nutnosti vytvářet nové prostředí.

Definice 8.14 (koncová pozice, koncová aplikace, koncově rekurzivní procedura)**.** Množina *koncových pozic λ-výrazu* Λ je definována následovně:

(i) poslední výraz v těle výrazu Λ je v koncové pozici výrazu Λ;

(ii) je-li `(if` $\langle$*test*$\rangle$ $\langle$*důsledek*$\rangle$ $\langle$*náhradník*$\rangle$`)` v koncové pozici výrazu Λ,
pak $\langle$*důsledek*$\rangle$ i $\langle$*náhradník*$\rangle$ (pokud je přítomen) jsou v koncové pozici výrazu Λ;

(iii) je-li

```
(cond (⟨test1⟩ ⟨důsledek1⟩)
      (⟨test2⟩ ⟨důsledek2⟩)
        ⋮
      (⟨testn⟩ ⟨důsledekn⟩)
      (else ⟨náhradník⟩))
```

v koncové pozici výrazu Λ, pak $\langle$*důsledek*$_1\rangle, \ldots, \langle$*důsledek*$_n\rangle$ jsou v koncové pozici výrazy Λ a pokud je přítomna i `else`-větev, pak je i $\langle$*náhradník*$\rangle$ v koncové pozici Λ;

(iv) je-li `(and` $\cdots$ $\langle$*výraz*$\rangle$`)` v koncové pozici výrazu Λ, pak je $\langle$*výraz*$\rangle$ v koncové pozici výrazu Λ;
totéž platí pro speciální formu `or`;

(v) je-li `(let (`$\cdots$`)` $\cdots$ $\langle$*výraz*$\rangle$`)` v koncové pozici výrazu Λ, pak je $\langle$*výraz*$\rangle$ v koncové pozici výrazu Λ;
totéž platí pro všechny ostatní varianty speciální formy `let`.

Koncová aplikace procedury vzniklé vyhodnocením λ-výrazu Λ je aplikace vyvolaná z *koncové pozice výrazu* Λ. Rekurzivní procedura se nazývá *koncově rekurzivní*, pokud aplikuje sebe sama pouze z *koncových pozic* (λ-výrazu jehož vyhodnocením vznikla). ■

Ve standardu jazyka Scheme R^5RS a také v jeho IEEE standardu se koncové pozice definují ještě o něco složitějším způsobem (částečně proto, že jsme ještě nepředstavili úplně všechny speciální formy jazyka Scheme). My si ale prozatím vystačíme s předchozím chápáním.

Příklad 8.15. Vraťme se nyní k proceduře `fac-iter` z programy 8.5. Označme Λ výraz, jehož vyhodnocením vzniká procedura, která je potom navázána na symbol `fac-iter`. Jelikož je v těle výrazu Λ pouze jediný výraz, je v koncové pozici. Tento výraz je navíc ve tvaru `(if`$\cdots$, podle (ii) předchozí definice tedy dostáváme, že i výrazy `accum` a `(fac-iter (- i 1) (* accum i))` jsou v koncových pozicích Λ.

Aplikace iniciovaná vyhodnocením výrazu `(fac-iter (- i 1) (* accum i))` je tedy koncová aplikace procedury `fac-iter`, protože byla vyvolána z koncové pozice. Jelikož v těle procedury `fac-iter` již na jiném místě aplikace `fac-iter` není vyvolána, procedura `fac-iter` je tedy koncově rekurzivní. Kdybychom si vzali například proceduru `fac` z programu 8.3 na straně 206, tak tato procedura aplikuje sebe sama také na jediném místě, ale toto místo není koncová pozice. V koncové pozici se totiž nachází celý výraz `(* n (fac (- n 1)))`, nikoliv pouze `(fac (- n 1))`. Procedura `fac` z programu 8.3 tedy není koncově rekurzivní.

Koncově rekurzivní procedury lze ve skutečnosti rozpoznat velmi jednoduše. Rekurzivní procedura je *koncově rekurzivní*, pokud výsledkem každé její aplikace pouze nová aplikace sebe sama, která není součástí žádného složeného výrazu (až na konstrukce `if`, `cond` a podobně, viz definici 8.14). Tím, že koncové aplikace nejsou již v daném prostředí použity k žádnému výpočtu, může se pro provedení aplikace použít právě aktuální prostředí a není potřeba vytvářet prostředí nové.

Aplikaci uživatelsky definovaných procedur můžeme nyní rozšířit o speciální případ aplikace procedury z její koncové pozice. Postup již nebudeme popisovat formálně tak, jak jsme to udělali třeba v definici 2.12 na straně 50, ale vysvětlíme jej pouze slovně. Pokud je při aplikaci procedury (to jest během vyhodnocení jejího těla) zaznamenán pokus o opětovnou aplikaci z koncové pozice, pak není vytvářeno nové prostředí, pouze se *předefinují vazby symbolů v aktuálním prostředí* tak, aby odpovídaly hodnotám při nové aplikaci a *v aktuálním prostředí* (s upravenými vazbami symbolů) *je opět vyhodnoceno tělo procedury.*

Jelikož je procedura `fac-iter` z programu 8.5 koncově rekurzivní, pak se při výpočtu faktoriálu (pro libovolně velké n) vytvoří pouze dvě prostředí – jedno při aplikaci `fac` z programu 8.5 a druhé při následné aplikaci `fac-iter`. V tomto druhém prostředí pak již ale probíhá vyhodnocení všech aplikací `fac-iter`. Jelikož se v těle `fac-iter` provádějí operace s konstantní prostorovou složitostí, pak je zřejmé, že prostorová složitost tohoto výpočetního procesu je $O(1)$.

Ačkoliv procedury `fac` z programů 8.3 a 8.5 můžeme chápat jako reprezentace stejného zobrazení, je mezi nimi z hlediska výpočetních procesů, které generují, výrazný rozdíl. Tento rozdíl se promítá především to prostorové složitosti obou procedur. Na dalších příkladech uvidíme řadu výpočetních procesů, které se chovají jako výpočetní procesy v případě procedur z obou programů. Proto se vedle *rekurzivních procedur* budeme také bavit *rekurzivních výpočetních procesech,* které tyto procedury generují a budeme rozlišovat jejich různé typy.

Obecně řečeno, výpočetní proces budeme nazývat *rekurzivní výpočetní proces,* pokud se bude jednat o proces *generovaný rekurzivní procedurou* nebo *několika procedurami, které se vzájemně aplikují,* u kterého lze rozlišit fáze *navíjení* a *odvíjení* (přitom fáze odvíjení může být triviální).

Příklad 8.16. V předchozí formulaci rekurzivního výpočetního procesu je možná nejasné, co je myšleno „několika vzájemně se aplikujícími procedurami“. Například pokud budeme uvažovat následující dvě procedury

```
(define fac-a (lambda (n) (if (<= n 1) 1 (* n (fac-b (- n 1))))))
(define fac-b (lambda (n) (if (<= n 1) 1 (* n (fac-a (- n 1)))))),
```

pak snadno vidíme, že obě jsou modifikace rekurzivní verze procedury pro výpočet faktoriálu. Ovšem, v těle procedury navázané na `fac-a` je aplikována procedura navázaná na `fac-b` a obráceně. Přísně vzato tedy ani `fac-a` ani `fac-b` *nejsou rekurzivní procedury.* Na druhou stranu je ale zřejmé, že při aplikaci libovolné z nich (pro `n` větší než jedna) bude výpočetní proces pokračovat aplikací druhé procedury, poté opět aplikací první procedury až do bodu, kdy bude dosažena limitní podmínka. Výpočetní proces tedy *má* fázi navíjení. Stejně tak má fázi odvíjení, která začíná splněním limitní podmínky. V tomto případě tedy máme dvě procedury, které se navzájem aplikují a generují rekurzivní výpočetní proces i když nejsou rekurzivní. Všimněte si, že obě procedury počítají faktoriál.

V této sekci jsme zatím poznali dva typy rekurzivních výpočetních procesů:

lineární rekurzivní proces je rekurzivní proces, který má netriviální fáze navíjení a odvíjení. Během fáze navíjení je budována „série odložených výpočtů“, přitom počet prostředí roste lineárně vzhledem k velikosti vstupních argumentů. Po dosažení limitní podmínky nastává fáze odvíjení, ve které je zpětně dokončeno vyhodnocování výrazů, které bylo započato a „odloženo“ ve fází navíjení. Činnost lineárně rekurzivního výpočetního proces nelze jednoduše přerušit ani není možné „skočit“ na nějaké místo rekurze[17]. Procedura `fac` z programu 8.3 generuje právě lineárně rekurzivní výpočetní proces.

lineární iterativní proces je výpočetní proces generovaný koncově rekurzivními procedurami. Nedochází při něm k vytváření „odložených výpočtů.“ Každý lineárně iterativní výpočetní proces je během své činnosti jednoznačně určen

(i) vazbami symbolů v prostředí $\mathcal{P}$ svého běhu,
(ii) předpisem, jak změnit stav vazeb v $\mathcal{P}$ na základě aktuálních vazeb,
(iii) limitní podmínkou ukončující iterativní proces.

Lineárně iterativní výpočetní proces může být během svého výpočtu přerušen a posléze opět obnoven v místě přerušení. Fáze průběhu lineárně iterativní procesu je dána vazbami symbolů v prostředí jeho běhu, viz bod (i) z předchozího výpisu. Pokud tyto vazby uchováme a lineárně iterativní proces opustíme, je možné jej opět aktivovat s počátečními hodnotami nastavenými na uschované hodnoty. Tím pádem se iterativní proces „rozběhne“ od místa zastavení.

Pro někoho možná nyní nastal malý terminologický zmatek, protože se bavíme o *rekurzivních procedurách* a o *rekurzivních výpočetních procesech.* Navíc některé *rekurzivní procedury* generují *iterativní výpočetní procesy.* Ve většině programovacích jazyků (zejména procedurálních jazyků) rekurzivní procedury nemohou generovat iterativní procesy. Na vytváření iterativních procesů jsou v těchto jazycích speciální prostředky – nejčastěji *cykly.* Z pohledu jazyka Scheme je tedy iterativní výpočetní proces ekvivalentem cyklů známých z procedurálních jazyků.

Příklad 8.17. Přerušení iterativního procesu si lze představit následovně. Vezměme si třeba proces schématicky ukázaný na obrázku 8.6. Kdybychom na 4. řádku proces „uřízli“ a zapamatovali si hodnoty `3` a `20`, což byly aktuální vazby symbolů v prostředí, ve kterém byla aplikována procedura `fac-iter`, pak bychom mohli kdykoliv iterativní proces „spustit“ počínaje tímto krokem prostě tak, že bychom `fac-iter` aplikovali s danými hodnotami. Výsledkem by byla požadovaná hodnota `120`. Naprogramujte tuto proceduru a prakticky se přesvědčte o jejím chování.

Poznámka 8.18. (a) Kvůli zjednodušení terminologie budeme rekurzivním procedurám generujícím pouze iterativní výpočetní procesy říkat *iterativní procedury* a procedurám generujícím pouze lineárně rekurzivní procesy *lineárně rekurzivní procedury.* Pořád je ale potřeba mít na paměti rozdíl mezi pojmy rekurzivní procedura versus rekurzivní výpočetní proces.

(b) Některé argumenty iterativních procedur hrají speciální role. Jedním typem argumentů jsou tak zvané *čítače.* Úkolem čítačů je nějakým způsobem počítat kroky iterace. Podle stavu čítače lze v případě potřeby zastavit iteraci. Například v proceduře `fac-iter` z programu 8.5 představuje formální argument `i` čítač, který je na počátku iterace nastaven na hodnotu n a v každém kroku iterace je snížen o jedna. Druhým typem významných argumentů jsou *střadače.* Účelem střadačů je nějakým způsobem akumulovat hodnoty během iterace. Ve výše uvedené proceduře `fac-iter` je `accum` střadač, který je na počátku iterace nastaven na hodnotu 1 a v každém kroku je akumulátor násoben aktuální hodnotou čítače – tímto způsobem se postupně akumulují součiny, které ve výsledku dávají hodnotu faktoriálu.

(c) Některé rekurzivní procedury aplikují sebe sama z několika pozic, některé z nich jsou koncové a některé koncové nejsou. Takové procedury obecně nejsou iterativní, protože při aplikaci z nekoncových pozic je potřeba vytvořit nové prostředí. Na druhou stranu, i u těchto procedur jsou koncové aplikace prováděny v témže prostředí. Takto se chová třeba následující procedura

[17] V dalším díle toho učebního textu uvidíme, že to ve skutečnosti možné je pomocí tak zvaných *únikových funkcí* a *aktuálního pokračování*. Jelikož se však jedná o pokročilé speciální konstrukty, věnujeme se jim až v dalším díle textu. Zatím bychom se na „výskok z rekurze“ měli dívat jako na něco, co je běžnými programátorskými prostředky neřešitelné.

```
(define proc
  (lambda (x)
    (cond ((<= x 0) 0)
          ((even? x) (+ x (proc (/ x 2))))
          (else (proc (- x 1)))))),
```

která koncově aplikuje sebe sama v případě, že na `x` je navázané liché číslo (viz tělo procedury).Pokud je na `x` navázané sudé číslo různé od nuly, pak je provedena aplikace z nekoncové pozice a při ní je vytvořeno nové prostředí.

Na závěr rozboru rekurzivní a iterativní verze procedur pro výpočet faktoriálu dodejme, že z hlediska programátorské čistoty by bylo vhodné nadefinovat pomocnou proceduru `fac-iter` interně v proceduře `fac`, protože `fac-iter` byla vytvořena za účelem, že ji bude aplikovat pouze `fac`. Iterativní procedura `fac` s interně definovanou pomocnou procedurou jsou uvedeny v programu 8.6. V programu 8.7 máme

Program 8.6. Iterativní procedura pro výpočet faktoriálu s interní definicí.

```
(define fac
  (lambda (n)
    (define iter
      (lambda (i accum)
        (if (<= i 1)
            accum
            (iter (- i 1) (* accum i)))))
    (iter n 1)))
```

uvedenou iterativní verzi procedury `expt` (procedura má opět pomocnou proceduru `expt-iter`, provádějící samotnou iteraci) pracující s časovou složitostí $O(\log n)$ a prostorovou složitostí $O(1)$. Na proceduře

Program 8.7. Iterativní procedura pro výpočet mocniny.

```
(define expt-iter
  (lambda (x n accum)
    (cond ((= n 0) accum)
          ((even? n) (expt-iter (* x x) (/ n 2) accum))
          (else (expt-iter x (- n 1) (* accum x))))))

(define expt
  (lambda (x n)
    (expt-iter x n 1)))
```

si všimněte toho, že jsme museli mírně upravit rekurzivní předpis tak, aby bylo možné jej vyjádřit pomocí koncové rekurze. Místo faktu $x^{2n} = (x^n)^2$, který jsme využili v programu 8.2 na straně 203, jsme nyní využili vztahu $x^{2n} = (x^2)^n$, to jest během iterativního výpočtu průběžně měníme kromě exponentu také základ. Přesvědčte se sami, že tato úvaha je správná a vede k řešení. Na obrázku 8.7 je pro demonstraci uveden průběh výpočtu 2^{25}. Zde si můžeme povšimnout průběžné změny základu (první argument), exponentu (druhý argument) a pomocného střadače (třetí argument), jehož hodnota je nakonec vrácena jako výsledek.

Přirozenou otázkou je, zda-li lze vždy rekurzivní výpočetní procesy nahradit iterativními výpočetními procesy. V terminologii procedur, které tyto procesy generují, se tedy ptáme, zda-li pro každou

Obrázek 8.7. Schématické zachycení iterativní verze procedury `expt`.

`(expt 2 25)`	
`(expt-iter 2 25 1)`	*navíjení*: vyvolání 1. aplikace
`(expt-iter 2 24 2)`	*navíjení*: vyvolání 2. aplikace
`(expt-iter 4 12 2)`	*navíjení*: vyvolání 3. aplikace
`(expt-iter 16 6 2)`	*navíjení*: vyvolání 4. aplikace
`(expt-iter 256 3 2)`	*navíjení*: vyvolání 5. aplikace
`(expt-iter 256 2 512)`	*navíjení*: vyvolání 6. aplikace
`(expt-iter 65536 1 512)`	*navíjení*: vyvolání 7. aplikace
`(expt-iter 65536 0 33554432)`	*navíjení*: vyvolání 8. aplikace
`33554432`	*dosažení limitní podmínky a vrácení hodnoty*

proceduru vedoucí na rekurzivní proces můžeme napsat proceduru vedoucí na iterativní proces. Odpověď je kladná, ale v některých případech to může být dost těžké a výsledek mnohdy „nestojí za to" – výsledná procedura generující iterativní proces je výrazně méně čitelná než výchozí procedura. Díky čemu je možné vždy najít proceduru vedoucí na iterativní proces? Je to tím, že rekurzivní aplikaci a konstrukci odložených výpočtů můžeme plně simulovat pomocí *zásobníku*, který lze v případě jazyka Scheme udržovat pomocí seznamu odložených hodnot čekajících na další zpracování.

V programu 8.8 je uvedena iterativní verze procedury `expt`, která provádí simulace fáze navíjení a odvíjení původní rekurzivní procedury z programu 8.1 pomocí zásobníku. Procedura `expt` z programu 8.8 má

Program 8.8. Iterativní procedura pro výpočet mocniny s pomocí zásobníku.

```
(define expt
  (lambda (x n)
    (define expt-stack
      (lambda (n accum stack)
        (if (= n 0)
            (cond ((null? stack) accum)
                  ((car stack) (expt-stack 0 (na2 accum) (cdr stack)))
                  (else (expt-stack 0 (* accum x) (cdr stack))))
            (if (even? n)
                (expt-stack (/ n 2) accum (cons #t stack))
                (expt-stack (- n 1) accum (cons #f stack))))))
    (expt-stack n 1 '())))
```

v sobě interně definovanou pomocnou proceduru `expt-stack` tří argumentů. Prvním z nich je exponent, druhým je střadač, který bude na konci výpočtu obsahovat hodnotu mocniny x^n a posledním argumentem je `stack`. Základ (navázaný na symbol `x` v prostředí aplikace `expt`) není potřeba předávat, protože jeho hodnota je dostupná pro interně definovanou proceduru `expt-stack` a hodnota `x` nebude během výpočtu měněna. Procedura `expt-stack`, která provádí samotný výpočet, pracuje tak, že simuluje fázi navíjení a odvíjení na zásobníku. V `if`-výrazu v těle procedury jsou obě fáze rozlišeny limitní podmínkou `(= n 0)`. Pokud limitní podmínky není dosaženo, je zmenšován exponent (buď o jedna nebo na polovinu podle toho jestli je lichý nebo sudý) a na zásobník se pokládají hodnoty `#f` (příznak pro násobení základem v další fázi výpočtu) a `#t` (příznak pro umocnění na druhou v další fází výpočtu). Po dosažení limitní podmínky začne být zpracováván zásobník, což je de facto simulace fáze odvíjení. Pokud je zásobník vyprázdněn, je vrácena hodnota akumulátoru. Pokud je prvním prvkem na zásobníku `#t` (příznak pro provedení umocnění), provede se další iterace s umocněnou hodnotou akumulátoru, v opačném případě je akumulátor vynásoben základem. Průběh výpočtu (pouze procedura `expt-stack`) je zobrazen na obrázku 8.8. Jak je z příkladu

Obrázek 8.8. Schématické zachycení aplikace `expt` vytvořené s využitím zásobníku.

`(expt-stack 25 1 ())`	*simulace navíjení*: vyvolání 1. aplikace
`(expt-stack 24 1 (#f))`	*simulace navíjení*: vyvolání 2. aplikace
`(expt-stack 12 1 (#t #f))`	*simulace navíjení*: vyvolání 3. aplikace
`(expt-stack 6 1 (#t #t #f))`	*simulace navíjení*: vyvolání 4. aplikace
`(expt-stack 3 1 (#t #t #t #f))`	*simulace navíjení*: vyvolání 5. aplikace
`(expt-stack 2 1 (#f #t #t #t #f))`	*simulace navíjení*: vyvolání 6. aplikace
`(expt-stack 1 1 (#t #f #t #t #t #f))`	*simulace navíjení*: vyvolání 7. aplikace
`(expt-stack 0 1 (#f #t #f #t #t #t #f))`	*ukončení simulace navíjení*
`(expt-stack 0 2 (#t #f #t #t #t #f))`	*simulace stavu* po *odvinutí* 8. aplikace
`(expt-stack 0 4 (#f #t #t #t #f))`	*simulace stavu* po *odvinutí* 7. aplikace
`(expt-stack 0 8 (#t #t #t #f))`	*simulace stavu* po *odvinutí* 6. aplikace
`(expt-stack 0 64 (#t #t #f))`	*simulace stavu* po *odvinutí* 5. aplikace
`(expt-stack 0 4096 (#t #f))`	*simulace stavu* po *odvinutí* 4. aplikace
`(expt-stack 0 16777216 (#f))`	*simulace stavu* po *odvinutí* 3. aplikace
`(expt-stack 0 33554432 ())`	*simulace stavu* po *odvinutí* 2. aplikace
`33554432`	*výsledná hodnota*

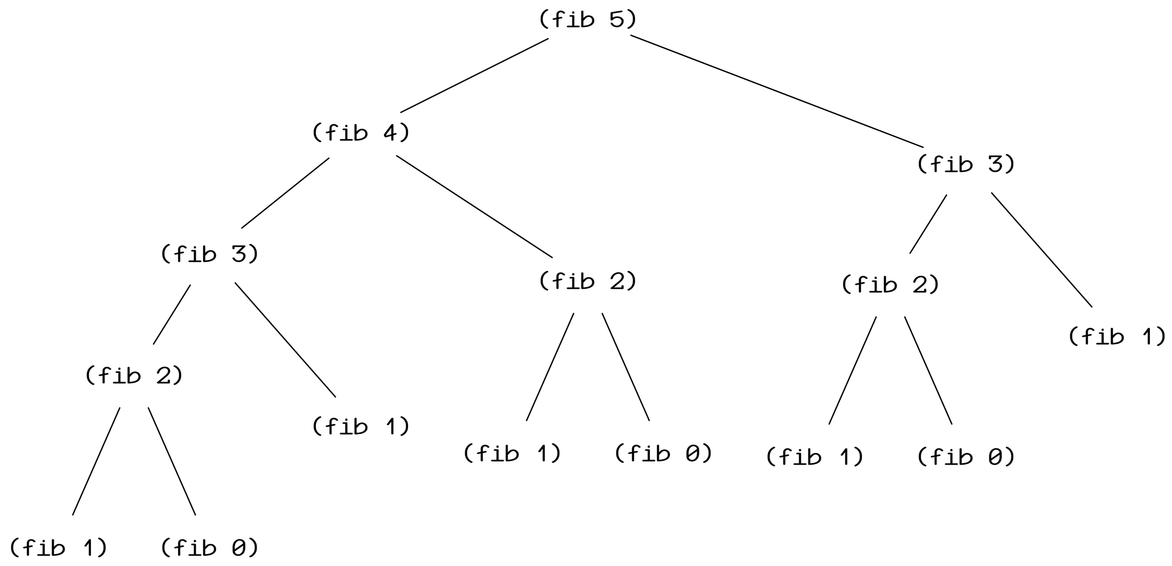
a zřejmě i z programu 8.8 patrné, přepisování rekurzivních procedur pomocí dodatečného zásobníku v mnoha případech nepřispívá k čitelnosti programu, ani výrazně nezlepšuje efektivitu. Iterativní verze procedury `expt` z programu 8.8 má časovou složitost $O(\log n)$, protože počet kroků je řádově stejný jako u efektivní rekurzivní varianty z programu 8.2. Prostorová složitost je také $O(\log n)$, protože v každém kroku aplikace je při simulaci navíjení zvětšen zásobník o jeden prvek. Prostorová složitost je tedy řádově stejná jako u algoritmu 8.2. Iterativní verze `expt` pracující se zásobníkem tedy není z pohledu výpočtové efektivity efektivnější než lineárně rekurzivní varianta. Iterativní proces generovaný procedurou `expt-iter` z programu 8.8 má tedy prostorovou složitost $O(\log n)$ (a nikoliv pouze $O(1)$).

Na programu 8.8 si můžeme uvědomit, že stanovení prostorové složitosti se *neodvíjí pouze od počtu aplikací procedur* (a tím pádem od počtu vzniklých prostředí), ale musíme brát v potaz i *konstrukci hierarchických datových struktur* jejichž délka je nějakým způsobem závislá na velikosti vstupních argumentů.

Nyní se budeme zabývat třetím základní typem rekurzivních výpočetních procesů. Doposud jsme se zabývali procedurami, které ve svých rekurzivních předpisech aplikovaly sebe sama právě jednou. Jedinou výjimkou byla procedura `fib` z programu 8.4 pro výpočet prvků Fibonacciho posloupnosti pomocí jejich rekurzivního předpisu. Tuto proceduru jsme „snadno naprogramovali“, ale již na konci sekce 8.2 jsme konstatovali, že procedura je trestuhodně neefektivní. To, jak moc je neefektivní, rozebereme nyní. Znázorněme si nejdřív schématicky průběh aplikace této procedury. Jelikož její rekurzivní předpis obsahuje dvě aplikace sebe sama, budeme vývoj výpočetního procesu zachycovat *stromem*. Uzly ve stromu budou zastupovat jednotlivé aplikace. Vrcholem stromu bude výchozí aplikace. Každý uzel má ve stromu tolik potomků, kolik je provedeno v rekurzivním předpisu dalších aplikací.

V případě procedury `fib` z programu 8.4 bude situace pro vstupní hodnotu `5` vypadat tak, jak je znázorněno na obrázku 8.9. Strom z obrázku 8.9 reprezentuje průběh výpočtu. Je to ale *pouze jeho schéma*. Tento obrázek bychom si neměli vykládat tak, že při výpočtu je nějaký takový strom skutečně „sestaven“ a pak se v něm něco „hledá“. Samotný průběh aplikace je zachycen na obrázku 8.10. Výpočet zahájený vyhodnocením `(fib 5)` pokračuje další aplikací `(fib 4)` (za předpokladu, že interpret vyhodnocuje prvky seznamu před aplikací procedury zleva doprava), dále se pokračuje aplikací `(fib 3)`, `(fib 2)`, a `(fib 1)`. V tomto bodu a v této fázi výpočtu jsme narazili na limitní podmínku, nastává fáze odvíjení. V dalším kroku se tedy

Obrázek 8.9. Schématické zachycení aplikace rekurzivní verze `fib`.



Obrázek 8.10. Postupné provádění aplikací při použití rekurzivní verze `fib`.

```
(fib 5)
(+ (fib 4) (fib 3))
(+ (+ (fib 3) (fib 2)) (fib 3))
(+ (+ (+ (fib 2) (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ (+ (fib 1) (fib 0)) (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ (+ 1 (fib 0)) (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ (+ 1 0) (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ 1 (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ 1 1) (fib 2)) (fib 3))
(+ (+ 2 (fib 2)) (fib 3))
(+ (+ 2 (+ (fib 1) (fib 0))) (fib 3))
(+ (+ 2 (+ 1 (fib 0))) (fib 3))
(+ (+ 2 (+ 1 0)) (fib 3))
(+ (+ 2 1) (fib 3))
(+ 3 (fib 3))
(+ 3 (+ (fib 2) (fib 1)))
(+ 3 (+ (+ (fib 1) (fib 0)) (fib 1)))
(+ 3 (+ (+ 1 (fib 0)) (fib 1)))
(+ 3 (+ (+ 1 0) (fib 1)))
(+ 3 (+ 1 (fib 1)))
(+ 3 (+ 1 1))
(+ 3 2)
5
```

opět vrátíme do bodu aplikace `(fib 2)`, abychom zde dokončili „odložený výpočet". Součástí odloženého výpočtu je však další rekurzivní aplikace `(fib 0)`. Po jeho provedení se opět narazí na limitní podmínku a výpočet pokračuje v prostředí aplikace `(fib 2)`, kde je dokončen výpočet, ten již skutečně dokončen být může, protože hodnoty `1` a `0` pro součet již byly spočteny předchozími rekurzivními aplikacemi. Nastává tedy dále fáze odvíjení v prostředí aplikace `(fib 3)`. Zde je součástí odloženého výpočtu další rekurzivní aplikaci `(fib 1)`, ... Fáze navíjení a odvíjení se dále střídají ve směru šipky nakreslené v obrázku 8.9. Všimněte si, že neefektivita výpočetního procesu spočívá v opakovaném vypočítávání týchž hodnot.

Nyní můžeme provést hrubý odhad složitosti výpočetního procesu generovaného rekurzivní procedurou `fib` z programu 8.4. Počet kroků, které je pro dané n potřeba k vypočtení výsledku je daný počtem uzlů, které se nacházejí ve stromě zachycujícím všechny aplikace (protože sčítání v jednotlivých aplikacích probíhá v konstantním čase). V našem případě má každý uzel buď žádný nebo dva potomky. Nejdelší cesta od kořene k listu (uzlu bez potomka), tak zvaná *hloubka stromu*, má pro dané n délku n. Pokud bychom si nyní zjednodušili situaci tím, že bychom konstatovali, že strom je vyvážený, pak víme, že počet jeho uzlů je $2^n - 1$. Pesimistický odhad časové složitosti by tedy byl $O(2^n)$. Podotkněme, že strom aplikací `fib` obecně není vyvážený a lze udělat mnohem přesnější odhady časové složitosti výpočty hodnoty `fib` pro n, ale důležitým faktem je, že časová složitost narůstá exponenciálně vzhledem k n. Procedura `fib` z programu 8.4 je tedy použitelná pouze pro malé vstupní hodnoty n (už při $n = 30$ je prodleva znatelná, protože proběhne celkem 2692537 aplikací). Procedura je tedy prakticky nepoužitelná. S prostorovou složitostí na tom nejsme tak špatně, ta je v třídě $O(n)$. Při jejím stanovení je klíčové pozorování, kolik prostoru je spotřebováno (maximálně) při průchodu stromem z obrázku 8.9 ve směru postupných aplikací. Pro dané n má strom největší hloubku n, tedy spotřebovaný prostor roste lineárně s hloubkou stromu (to jest s rostoucím n).

Výpočetní proces generovaný procedurou `fib` z programu 8.4 nazýváme *stromově rekurzivní výpočetní proces.* Charakteristiku procesů toho typu bychom mohli shrnout takto:

stromově rekurzivní proces je výpočetní proces svým průběhem připomínající lineární výpočetní proces, jen s tím rozdílem, že počet aplikací rekurzivní neroste lineárně s velikostí vstupu. Stromově rekurzivní proces je typicky generován procedurami, které ve svých rekurzivních předpisech vyvolají dvě nebo více aplikací sebe sama. Pro stromově rekurzivní výpočetní procesy je charakteristické postupné střídání fází navíjení a odvíjení.

> V případě stromově rekurzivních výpočetních procesů stanovujeme časovou složitost stanovením počtu aplikací procedury v závislosti na vstupních datech, což je *počet uzlů* v pomyslném „stromu" zachycujícím průběh celého výpočtu. Při stanovení prostorové složitosti se lze odrazit od *hloubky stromu,* která udává maximální hloubku (zanoření) rekurzivních aplikací.

Dobrou zprávou je, že i v případě Fibonacciho čísel můžeme naprogramovat proceduru pro jejich výpočet efektivně. Snadno totiž můžeme najít iterativní proceduru, která bude Fibonacciho čísla počítat v čase $O(n)$ a v prostoru $O(1)$. Tato procedura bude mít jeden argumentem jímž bude *čítač* zachycující kolik je potřeba udělat iterací a *dva pomocné argumenty* v nichž bude uchovávat poslední dvě vypočtená Fibonacciho čísla. Procedura je uvedena v programu 8.9. Procedura `fib` opět provede pouze aplikaci pomocné iterativní procedury `fib-iter`. Procedura `fib-iter` je z `fib` aplikována s čítačem nastavením na hodnotu vstupního čísla (poslední argument), první dva prvky Fibonacciho posloupnosti jsou `0` a `1` (první dva argumenty). Z kódu procedury `fib-iter` lze snadno vyčíst, že iterace je zastavena pokud čítač klesne na nulu. Potom je vrácena hodnota navázaná na prvním symbolu (první pomocný argument). Rekurzivní předpis říká, že v každém kroku je snížen čítač o jedna, a záznam o posledních dvou Fibonacciho číslech je změněn tak, aby se v dalším kroku opět jednalo o (další) dvě poslední Fibonacciho čísla v posloupnosti. Na obrázku 8.11 je ukázka aplikace procedury `fib` z programu 8.9 pro vstupní hodnotu `5`. Časová složitost výpočtu je zřejmě ve třídě $O(n)$, prostorová v $O(1)$. Procedur `fib-iter` v programu 8.9 bychom opět mohli definovat jako interní ve `fib`, abychom provedli odstínění pomocné procedury `fib-iter` od uživatele procedury `fib`, viz program 8.10.

Program 8.9. Iterativní procedura pro výpočet Fibonacciho čísel.

```
(define fib-iter
  (lambda (a b i)
    (if (<= i 0)
        a
        (fib-iter b (+ a b) (- i 1)))))

(define fib
  (lambda (n)
    (fib-iter 0 1 n)))
```

Obrázek 8.11. Schématické zachycení aplikace iterativní verze `fib`.

`(fib 5)`	
`(fib-iter 0 1 5)`	*navíjení*: vyvolání 1. aplikace
`(fib-iter 1 1 4)`	*navíjení*: vyvolání 2. aplikace
`(fib-iter 1 2 3)`	*navíjení*: vyvolání 3. aplikace
`(fib-iter 2 3 2)`	*navíjení*: vyvolání 4. aplikace
`(fib-iter 3 5 1)`	*navíjení*: vyvolání 5. aplikace
`(fib-iter 5 8 0)`	*navíjení*: vyvolání 6. aplikace
`5`	*dosažení limitní podmínky a vrácení hodnoty*

V této sekci jsme ukázali, že různé rekurzivní procedury generují kvalitativně různé výpočetní procesy s různou náročností na výpočetní zdroje počítače. S tímto faktem je vždy potřeba dopředu počítat. Při programování bychom se měli soustředit jak na efektivitu procedur tak na čistotu jejich provedení a na jejich celkovou čitelnost. Efektivita a čistota provedení jdou leckdy proti sobě, u konkrétních aplikací je tedy potřeba najít přijatelný kompromis. Uživatele obvykle zajímá „rychlost programu“, což hovoří pro větší důraz na efektivitu. Na druhou stranu uživatele také často zajímá „rozšiřitelnost, přizpůsobitelnost a přenositelnost programu“, což naopak vede k důrazu na čisté provedení programu (vysoce optimalizované programy jsou zpravidla těžko čitelné a jejich rozšiřitelnost je tím pádem těžší nebo prakticky nemožná).

Poznámka 8.19. Stromová rekurze je často používaným nástrojem při zpracování hierarchických dat. Touto problematikou se budeme zabývat především v další lekci. V této sekci jsme si představili stromově rekurzivní výpočetní procesy jako jeden typ procesů generovaných rekurzivními procedurami. Obecně nelze říct, jak by se na první pohled možná mohlo zdát, že stromově rekurzivní procesy jsou „neefektivní “, nebo že vždy vedou na exponenciální časovou složitost. Tak tomu *není*. Složitost totiž vždy stanovujeme *vzhledem k velikosti vstupních dat*. Pokud budou vstupní data reprezentována (nějakou) hierarchickou datovou strukturou s n uzly a tyto uzly budou nějakou procedurou procházeny s použitím stromové rekurze jeden po druhém, pak bude časová složitost této procedury lineární, i když šlo o proceduru generující stromově rekurzivní výpočetní proces.

8.4 Jednorázové použití procedur

U mnoha příkladů v předchozí lekci jsme při vytváření procedury řešící daný problém vytvořili *pomocnou proceduru*, kterou jsme z hlavní procedury *aplikovali pouze jednou*. Například tomu tak bylo v programech 8.9 a 8.10. Třeba v programu 8.9 byla pomocná iterativní procedura `fib-iter` použita jednorázově v těle

Program 8.10. Iterativní procedura pro výpočet Fibonacciho čísel s interní definicí.

```
(define fib
  (lambda (n)
    (define iter
      (lambda (a b i)
        (if (<= i 0)
            a
            (iter b (+ a b) (- i 1)))))
    (iter 0 1 n)))
```

procedury `fib`. Na žádném dalším místě programu (mimo tělo samotné procedury `fib-iter`) již k aplikaci procedury `fib-iter` nedochází. V předchozí sekci byla vždy pomocná procedura iterativní, obecně se ale můžeme do podobné situace dostat i v případě, kdy pomocná procedura generuje lineární nebo stromově rekurzivní výpočetní proces.

V jazyku Scheme máme k dispozici aparát pro stručnější definici rekurzivní procedury, která je po své definici jednorázově použita (s danými argumenty). K tomuto účelu slouží rozšíření speciální formy `let`, kterému říkáme *pojmenovaný* `let`. Popis tohoto rozšíření `let` následuje.

Definice 8.20 (speciální forma `let`, *pojmenovaná verze*)**.** Speciální forma *pojmenovaný* `let` se používá se třemi nebo více argumenty ve tvaru

```
(let ⟨jméno⟩ ((⟨symbol₁⟩ ⟨výraz₁⟩)
              (⟨symbol₂⟩ ⟨výraz₂⟩)
                 ⋮
              (⟨symbolₙ⟩ ⟨výrazₙ⟩))
  ⟨tělo₁⟩
  ⟨tělo₂⟩
    ⋮
  ⟨těloₖ⟩),
```

kde $n \geq 0$, $k \geq 1$; $\langle symbol_1 \rangle, \ldots, \langle symbol_n \rangle$ jsou vzájemně různé symboly, $\langle jméno \rangle$ je symbol (obecně se může jednat i o některý ze symbolů $\langle symbol_1 \rangle, \ldots, \langle symbol_n \rangle$), $\langle výraz_1 \rangle, \ldots, \langle výraz_n \rangle$ jsou libovolné výrazy jejichž vyhodnocením vznikají počáteční vazby příslušných symbolů, a $\langle tělo_1 \rangle, \ldots, \langle tělo_k \rangle$ jsou výrazy tvořící tělo. Aplikace pojmenované verze speciální formy `let` s výše uvedenými argumenty probíhá tak, že interpret tuto formu nahradí následujícím kódem

```
(let ()
  (define ⟨jméno⟩
    (lambda (⟨symbol₁⟩ ⟨symbol₂⟩ ··· ⟨symbolₙ⟩)
      ⟨tělo₁⟩
      ⟨tělo₂⟩
        ⋮
      ⟨těloₖ⟩))
  (⟨jméno⟩ ⟨výraz₁⟩ ⟨výraz₂⟩ ··· ⟨výrazₙ⟩)),
```

který je vyhodnocen v aktuálním prostředí. ■

Na první pohled se pojmenovaný `let` od tradiční formy `let` liší tím, že jeho první argument není seznam (vazeb), ale symbol. Ve zbylé části již ze syntaktického hlediska rozdíl není. Z předchozího popisu je vidět, že výskyty pojmenovaného `let` v programu jsou během vyhodnocování nahrazovány jiným kódem, ve kterém je použita tradiční speciální forma `let` k *vytvoření nového prostředí* (bez vazeb, všimněte si, že `let` má prázdný seznam vazeb). V těle speciální formy `let` je potom klasickým způsobem *definována procedura*, která je navázána na symbol ⟨*jméno*⟩. Tělo procedury je tvořeno tělem pojmenovaného `let` a argumenty procedury jsou symboly, které v pojmenovaném `let` označovaly jména nových vazeb. Tato procedura je potom jednorázově aplikována s argumenty jimiž jsou hodnoty vzniklé vyhodnocením výrazů předaných pojmenovanému `let`.

Pokud použijeme pojmenovaná `let` (to jest `let`, kde prvním argumentem je symbol – jméno) aniž bychom dané jméno použili v těle této speciální formy, pak má pojmenovaný `let` stejný efekt jako klasický `let`:

```
(let proc ((x (* 2 5))
           (y 20))
  (+ x y))  ⟹  30
```

což je dobře vidět, když si uvědomíme, že předchozí kód je ekvivalentní:

```
(let ()
  (define proc
    (lambda (x y)
      (+ x y)))
  (proc (* 2 5) 20))  ⟹  30
```

Z definice pojmenovaného `let` uvedené v předchozím příkladě je dobře vidět, že prostředí vyhodnocení těla je prostředí aplikace procedury navázané na symbol ⟨*jméno*⟩. Navíc v těle má symbol ⟨*jméno*⟩ aktuální vazbu jíž je právě aplikovaná procedura. Pomocí symbolu ⟨*jméno*⟩ a jeho vazby je tedy možné provádět aplikaci „sebe sama“. Například následující kód ukazuje provede sečtení čísel od 0 do 10 pomocí rekurzivní aplikace (jednorázové) procedury vytvořené touto speciální formou:

```
(let proc ((n 10))
  (if (= n 0)
      0
      (+ n (proc (- n 1)))))  ⟹  55
```

Předchozí použití pojmenovaného `let` je během vyhodnocování přepsáno na

```
(let ()
  (define proc
    (lambda (n)
      (if (= n 0)
          0
          (+ n (proc (- n 1))))))
  (proc 10))
```

Následující příklad ukazuje rekurzivní verzi procedury `fib` pro výpočet prvků Fibonacciho posloupnosti, která je definována pomocí pojmenovaného `let` a jednorázově aplikována s hodnotou `30`:

```
(let fib ((n 30))
  (if (<= n 1)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))  ⟹  832040
```

Jméno uvedené v pojmenovaném `let` se může shodovat s některým ze symbolů pojmenovávajícím vazby, i když to není účelné. V takovém případě je totiž vazba symbolu ⟨*jméno*⟩ překryta hodnotou předávaného argumentu, viz příklad:

```
(let f ((x (* 2 5))
        (y 20)
        (f 100))
  (list x y f))  ⟹  (10 20 100),
```

což je opět patrné, pokud si předchozí pojmenovaný `let` rozepíšeme:

```
(let ()
  (define f
    (lambda (x y f)
      (list x y f)))
  (f (* 2 5) 20 100))  ⟹  (10 20 100)
```

V programech 8.11 a 8.12 jsou uvedeny verze programů 8.6 a 8.10 ze stran 212 a 218, ve kterých byla interně definovaná a jednorázově aplikovaná procedura vytvořena pomocí pojmenovaného `let`.

Program 8.11. Iterativní procedura pro výpočet faktoriálu s interní definicí pomocí pojmenovaného `let`.

```
(define fac
  (lambda (n)
    (let iter ((i n)
               (accum 1))
      (if (<= i 1)
          accum
          (iter (- i 1) (* accum i))))))
```

Program 8.12. Iterativní procedura pro výpočet Fib. čísel s interní definicí pomocí pojmenovaného `let`.

```
(define fib
  (lambda (n)
    (let iter ((a 0)
               (b 1)
               (i n))
      (if (<= i 0)
          a
          (iter b (+ a b) (- i 1))))))
```

Použití pojmenovaného `let` je vhodné v případě, kdy potřebujeme okamžitě a jednorázově aplikovat právě definovanou pomocnou proceduru, která je rekurzivní. Z hlediska použití pojmenovaného `let` je přitom jedno, jaký rekurzivní výpočetní proces tato pomocní procedura generuje.

8.5 Rekurze a indukce na seznamech

Nyní se budeme blíže zabývat rekurzivními procedurami pracujícími se seznamy. Po teoretické stránce jde opět o klasické uživatelsky definované procedury, jak jsme se představili již v lekci 2. Rekurzivní procedury pracující se seznamy mohou stejně jako procedury pracující nad čísly generovat kvalitativně různé rekurzivní výpočetní procesy. Opět se jedná o lineárně rekurzivní výpočetní proces, lineárně iterativní výpočetní proces a stromově rekurzivní výpočetní proces.

Jako první si ukážeme rekurzivní variantu procedury `length` pro výpočet délky seznamu. Připomeňme, že tuto proceduru jsme již implementovali hned dvakrát. První verze používala `map` a `apply`, viz program 6.1 na straně 145. Druhou verzi jsme implementovali pomocí `foldr`, viz program 7.1 na straně 174. Varianta pomocí `foldr` byla efektivnější než verze používající `map` a `apply`. V programu 8.13 máme uvedenu rekurzivní verzi procedury `length`, která je co se týče efektivity stejně kvalitní jako verze z programu 7.1. Limitní podmínkou výpočtu délky seznamu je prázdný seznam, ten má délku nula. Pokud je seznam

Program 8.13. Výpočet délky seznamu pomocí rekurze.

```
(define length
  (lambda (l)
    (if (null? l)
        0
        (+ 1 (length (cdr l))))))
```

Program 8.14. Iterativní výpočet délky seznamu.

```
(define length
  (lambda (l)
    (let iter ((l l)
               (steps 0))
      (if (null? l)
          steps
          (iter (cdr l) (+ steps 1))))))
```

neprázdný, pak je rekurzivním předpisem problém výpočtu délky tohoto seznamu redukován na problém nalezení délky seznamu bez prvního prvku (rekurzivní aplikace) a přičtení jedničky (započtení prvního prvku výchozího seznamu). Všimněte si, že rekurzivní procedura z programu 8.13 je v podstatě jen přepisem rekurzivního definičního vztahu ze sekce 8.1.

Procedura `length` z programu 8.13 má prostorovou složitost $O(n)$, kde n je délka seznamu, protože při každé rekurzivní aplikaci je vytvořeno nové prostředí (rekurzivní aplikace `length` není koncová). Snadno bychom ale mohli `length` naprogramovat iterativně. K tomuto účelu bychom potřebovali (jednorázově používanou) pomocnou proceduru s dvěma argumenty. Jedním by byl seznam, který by sloužil jako čítač – – při každém kroku bychom z něj odebírali jeden prvek. Druhý argument by sloužil jako číselný střadač, který by počítal počet kroků. Iterativní verze `length` je uvedena v programu 8.14. Tato iterativní verze má prostorovou složitost $O(1)$, během aplikace jsou vytvořena vždy jen dvě prostředí a nejsou konstruovány žádné hierarchické datové struktury. Průběh aplikace iterativní verze `length` pro konkrétní vstupní seznam ve tvaru `(a 10.2 b #f)` je zobrazen na obrázku 8.12. Jak je z tohoto obrázku patrné, průběh iterativního procesu se ničím neliší od průběhu iterativních procesů uvedených v předchozí sekci. Na věci nic nemění to, že nyní zpracováváme seznam a limitní podmínka je formulována pro seznam. V dalších ukázkách v této sekci již tedy nebudeme podrobně znázorňovat průběhy jednotlivých výpočetních procesů a ponecháme je na laskavém čtenáři.

Nyní se budeme věnovat rekurzivnímu spojení dvou seznamů. V úvodu této lekce jsme ukázali rekurzivní definici zobrazení append2. V programu 8.15 je definována procedura `append2`, která reprezentuje rekurzivně definované append2 ze sekce 8.1. Limitní podmínkou je, pokud je první ze spojovaných seznamů prázdný, v tom případě je spojení rovno druhému seznamu. V opačném případě je problém spojení dvou seznamů redukován na problém spojení prvního seznamu bez prvního prvku s druhým seznamem (rekurzivní aplikace) a následné připojení prvního prvku prvního seznamu na začátek výsledku předchozího spojení. Proceduru `append2` jsme již také několikrát implementovali. Poprvé to byla neefektivní implementace v programu 5.2 na straně 124, která byla založená na `build-list` a přístupu k prvkům seznamu pomocí `list-ref`. Dále jsme provedli implementaci pomocí `foldr`, viz program 7.2 na straně 175. Z hlediska efektivity, je rekurzivní verze `append2` shodná s `append2` vytvořené pomocí `foldr`. Samotnou proceduru `foldr` bychom mohli rovněž snadno naprogramovat jako rekurzivní proceduru. Tímto a dalšími problémy se budeme zabývat v následující lekci.

Ne všechny procedury, se kterými jsme se setkali, však lze efektivně naprogramovat pomocí `foldr`. Přinejmenším to není z programátorského pohledu nijak „přímočaré“. Takovou procedurou je třeba procedura

Obrázek 8.12. Schématické zachycení aplikace iterativní verze `length`.

`(length (a 10.2 b #f))`	
`(iter (a 10.2 b #f) 0)`	*navíjení*: vyvolání 1. aplikace
`(iter (10.2 b #f) 1)`	*navíjení*: vyvolání 2. aplikace
`(iter (b #f) 2)`	*navíjení*: vyvolání 3. aplikace
`(iter (#f) 3)`	*navíjení*: vyvolání 4. aplikace
`(iter () 4)`	*navíjení*: vyvolání 5. aplikace
`4`	*dosažení limitní podmínky a vrácení hodnoty*

Program 8.15. Spojení dvou seznamů pomocí rekurze.

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

`list-ref`, která pro daný seznam a index vrací prvek seznamu nacházející se na pozici určené indexem. Snadno lze ale naprogramovat iterativní proceduru, která n-tý prvek seznamu vrací. Viz proceduru v programu 8.16. Tato procedura skutečně vždy generuje iterativní výpočetní proces, protože její jediná rekur-

Program 8.16. Rekurzivní verze proceduru vracející prvek na dané pozici v seznamu.

```
(define list-ref
  (lambda (l index)
    (if (= index 0)
        (car l)
        (list-ref (cdr l) (- index 1)))))
```

zivní aplikace se nachází v koncové pozici. Limitní podmínkou procedury je, pokud je index nulový. V tom případě vracíme první prvek seznamu, který lze získat v konstantním čase pomocí `car`. Pokud je index nenulový, pak víme, že hledaný prvek se nachází hlouběji v seznamu. Hledání n-tého prvku seznamu je tedy redukováno na hledání prvku na pozici $n-1$ v seznamu zkráceném o první prvek. Procedura `list-ref` má časovou složitost $O(n)$, kde n je pozice (index) v seznamu. Časová složitost tedy není závislá na délce vstupního seznamu, ale na pozici, ze které chceme prvek vrátit. Prostorová složitost je $O(1)$. Z hlediska složitosti jde o významný posun oproti verzi `list-ref` z programu 6.4 na straně 147, která měla při efektivní implementaci procedur `filter` (pracující v čase a prostoru $O(n)$) a `length` (z programu 8.14) časovou složitost $O(4n)$ a prostorovou složitost $O(3n)$ (zdůvodněte si podrobně proč).

Analogicky jako proceduru `list-ref` bychom mohli naprogramovat obecnější proceduru `list-tail`, která pro daný seznam a číselný argument n vrátí seznam bez prvních n prvků. Viz program 8.17. Jedná se opět o iterativní proceduru s časovou složitostí $O(n)$, kde n je počet vypuštěných prvků, a prostorovou složitostí $O(1)$. Z definice procedury `list-tail` je zřejmé, že pomocí ní bychom mohli definovat `list-ref`:

```
(define list-ref
  (lambda (l index)
    (car (list-tail l index))))
```

Mapování přes jeden seznam bychom mohli přímočaře naprogramovat pomocí rekurzivní procedury. V předchozích částech textu jsme již dvě implementace procedury `map1` (mapování přes jediný seznam)

Program 8.17. Procedura vracející seznam bez zadaného počtu prvních prvků.

```
(define list-tail
  (lambda (l skip)
    (if (= skip 0)
        l
        (list-tail (cdr l) (- skip 1)))))
```

viděli. První z nich byla v programu 5.3 na straně 126, využívala těžkopádně `build-list` a `list-ref` a byla hodně neefektivní. Další implementace byla ukázána v programu 7.4 na straně 176 a používala `foldr`. Tato verze již byla efektivní a efektivnější verzi z principu vytvořit nelze – při mapování totiž vytváříme nový seznam délky n, takže časová i prostorová složitost nemohou být řádově lepší než $O(n)$. Rekurzivní verze `map1` však může být čitelnější než verze vytvořená pomocí `foldr`. Rekurzivní verzi `map1` máme zobrazenu v programu 8.18. Mezním případem je opět případ pro prázdný seznam. V tomto případě je výsledkem

Program 8.18. Mapovací procedura pracující s jedním seznamem pomocí rekurze

```
(define map1
  (lambda (f l)
    (if (null? l)
        '()
        (cons (f (car l))
              (map1 f (cdr l))))))
```

mapování procedury přes prázdný seznam opět prázdný seznam. V případě, že je daný seznam neprázdný, je výsledkem mapování seznam vzniklý připojením modifikace prvního prvku na výsledek mapování přes seznam bez prvního prvku.

Jednou z procedur, kterou by bylo bez rekurze těžké vytvořit, je procedura `build-list` sloužící k vytváření seznamů dané délky jež obsahují prvky dané jako výsledky aplikace procedury jednoho argumentu (argumentem je pozice prvku v seznamu). Naprogramovat proceduru `build-list` pomocí rekurze je přímočaré. Triviálním případem je vytvoření seznamu délky nula – ten je vždycky prázdný. V opačném případě lze redukovat problém vytvoření n-prvkového seznamu na problém vytvoření seznamu o $n-1$ prvcích na jehož počátek je připojen nový prvek. Rekurzivní procedura je uvedena v programu 8.19. V proceduře

Program 8.19. Rekurzivní verze procedury na vytváření seznamů.

```
(define build-list
  (lambda (n f)
    (let build-next ((i 0))
      (if (= i n)
          '()
          (cons (f i) (build-next (+ i 1)))))))
```

`build-list` je pomocí pojmenovaného `let` vytvořena jednorázově aplikovaná rekurzivní procedura vytvářející seznam. Tuto proceduru jsme vytvořili proto, že jsme potřebovali další argument pomocí nějž si předáváme informaci o pozici prvku, který chceme vytvořit. V limitní podmínka vyjadřuje zastavení navíjení (konstrukce seznamu) v případě, že již jsme na pozici vyskytující se „za posledním požadovaným prvkem“. Dokud není limitní podmínky dosaženo, jsou rekurzivně konstruovány prvky pomocí `cons` a pomocí předané procedury navázané na `f`. Implementace `build-list` z programu 8.19 má časovou i prostorovou složitost $O(n)$, kde n je délka seznamu, který chceme vytvářet. Jedná se tedy o maximálně efektivní implementaci (lepší řádové složitosti již nelze dosáhnout).

V programu 8.19 jsme výsledný seznam konstruovali jakoby odpředu. To jest od prvku s nejnižším indexem až k prvku s nejvyšším indexem. Mohli bychom postupovat i obráceně. To by mělo zdánlivou výhodu v tom, že bychom nemuseli definovat pomocnou proceduru. Řešení je uvedeno v programu 8.20. Při

Program 8.20. Neefektivní verze procedury na vytváření seznamů.

```
(define build-list-rev
  (lambda (n f)
    (if (= n 0)
        '()
        (append2 (build-list-rev (- n 1) f)
                 (list (f (- n 1)))))))
```

pozorném studiu programu záhy zjistíme, že program je ale velmi neefektivní. Je to způsobeno používáním `append2` při každé rekurzivní aplikaci. Pomocí `append2` spojujeme seznamy délky $n-1$ s jednoprvkovým seznamem – tím realizujeme připojení nově vytvořeného prvku na konec seznamu. Nejefektivnější verze `append2` potřebuje n kroků pro spojení prvního seznamu délky n s libovolným seznamem. Tím pádem procedura `build-list-rev` potřebuje postupně $n-1, n-2, \ldots, 1$ kroků k postupnému zařazení všech vytvářených prvků na konec seznamu. Součtem prvků aritmetické posloupnosti tedy získáme celkem $\frac{n\cdot(n-1)}{2}$ kroků. Časová složitost `build-list-rev` je tedy *kvadratická* vzhledem k délce vytvářeného seznamu. I prostorová složitost je v této třídě, protože během používání `append2` jsou postupně dokola konstruovány nové seznamy délek $n, n-1, \ldots, 1$. Celkově vzato je tedy procedura `build-list-rev` *velmi neefektivní.* Na této proceduře je taky dobré uvědomit si, že i když procedura generuje lineárně rekurzivní výpočetní proces, její časové složitost není lineární vzhledem ke velikosti vstupu. Slovo „lineární“ v pojmu „lineárně rekurzivní výpočetní proces“ se totiž vztahuje k počtu rekurzivních aplikací procedury. Vždy je ale potřeba ještě brát v potaz, co se (z hlediska časové náročnosti) děje v každé z aplikací.

Samotné vytváření seznamů počínaje posledním prvkem však není zavrženíhodná myšlenka. Místo rekurzivní verze konstrukce je však vhodné používat ke konstrukci seznamu iterativní proceduru, která bude nově vytvářené prvky postupně „střádat“ do nového seznamu. V programu 8.21 je uvedena procedura `build-list-iter` vytvářející seznam právě tímto způsobem. Procedura `build-list-iter` opět používá

Program 8.21. Rekurzivní verze procedury na vytváření seznamů.

```
(define build-list-iter
  (lambda (n f)
    (let iter ((i (- n 1))
               (accum '()))
      (if (< i 0)
          accum
          (iter (- i 1) (cons (f i) accum))))))
```

pomocnou proceduru vytvořenou pomocí pojmenovaného `let`. Tato pomocí procedura má dva argumenty, jedním je čítač jdoucí od $n-1$ (poslední pozice v konstruovaném seznamu) k 0 (první pozice v konstruovaném seznamu). V každém kroku konstrukce je tento čítač zmenšen o jedna. Iterace končí pokud je čítač menší než hodnota nula (všechny prvky již byly vytvořeny). Druhým argumentem pomocné procedury je akumulátor, ke kterému jsou postupně (zepředu) přidávány nově vytvořené prvky. Po dosažení limitní podmínky je vrácena hodnota akumulátoru. Časová a prostorová složitost tohoto řešení je $O(n)$. Oproti verzi z programu 8.19 je rekurzivní aplikace prováděna v jednom prostředí, protože jde o iteraci. I přesto je ale prostorová složitost $O(n)$, protože dojde k vytvoření n-prvkového seznamu.

Nyní se budeme věnovat efektivní proceduře pro převracení prvků seznamu. Jde nám tedy o efektivní implementaci procedury `reverse`. Stejně jako v případě vytváření seznamů „odzadu“, které jsme viděli v procedurách `build-list-rev` a `build-list-iter`, bude existovat několik řešení lišících se ve své efektivitě. Nejprve si ukážeme řešení, které není příliš efektivní. V programu 8.22 je uvedena lineárně rekurzivní procedura `reverse`. Tato procedura je založena na faktu, že převrácení prázdného seznamu

Program 8.22. Neefektivní verze převracení prvků v seznamu.

```
(define reverse
  (lambda (l)
    (if (null? l)
        '()
        (append (reverse (cdr l))
                (list (car l))))))
```

je triviální a pokud máme převrátit neprázdný seznam, pak stačí převrátit zbytek tohoto seznamu bez prvního prvku a potom nakonec takového seznamu připojit původní první prvek. Procedura `reverse` z programu 8.22 tedy trpí stejným neduhem jako lineárně rekurzivní procedura `build-list-rev` – při každé rekurzivní aplikaci je použita procedura `append2` s lineární časovou složitostí. Převrácení seznamu tímto způsobem má tedy opět řádově kvadratickou časovou i prostorovou složitost.

Efektivní verzi převrácení seznamu bychom mohli vytvořit pomocí iterativní procedury provádějící spojení dvou seznamů tak, že ve výsledku je první z těchto dvou seznamů spojen v opačném pořadí. Proceduru provádějící tento typ spojení nazveme `rev-append2` a její kód je uveden v programu 8.23. Procedura

Program 8.23. Iterativní verze převrácení prvků v seznamu.

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

(define reverse
  (lambda (l)
    (rev-append2 l '())))
```

`rev-append2` používá svůj první argument (první ze spojovaných seznamů) jako čítač a druhý argument (druhý ze spojovaných seznamů) jako akumulátor. Pokud je první ze spojovaných seznamů prázdný, je vrácen druhý seznam. V opačném případě je z prvního seznamu odebrán jeho první prvek, který je přidán na začátek druhého seznamu. Viz následující příklady použití procedury `rev-append2`:

```
(rev-append2 '() '())              ⟹  ()
(rev-append2 '() '(1 2 3 4))       ⟹  (1 2 3 4)
(rev-append2 '(a) '(1 2 3 4))      ⟹  (a 1 2 3 4)
(rev-append2 '(a b) '(1 2 3 4))    ⟹  (b a 1 2 3 4)
(rev-append2 '(a b c) '(1 2 3 4))  ⟹  (c b a 1 2 3 4)
(rev-append2 '(a b c) '())         ⟹  (c b a)
```

Nyní je jasné, že efektivní `reverse` můžeme naprogramovat jako proceduru, která pouze aplikuje proceduru `rev-append2` s druhým seznamem, který je prázdný, viz program 8.23. Procedura `rev-append2` a tím pádem i procedura `reverse` má lineární časovou i prostorovou složitost.

Nyní, na konci sekce, uvedeme implementaci třídění nazývaného *mergesort*. Tato metoda je založena na takzvaném slévání již setříděných seznamů. Ze dvou setříděných seznamů je totiž možné snadno vytvořit jeden setříděný seznam. Stačí nám totiž postupně odebírat vždy menší z prvních prvků těchto seznamů a konstruovat z nich výsledný seznam. Předveďme si to na konkrétních seznamech:

první seznam:	`(1 7 8)`	`(7 8)`	`(7 8)`	`(7 8)`	`(8)`	`()`
druhý seznam:	`(2 3 9 10)`	`(2 3 9 10)`	`(3 9 10)`	`(9 10)`	`(9 10)`	`(9 10)`
výsledek:	`()`	`(1)`	`(1 2)`	`(1 2 3)`	`(1 2 3 7)`	`(1 2 3 7 8)`

první seznam:	`()`	`()`
druhý seznam:	`(10)`	`()`
výsledek:	`(1 2 3 7 8 9)`	`(1 2 3 7 8 9 10)`

Proceduru na slévání dvou setříděných seznamů můžeme napsat třeba takto:

```
(define merge
  (lambda (s1 s2)
    (cond ((null? s1) s2)
          ((null? s2) s1)
          ((<= (car s1) (car s2)) (cons (car s1) (merge (cdr s1) s2)))
          (else (cons (car s2) (merge s1 (cdr s2)))))))
```

Jedná se tedy o rekurzívní proceduru. Její limitní podmínkou je to, že je aspoň jeden ze seznamů prázdný. Je-li této podmínky dosaženo, je vrácen zbývající (potenciálně neprázdný) seznam. Není-li limitní podmínky dosaženo, redukujeme problém na připojení prvku procedurou `cons` a slévání seznamů, z nichž je jeden původní a jeden vznikne z druhého původního odebráním prvního prvku.

V uvedené definici procedury `merge` používáme k porovnávání prvních prvků seznamů predikát `<=`. To samozřejmě není jediné uspořádání, podle kterého můžeme chtít slévat. Jednoduchým rozšířením procedury `merge` na proceduru vyššího řádu, které budeme mimo setříděných seznamů předávat navíc predikát určující uspořádání, dosáhneme výrazného zobecnění. Proceduru navíc napíšeme tak, aby tento nový argument byl volitelný. Následuje definice rozšířené procedury.

```
(define merge
  (lambda (s1 s2 . pred?)
    (let ((<= (if (null? pred?) <= (car pred?))))
      (let merge ((s1 s1)
                  (s2 s2))
        (cond ((null? s1) s2)
              ((null? s2) s1)
              ((<= (car s1) (car s2)) (cons (car s1) (merge (cdr s1) s2)))
              (else (cons (car s2) (merge s1 (cdr s2)))))))))
```

V těle nové verze procedury `merge` vytváříme lokální prostředí s vazbou na symbol `<=`. V případě, že je proceduře `merge` předán volitelný argument (seznam navázaný na symbol `pred?`) je tedy neprázdný, bude na symbol `<=` navázán tento argument, tedy první prvek seznamu `pred?`. V opačném případě ponecháme symbolu `<=` jeho původní význam. V tomto lokálním prostředí pak pomocí pojmenovaného `let` vytváříme a jednorázově aplikujeme proceduru `merge`, která se nijak neliší od její předchozí verze.

Novou proceduru `merge` tedy můžeme použít se dvěma argumenty, v tom případě se bude chovat stejně jako ta původní:

```
(merge '(1 5 6 7 9) '(2 2 3 4 4 4 10))  ⟹  (1 2 2 3 4 4 4 5 6 7 9 10)
```

nebo se třemi argumenty. Tehdy je třetím argumentem určeno uspořádání, v jakém by měly být setříděny slévané seznamy, a jakým způsobem probíhá výběr „menšího prvku“ při samotném slévání:

```
(merge '(9 7 6 5 1) '(10 4 4 4 3 2 2) >=)  ⟹  (10 9 7 6 5 4 4 4 3 2 2 1)
```

Právě to nám umožňuje použít tuto proceduru ke slévání jiných seznamů než číselných. Můžeme například slévat seznamy obsahující racionální čísla v reprezentaci představené v sekci 4.6:

```
(merge (list (make-r 1 3) (make-r 2 3))
       (list (make-r 1 2) (make-r 2 2))
       r<=)  ⟹  ((1 . 3) (1 . 2) (2 . 3) (1 . 1))
```

Procedura slévání seznamů je tedy základem třídící metody mergesort. Celý popis této metody je následující:

Vstup: seznam (a predikát uspořádání)

Výstup: setříděný seznam

Postup:

- Pokud je seznam prázdný nebo jednoprvkový, je již setříděný a je výstupem.
- Jinak seznam rozdělíme na dvě části (seznamy), každou z nich setřídíme algoritmem mergesort.
- Tyto dva setříděné seznamy slijeme do jednoho, výsledný setříděný seznam je pak výstupem.

Samotné slévání bychom tedy měli implementované. Zbývá nám implementace procedury na rozdělení seznamu na dvě části. Při dělení nám nezáleží na zachování pořadí prvků, protože výsledné seznamy budou stejně setříděny metodou mergesort. Můžeme proto během jednoho průchodu projít všechny prvky seznamu a střídavě je dávat do dvou různých seznamů. Tím bude zajištěna časová složitost $O(n)$, kde n je délka seznamu. Implementace takové procedury by vypadala takto:

```
(define divide-list
  (lambda (l)
    (let divide
        ((l l)
         (1st '())
         (2nd '()))
      (if (null? l)
         (cons 1st 2nd)
         (divide (cdr l) 2nd (cons (car l) 1st))))))
```

Pomocí pojmenovaného `let` jsme nadefinovali iterativní proceduru `divide`. Její argumenty `1st` a `2nd` mají roli střadačů. Při každém kroku přidáváme první prvek seznamu `l` do jednoho z nich. Přitom je stále zaměňujeme, všimněte si, že při rekurzivním volání předáváme proceduře `divide` střadač `2nd` jako argument `1st`. Tím je dosaženo rovnoměrného rozdělení. Iterace končí po průchodu celým seznamem `l`, pak vracíme tečkový pár obou střadačů. Viz příklady aplikace této procedury.

```
(divide-list '())             ⟹  (())
(divide-list '(1))            ⟹  (() 1)
(divide-list '(1 2 3 5 6 7))  ⟹  ((6 4 2) 7 5 3 1)
```

Nyní tedy napíšeme proceduru mergesort.

```
(define mergesort
  (lambda (l . pred?)
    (let ((<= (if (null? pred?) <= (car pred?))))
      (if (or (null? l) (null? (cdr l)))
          l
          (let ((divided-list (divide-list l)))
            (merge (mergesort (car divided-list) <=)
                   (mergesort (cdr divided-list) <=)
                   <=))))))
```

Jedná se o přímý přepis předpisu uvedeného výše. Procedurou `mergesort` můžeme třídit libovolné seznam na základě predikátu určujícího uspořádání jejich prvků. Viz následující příklady:

```
(mergesort '(2 5 3 4 1))     ⟹  (1 2 3 4 5)
(mergesort '(2 5 3 4 1) >=)  ⟹  (5 4 3 2 1)
```

Následující příklad ukazuje třídění seznamu racionálních čísel reprezenrovaných páry:

```
(map r->number
     (mergesort (list (make-r 2 1)
                      (make-r 1 2)
                      (make-r 3 2))
                r<=))  ⟹  (1 3/2 2)
```

8.6 Reprezentace polynomů

V této sekci se budeme věnovat reprezentaci polynomů a procedurami manipulujícími s polynomy v této reprezentaci. Jedná se o komplexní příklad na kterém předvedeme práci se seznamy, použití rekurze, a použití procedur `apply`, `eval` a akumulačních procedur představených v předcházející lekci.

Polynomy jedné proměnné budeme reprezentovat pomocí seznamů čísel. Číslo na pozici i v naší reprezentaci polynomu bude představovat koeficient členu stupně i. Tedy například polynom

$$5x^3 - 2x^2 + 1$$

bude reprezentován seznamem `(1 0 -2 5)`. Konstantní polynomy jsou tedy reprezentovány jednoprvkovými číselnými seznamy, například nulový a jednotkový polynom definujeme takto:

```
(define the-zero-poly '(0))
(define the-unit-poly '(1))
```

U reprezentací polynomu přitom nepovolujeme nadbytečné nuly na konci seznamu. Třeba seznamy

`(1 2 0)`, `(1 2 0 0)`, `(1 2 0 0 0)`

nejsou reprezentacemi polynomů. Z tohoto pravidla vyjímáme nulový polynom, který je vždy reprezentován jednoprvkovým seznamem `(0)`. Dále prázdný seznam není reprezentací polynomu.

První procedurou v této sekci je konstruktor polynomu `poly`. Jedná se o proceduru libovolného počtu argumentů. Těmito argumenty jsou čísla – koeficienty polynomu, seřazené vzestupně podle řádu. Procedura `poly` vrací seznam reprezentující polynom se zadanými koeficienty. Činnost procedury si dobře uvědomíme na příkladech její aplikace:

`(poly)`	⟹	`(0)`	*reprezentace polynom* 0
`(poly 0)`	⟹	`(0)`	*reprezentace polynom* 0
`(poly 0 0)`	⟹	`(0)`	*reprezentace polynom* 0
`(poly -1 2)`	⟹	`(-1 2)`	*reprezentace polynomu* $2x - 1$
`(poly -1 0 2)`	⟹	`(-1 0 2)`	*reprezentace polynomu* $2x^2 - 1$
`(poly -1 0 2 0 0 0)`	⟹	`(-1 0 2)`	*reprezentace polynomu* $2x^2 - 1$
`(poly -1 0 2 0 0 0 3)`	⟹	`(-1 0 2 0 0 0 3)`	*reprezentace polynomux* $3x^6 + 2x^2 - 1$

Nyní se podíváme na implementaci této procedury:

```
(define poly
  (lambda coeffs
    (let ((result
           (let cons-poly
               ((coeffs coeffs)
                (zero-accum '()))
             (cond ((null? coeffs) '())
                   ((= (car coeffs) 0) (cons-poly (cdr coeffs)
                                                  (cons 0 zero-accum)))
                   (else (append zero-accum
                                 (list (car coeffs))
                                 (cons-poly (cdr coeffs) '())))))))
      (if (null? result)
          the-zero-poly
          result))))
```

V těle procedury `poly` definujeme pomocí pojmenovaného `let` rekurzivní proceduru `cons-poly`, která bere dva argumenty. Prvním argumentem je seznam zbývajících koeficientů `coeffs`. Druhý argument je seznam `zero-accum`, který má funkci střadače. V tom si procedura `cons-poly` pamatuje počet nul od posledního nenulového koeficientu. Důvodem, proč tento střadač potřebujeme, je to, že reprezentace polynomu nesmí obsahovat nuly na konci. Pokud při zpracování seznamu koeficientů narazíme na nulu, nemůžeme v daném okamžiku vědět, jestli *všechny* další koeficienty nebudou také nulové. Proto nulové koeficienty střádáme do seznamu `zero-accum` a použijeme je až při nalezení nenulového koeficientu. Limitní podmínkou procedury `cons-poly` je skutečnost, že je seznam nezpracovaných koeficientů prázdný. Pokud je splněna, je vrácen prázdný seznam. V opačném případě mohou nastat dvě situace. Buďto je další nezpracovaný koeficient (hlava seznamu `coeffs`) roven nule, a pak přidáme nulu do střadače `zero-accum` a pokračujeme zpracováním zbytku koeficientů procedurou `cons-poly`. Nebo je nenulový a v tom případě přidáme všechny nuly ze střadače `zero-accum` a tento nenulový koeficient do seznamu, který vznikne aplikací procedury `cons-poly` seznam `coeffs` bez prvního prvku a prázdný střadač.
Výsledkem aplikace procedury `cons-poly` bude seznam, který nemá na konci nuly. Může se však stát, že tento seznam bude prázdný. To nastane v případě, že všechny prvky seznamu `coeffs` budou čísla `0`. Pokud tedy bude výsledkem prázdný seznam, budeme vracet reprezentaci nulového polynomu.

Pomocí právě nadefinovaného konstruktoru `poly` a procedury `apply` můžeme vytvořit proceduru konvertující seznam na reprezentaci polynomu:

```
(define list->poly
  (lambda (l) (apply poly l)))
```

Dále vytvoříme predikáty `constant-poly?` a `zero-poly?`. Jejich definice je velice jednoduchá proto kód nebudeme nijak komentovat. Tyto predikáty využijeme při implementaci dalších procedur v této sekci.

```
(define constant-poly?
  (lambda (p)
    (and (pair? p)
         (number? (car p))
         (null? (cdr p)))))

(define zero-poly?
  (lambda (p)
    (equal? p the-zero-poly)))
```

Nyní nadefinujeme proceduru `poly-degree`. Tato procedura bude pro zadaný polynom vracet jeho stupeň.

```
(define poly-degree
  (lambda (p)
    (if (constant-poly? p)
        0
        (+ 1 (poly-degree (cdr p))))))
```

Proceduru `poly-degree` jsme vytvořili jako jednoduchou rekurzivní proceduru. Limitní podmínkou je konstantnost polynomu. Je-li této podmínky dosaženo, je výsledkem nula. V opačném případě redukujeme problém na přičtení jedničky a nalezení stupně polynomu, který vznikne z původního polynomu odebráním prvního prvku z jeho reprezentace (což je v podstatě vydělení polynomu polynomem x). Uvedený kód je velice podobný programu 8.13 na straně 221, kde jsme implementovali proceduru zjišťující délku seznamu.

Procedura `poly-degree` tedy zjišťuje stupeň polynomu, jak se můžeme přesvědčit na následujících ukázkách použití této procedury:

```
(poly-degree (poly))                ⟾  0
(poly-degree (poly 0))              ⟾  0
(poly-degree (poly 0 0))            ⟾  0
(poly-degree (poly -1 2))           ⟾  1
(poly-degree (poly -1 0 2))         ⟾  2
(poly-degree (poly -1 0 2 0 0 0))   ⟾  2
```

```
(poly-degree (poly -1 0 2 0 0 0 3))  ⟹  5
```

Další dvě procedury budou provádět násobení polynomu speciálními polynomy. Konkrétně procedura `poly*xn` bude násobit vstupní polynom polynomem x^n a procedura `poly*c` bude násobit vstupní polynom konstantním polynomem c. Násobením polynomem x^n je vzhledem k naší reprezentaci přidání n nul na začátek seznamu. Následuje implementace procedury `poly*xn` provádějící násobení polynomem x^n:

```
(define poly*xn
  (lambda (p n)
    (if (= n 0)
        p
        (cons 0 (poly*xn p (- n 1))))))
```

Proceduru `poly*xn` jsme tedy napsali jako rekurzivní proceduru. Limitní podmínkou je rovnost čísla `n` nule. V takovém případě vracíme původní polynom. V opačném případě přidáváme nulu na začátek reprezentace součinu polynomu s polynomem x^{n-1}. Ve skutečnosti bychom měli ještě provádět test na zjištění, jestli polynom `p` není nulový a v takovém případě vracet vždy nulový polynom. Úpravu necháváme na čtenáři. Následují příklady volání:

```
(poly*xn (poly 1 0 -2) 0)  ⟹  (1 0 -2)
(poly*xn (poly 1 0 -2) 1)  ⟹  (0 1 0 -2)
(poly*xn (poly 1 0 -2) 2)  ⟹  (0 0 1 0 -2)
(poly*xn (poly 1 0 -2) 3)  ⟹  (0 0 0 1 0 -2)
```

Násobení polynomu konstantou c bude velice jednoduché. V případě, že je tato konstanta rovna nule, vracíme nulový polynom. Jinak stačí mapovat proceduru násobení konstantou na seznam reprezentující polynom. Definice takové procedury by vypadala takto:

```
(define poly*c
  (lambda (p c)
    (if (= c 0)
        the-zero-poly
        (map (lambda (x) (* c x)) p))))
```

Procedura `poly*c` tedy vrací polynom vynásobený konstantou:

```
(poly*c (poly 1 0 -2) 0)  ⟹  (0)
(poly*c (poly 1 0 -2) 1)  ⟹  (1 0 -2)
(poly*c (poly 1 0 -2) 2)  ⟹  (2 0 -4)
(poly*c (poly 1 0 -2) 3)  ⟹  (3 0 -6)
```

Další část sekce budeme věnovat operacím nad polynomy. Přesněji budeme implementovat procedury sčítání, odčítání násobení a dělení polynomů. Nejdříve tedy sčítání:

```
(define poly2+
  (lambda (p1 p2)
    (list->poly
     (let add ((p1 p1)
               (p2 p2))
       (cond ((null? p1) p2)
             ((null? p2) p1)
             (else (cons (+ (car p1) (car p2))
                         (add (cdr p1) (cdr p2)))))))))
```

V těle procedury `poly2+` jsme vytvořili pomocí pojmenovaného `let` rekurzivní proceduru `add`. Procedura realizuje jednoduché sčítání členů polynomů po odpovídajících si složkách (konstantní člen prvního polynomu je přičten ke konstantnímu členu druhého polynomu, stejně tak pro lineární, kvadratický a další členy). Jediným problémem, který je potřeba ošetřit, jsou obecně různé stupně sčítaných polynomů, což se projeví v různě dlouhých vstupních seznamech.

Limitní podmínkou procedury `poly2+` je skutečnost, že jeden ze vstupních seznamů je prázdný. V tom případě je vrácen druhý seznam. Prázdný seznam sice není reprezentací polynomu, ale může se stát že při rozkladu problému bude proceduře `add` předán prázdný seznam. Problém na nalezení součtu polynomů zde rozkládáme na problém nalezení součtu jednodušších polynomů, které vzniknou z původních odebráním prvního prvku jejich reprezentace (aplikací procedury `car`), a přidání součtu těchto prvních prvků na začátek (aplikací procedury `cons`). Výsledný seznam ale nemusí být platnou reprezentací polynomu. Například budou-li vstupem procedury `add` reprezentace polynomů $x + 2$ a $-x - 1$, tedy vstupem budou seznamy `(2 1)` a `(-1 -1)`, bude výsledkem její aplikace seznam `(1 0)`. To je dvouprvkový seznam končící nulou a tedy není platnou reprezentací seznamu. Proto na výsledek, který vrátí procedura `add`, aplikujeme ještě proceduru `list->poly`, kterou jsme nadefinovali výše. Viz příklady použití:

```
(poly2+ '(0) '(1 2 -3))          ⟹  (1 2 -3)
(poly2+ '(1 2 3) '(0))           ⟹  (1 2 3)
(poly2+ '(1 2 3) '(1 2 -3))      ⟹  (2 4)
(poly2+ '(1 2 3) '(-1 2 -3))     ⟹  (0 4)
(poly2+ '(1 2 3) '(-1 -2 -3))    ⟹  (0)
```

Uvedená implementace procedury `poly2+` je na jednu stranu elegantní, ale druhou stranu není příliš efektivní. Provádíme v ní vlastně dva průchody seznamem. V prvním ze dvou seznamů reprezentujících polynomy vytváříme seznam součtů koeficientů. Druhý průchod je proveden aplikací pomocné procedury `list->poly`, kdy z takto vytvořeného seznamu vytvoříme reprezentaci polynomu. Tyto dvě činnosti můžeme provádět současně a tak kód zefektivnit. Program definice procedury `poly2+` by pak mohl vypadat:

```
(define poly2+
  (lambda (p1 p2)
    (let ((result
           (let add ((p1 p1)
                     (p2 p2)
                     (zero-accum '()))
             (cond ((and (null? p1) (null? p2)) '())
                   ((null? p1) (append zero-accum p2))
                   ((null? p2) (append zero-accum p1))
                   (else (let ((sum (+ (car p1) (car p2))))
                           (if (= sum 0)
                               (add (cdr p1) (cdr p2) (cons 0 zero-accum))
                               (append zero-accum
                                       (list sum)
                                       (add (cdr p1) (cdr p2) '())))))))))
      (if (null? result)
          the-zero-poly
          result))))
```

Tato definice `poly2+` se liší od předchozí především v proceduře `add`, která nyní bere o jeden argument navíc. Tento argument `zero-accum` je pomocný a procedura `add` jej bude používat k zapamatování si počtu nul od posledního nenulového koeficientu, podobně jako tomu je u implementace procedury `list->poly`. Procedura `add` je rekurzivní a stejně jako v předchozí implementaci je limitní podmínkou skutečnost, že je jeden ze vstupních seznamů prázdný. V tom případě vracíme druhý z nich. V případě, že limitní podmínka nebyla splněna, je vypočten součet prvních prvků vstupních seznamů. S tímto číslem pak procedura zachází podobně jako procedura `list->poly`. Tedy pokud je součet nulový, je rekurzivně volána procedura `add`, se seznamy bez prvních prvků a s pomocným seznamem `zero-accum`, do kterého přidáme nuly. Je-li naopak součet nenulový, je procedura `add` aplikována na ocasy seznamu, ale jako argument `zero-accum` je předán prázdný seznam. Nuly ze seznamu `zero-accum` a vypočtený součet prvních členů je připojen na začátek seznamu, který bude výsledkem této aplikace procedury `add`. Výsledkem aplikace procedury `add` bude vždy reprezentace polynomu, až na jednu výjimkou. Tou je prázdný seznam. A to vyřešíme jednoduchou podmínkou.

Jelikož je sčítání polynomů monoidální operace, můžeme pomocí procedury `poly2+` a akumulační procedury `foldr` nadefinovat proceduru na sčítání libovolného množství polynomů:

```
(define poly+ (lambda polys (foldr poly2+ the-zero-poly polys)))
```

Tuto proceduru tedy můžeme použít ke sčítání jakéhokoli počtu polynomů. Viz příklady aplikace:

```
(poly+)                                                 ⟹  (0)
(poly+ '(1 2 3))                                        ⟹  (1 2 3)
(poly+ '(1 2 3) '(10 20 30))                            ⟹  (11 22 33)
(poly+ '(1 2 3) '(10 20 30) '(-11 0 -33))               ⟹  (0 22)
(poly+ '(1 2 3) '(10 20 30) '(-11 0 -33) '(0 -22))      ⟹  (0)
```

Dále můžeme pomocí procedury sčítání dvou polynomů implementovat proceduru odčítání jednoho polynomu od druhého. Odečtení polynomu je vlastně přičtením opačného polynomu, ve smyslu vynásobení konstantou -1. Implementace je tedy velice přímočará:

```
(define poly-
  (lambda (p1 p2)
    (poly+ p1 (poly*c p2 -1))))
```

Procedura `poly2*` bude vracet součin dvou polynomů:

```
(define poly2*
  (lambda (p1 p2)
    (if (or (zero-poly? p1) (zero-poly? p2))
        the-zero-poly
        (let mul ((p1 p1)
                  (p2 p2))
          (if (null? p1)
              '()
              (poly2+ (poly*c p2 (car p1))
                      (poly*xn (mul (cdr p1) p2) 1)))))))
```

Nejdříve jsme vyřešili speciální případ, tedy to, že alespoň jeden z polynomů byl nulový. K samotnému násobení jsme pak využili procedur `poly2+`, `poly*c` a `poly*xn`, které jsme nadefinovali výše v této sekci.

```
(poly2* '(0) '(1 2 3))          ⟹  (0)
(poly2* '(1 2 3) '(0))          ⟹  (0)
(poly2* '(1) '(1 2 3))          ⟹  (1 2 3)
(poly2* '(0 1) '(1 2 3))        ⟹  (0 1 2 3)
(poly2* '(-2 0 1) '(1 2 3))     ⟹  (-2 -4 -5 2 3)
```

Stejně jako sčítání polynomů i násobení polynomů je monoidální operace. Proto můžeme podobným způsobem jako jsme dříve implementovali proceduru `poly+` vytvořit proceduru `poly*`, která bude počítat součin libovolného počtu polynomů:

```
(define poly*
  (lambda polys
    (foldr poly2* the-unit-poly polys)))
```

A používat ji s libovolným počtem argumentů:

```
(poly*)                                 ⟹  (1)
(poly* '(1 2 3) '(0 1))                 ⟹  (0 1 2 3)
(poly* '(1 2 3) '(0 1) '(0 1))          ⟹  (0 0 1 2 3)
(poly* '(1 2 3) '(0 1) '(0 1) '(2))     ⟹  (0 0 2 4 6)
```

Poslední zbývající operací je dělení polynomů, představované procedurou `poly/`. Kód si uvedeme včetně jednoduchých komentářů.

```
(define poly/
  (lambda (p1 p2)

    ;; pomocná procedura: vrať poslední prvek neprázdného seznamu
    (define last
```

```
      (lambda (l)
        (if (null? (cdr l))
            (car l)
            (last (cdr l)))))

;; vrať seznam (podíl zbytek)
(let poly/
   ((p1 p1)
    (p2 p2))

 ;; rozlišení situace podle tvaru dělitele
 (cond

  ;; nulový dělitel
  ((zero-poly? p2) 'divided-by-zero)

  ;; dělitel je konstantní polynom
  ((= (poly-degree p2) 0) (list (poly*c p1 (/ 1 (car p2))) '()))

  ;; stupeň dělence je menší než stupeň dělitele
  ((< (poly-degree p1) (poly-degree p2)) (list '() p1))

  ;; stupeň dělence je větší nebo roven stupni dělitele
  (else (let* ((leader (poly*xn (list (/ (last p1) (last p2)))
                                (- (poly-degree p1) (poly-degree p2))))
               (rest (poly/ (poly- p1 (poly* leader p2)) p2)))
          (list (poly+ leader (car rest)) (cadr rest)))))))
```

Tato procedura nám tedy vrací dvouprvkový seznam, jehož prvním prvkem je výsledek dělení a druhým prvkem je zbytek po tomto dělení. Neřešili jsme skutečnost, že tento výsledný seznam může obsahovat prázdný seznam, což není reprezentace polynomu. Úpravu opět ponecháváme na čtenáři. Viz příklady použití procedury:

```
(poly/ '(1 2 3) '(0))        ⟹  „CHYBA: Pokus o dělení nulou.“
(poly/ '(1 2 3) '(1))        ⟹  ((1 2 3) ())
(poly/ '(1 2 3) '(2))        ⟹  ((1/2 1 1 1/2) (0))
(poly/ '(1 2 3) '(10))       ⟹  ((1/10 1/5 3/10) (0))
(poly/ '(1 2 3) '(0 1))      ⟹  ((2 3) (1))
(poly/ '(1 2 3) '(0 2))      ⟹  ((1 1 1/2) (1))
(poly/ '(1 2 3) '(5 2))      ⟹  ((-11/4 1 1/2) (59/4))
(poly/ '(1 2 3) '(5 2 3))    ⟹  ((1) (-4))
(poly/ '(1 2 3) '(5 2 3 4))  ⟹  ((0) (1 2 3))
```

Pomocí procedury `poly/` jednoduše definujeme procedury `poly-quotient` a `poly-modulo` na zjišťování podílu a zbytku po dělení:

```
(define poly-quotient
  (lambda (p q)
    (car (poly/ p q))))

(define poly-modulo
  (lambda (p q)
    (cadr (poly/ p q))))
```

Též můžeme napsat proceduru zjišťující hodnotu polynomu v daném bodě k, ta je totiž totožná se zbytkem po dělení polynomem $k - x$:

```
(define poly-value
```

```
  (lambda (p k)
    (car (poly-modulo p (poly (- k) 1)))))
```

Popřípadě můžeme s využitím curryingu vytvářet z polynomů polynomiální funkce:

```
(define poly-function
  (lambda (p)
    (lambda (k)
      (poly-value p k))))
```

Procedura `poly-sexpr` bude brát dva argumenty – polynom ⟨*p*⟩ a symbol ⟨*var*⟩ – a bude převádět polynom na vyhodnotitelný S-výraz. Navíc bude eliminovat násobení nulou a upravovat násobení jedničkou a přičítání nuly. Viz příklady aplikace této procedury:

```
(poly-sexpr '(0) 'x)       ⟹  0
(poly-sexpr '(2) 'x)       ⟹  2
(poly-sexpr '(0 1) 'x)     ⟹  x
(poly-sexpr '(2 1) 'x)     ⟹  (+ 2 x)
(poly-sexpr '(0 0 1) 'x)   ⟹  (* x x)
(poly-sexpr '(0 1 1) 'x)   ⟹  (+ x (* x x))
(poly-sexpr '(0 2 1) 'x)   ⟹  (+ (* 2 x) (* x x))
(poly-sexpr '(3 0 1) 'x)   ⟹  (+ 3 (* x x))
(poly-sexpr '(3 1 1) 'x)   ⟹  (+ 3 x (* x x))
(poly-sexpr '(3 2 1) 'x)   ⟹  (+ 3 (* 2 x) (* x x))
```

Nyní se podrobně podíváme na následující definici této procedury:

```
(define poly-sexpr
  (lambda (p var)
    (if (constant-poly? p)
        (car p)
        (let* ((result
                (let build ((p (cdr p))
                            (degree 1))
                  (cond ((null? p) '())
                        ((= (car p) 0) (build (cdr p) (+ degree 1)))
                        (else
                         (cons (if (and (= (car p) 1) (= degree 1))
                                   var
                                   (append (if (= (car p) 1)
                                               '(*)
                                               (list '* (car p)))
                                           (build-list degree
                                                       (lambda (i) var))))
                               (build (cdr p) (+ degree 1)))))))
               (result (if (= (car p) 0) result (cons (car p) result))))
          (cond ((null? result) 0)
                ((null? (cdr result)) (car result))
                (else (cons '+ result)))))))
```

V těle procedury `poly-sexpr` jsme nejdříve vyřešili speciální případ polynomů, kterým je konstantní polynom. S-výraz odpovídající takovému polynomu je právě jediný prvek z jeho reprezentace. Není-li polynom konstantní, procházíme procedurou `build` jeho reprezentaci – bez konstantního členu – a konstruujeme podle ní nový seznam, který obsahuje místo každého nenulového prvku ⟨*coef*⟩ seznam

    (* ⟨coef⟩ ⟨var⟩ ··· ⟨var⟩).

Tento seznam obsahuje tolikrát symbol ⟨*var*⟩, kolik je stupeň zpracovávaného členu polynomu. Stupeň aktuálního polynomu si přitom předáváme pomocí argumentu `degree` procedury `build`. Toto „nahrazování“

má výjimku. Výjimkou jsou členy jejichž koeficient je 1, v takovém případě do seznamu nedáváme ⟨*coef*⟩ (řešíme násobení jedničkou), pokud má navíc tento člen stupeň 1, nevytváříme seznam, ale nahrazujeme jej pouze symbolem ⟨*var*⟩. Na začátek takto vzniklého seznamu přidáme konstantní člen polynomu, je-li nenulový. Pokud je takto vzniklý seznam jednoprvkový, vracíme jeho jediný prvek. V opačném případě vracíme tento seznam s přidaným symbolem + na jeho začátku.

Poznámka 8.21. Ve zbytku této sekce budeme v příkladech použití implementovaných procedur používat k výpisu reprezentace polynomů výsledky aplikace procedury `poly-sexpr`.

Pomocí této procedury můžeme napsat proceduru nalezení hodnoty polynomu v daném bodě. Jednoduše vytvoříme `let`-blok, který bude vázat symbol `x`, a jehož těle bude S-výraz, který vytvoříme pomocí procedury `poly-sexpr`, ze zadaného polynomu a symbolu `x`. Takto zkonstruovaný `let`-blok vyhodnotíme pomocí procedury `eval`. Následuje definice právě popsané procedury:

```
(define poly-sexpr-value
  (lambda (p x)
    (eval (list 'let (list (list 'x x))
                (poly-sexpr p 'x)))))
```

Podobným způsobem jako jsme definovali proceduru `poly-sexpr-value` můžeme napsat i proceduru, která polynom převádí na proceduru jednoho argumentu, která představuje polynomiální funkci. Princip je stejný, jen výraz, který vznikne aplikací `poly-sexpr` „neobalíme“ `let`-blokem, ale λ-výrazem, a ten pak vyhodnotíme aplikací procedury `eval`. Definice této procedury by pak vypadala takto:

```
(define poly-sexpr-function
  (lambda (p)
    (eval (list 'lambda (list 'x)
                (poly-sexpr p 'x)))))
```

Na závěr této sekce naimplementujeme procedury symbolické derivace a symbolické integrace polynomů. Nejdříve tedy symbolická derivace realizovaná procedurou `poly-diff`:

```
(define poly-diff
  (lambda (p)
    (let ((result
           (let diff ((p (cdr p))
                      (n 1))
             (if (null? p)
                 p
                 (cons (* (car p) n)
                       (diff (cdr p) (+ n 1)))))))
      (if (null? result)
          the-zero-poly
          result))))
```

V kódu definujeme přes pojmenovaný `let` rekurzivní proceduru `diff`. Jejím prvním argumentem je seznam, druhým je pak čítač, ve kterém si procedura pamatuje kolikátá její aplikace právě probíhá. Toto číslo vlastně souhlasí s aktuálně zpracovávaným prvkem seznamu reprezentujícího polynom a tedy také se stupněm odpovídajícího členu tohoto polynomu. Výsledkem aplikace procedury `diff` na číselný seznam je seznam, jehož prvky jsou prvky původního seznamu vynásobené jejich pozicemi (tentokrát výjimečně počítaných od 1). Této proceduře ale nepředáváme přímo derivovaný polynom, ale jeho reprezentaci bez prvního prvku (odebrání prvního prvku z reprezentace polynomu, znamená odebrání konstantního členu tohoto polynomu a snížení stupně ostatních prvků). Výsledek aplikace procedury `diff` na takový seznam, je-li neprázdný, odpovídá reprezentaci polynomu, který je derivací původního. Pokud je vrácen prázdný seznam (což se stane v případě, že jsme derivovali konstantní polynom), vracíme nulový polynom. Viz aplikace této procedury:

```
(poly-sexpr (poly-diff '(5)) 'x)         ⟹ 0
(poly-sexpr (poly-diff '(5 2)) 'x)       ⟹ 2
(poly-sexpr (poly-diff '(5 2 4)) 'x)     ⟹ (+ 2 (* 8 x))
(poly-sexpr (poly-diff '(0 0 0 -7)) 'x)  ⟹ (* -21 x x)
(poly-sexpr (poly-diff '(5 2 4 -7)) 'x)  ⟹ (+ 2 (* 8 x) (* -21 x x))
```

Poslední proceduru, kterou napíšeme v této sekci, je procedura na symbolickou integraci polynomu. V jejím těle definujeme proceduru `int`. Ta prochází seznam podobně jako procedura `diff`, jen místo seznamu násobků prvků původního seznamu jejich pozicemi (bráno od 1), vrací seznam podílů prvků a jejich pozic. Jako konstantní člen pak přidáváme pro jednoduchost nulu. Následuje definice této procedury:

```
(define poly-int
  (lambda (p)
    (let ((result
           (let int ((p p)
                     (n 1))
             (if (null? p)
                 p
                 (cons (/ (car p) n)
                       (int (cdr p) (+ n 1)))))))
      (if (zero-poly? result)
          the-zero-poly
          (cons 0 result)))))
```

Viz aplikace procedury `poly-int`:

```
(poly-sexpr (poly-int '(5)) 'x)         ⟹ (* 5 x)
(poly-sexpr (poly-int '(5 2)) 'x)       ⟹ (+ (* 5 x) (* x x))
(poly-sexpr (poly-int '(5 2 4)) 'x)     ⟹ (+ (* 5 x) (* x x) (* 4/3 x x x))
(poly-sexpr (poly-int '(0 0 0 -7)) 'x)  ⟹ (* -7/4 x x x x)
(poly-sexpr (poly-int '(5 2 4 -7)) 'x)  ⟹ (+ (* 5 x) (* x x) (* 4/3 x x x)
                                             (* -7/4 x x x x))
```

Shrnutí

V této lekci jsme se zabývali rekurzivními procedurami a výpočetními procesy generovanými těmito procedurami. Nejprve jsme uvedli rekurzi a princip indukce přes přirozená čísla. Dále jsme se zabývali strukturální rekurzí a strukturální indukcí. Ukázali jsme také obecné principy rekurze a indukce. Principy rekurze a indukce byly v obou případech (rekurze přes čísla a přes seznamy) těsně vázané na strukturální vlastnosti prvků množin, se kterými jsme pracovali. Pro každé nezáporné celé číslo jsme vždy mohli udělat úvahu, že buď je číslo rovno nule, nebo je následníkem nějakého jiného čísla. Stejně tak každý seznam je buď prázdný, nebo vzniká z jiného seznamu připojením nového prvního prvku. Pomocí rekurze jsme definovali zobrazení mezi množinami čísel a seznamů. Pomocí indukce jsme prokazovali správnost a vlastnosti takto zavedených zobrazení. Dále jsme ze zabývali rekurzí a indukcí z pohledu programování. Představili jsme rekurzivní procedury jako procedury aplikující sebe sama. Každá rekurzivní procedura, kterou jsme uvažovali, měla svůj rekurzivní předpis (předpisy) a limitní podmínku (podmínky). Vytváření rekurzivních procedur je přímočaré, pokud si dobře uvědomíme, jak má vypadat limitní podmínka (vztahující se ke krajnímu případu) a jak vypadá rekurzivní předpis, který vyjadřuje redukci problému na problém (nebo několik problémů) o menším rozsahu. Dále jsme zjistili, že rekurzivní procedury generují různé výpočetní procesy. Zabývali jsme se třemi typy procesů generovaných rekurzivními procedurami, lineárně rekurzivním výpočetním procesem, lineárně iterativním výpočetním procesem a stromově rekurzivním výpočetním procesem. U každého z nich jsme uvedli několik metod stanovení složitosti výpočetního procesu. Pro efektivní provádění iterativního procesu jsme upravili aplikaci uživatelsky definovaných procedur vzhledem k aplikaci z koncových pozic. Ukázali jsme rovněž užitečný aparát, pomocí nějž je možné vytvářet jednorázově aplikovatelné rekurzivní procedury. Lekci jsme uzavřeli sérií příkladů rekurzivních procedur pracujících se seznamy.

Pojmy k zapamatování

- princip indukce, princip rekurze, aplikace sebe sama,
- rekurzivní definice, rekurzivní definice zobrazení,
- matematická indukce, indukce přes přirozená čísla,
- strukturálně jednodušší seznamy, strukturální indukce, strukturální rekurze,
- čistý funkcionální jazyk,
- rekurzivní procedura, rekurzivní aplikace procedury,
- limitní podmínka rekurze, předpis rekurze, série aplikací,
- fáze navíjení, odložený výpočet, deferred computation,
- dosažení limitní podmínky, fáze odvíjení,
- dekompozice, divide et impera, rozděl a panuj,
- rekurzivní výpočetní proces, lineární rekurzivní výpočetní proces,
- koncová pozice, koncová aplikace, koncová rekurze, koncově rekurzivní procedura,
- optimalizace na koncovou rekurzi, tail recursion optimization,
- degenerovaná fáze odvíjení, lineární iterativní výpočetní proces, cyklus,
- lineárně rekurzivní procedura, iterativní procedura,
- čítač, střadač,
- jednorázová aplikace,
- simulace navíjení a odvíjení pomocí zásobníku,
- stromově rekurzivní výpočetní proces.

Nově představené prvky jazyka Scheme

- speciální forma: pojmenovaný `let`.

Kontrolní otázky

1. *K čemu slouží princip rekurze?*
2. *K čemu byste použili indukci?*
3. *Jak lze dokázat správnost rekurzivních definic?*
4. *Jaký je rozdíl mezi matematickou indukcí a strukturální indukcí?*
5. *Co to znamená, že jeden seznam je strukturálně jednodušší než jiný seznam?*
6. *Je Scheme čistý funkcionální jazyk?*
7. *Co jsou to rekurzivní procedury?*
8. *Co říká princip „divide et impera“?*
9. *Jaký je rozdíl mezi rekurzivními procedurami a procedurami, se kterými jsme pracovali v předchozích lekcích?*
10. *Jaký je rozdíl mezi rekurzivní procedurou a rekurzivním výpočetním procesem?*
11. *Jaké typy rekurzivních výpočetních procesů znáte?*
12. *Co je charakteristické pro lineárně rekurzivní výpočetní proces?*
13. *Jak je definována koncová pozice?*
14. *Co je to koncová aplikace a koncově rekurzivní procedura?*
15. *Jak je ve většině programovacích jazyků realizována iterace?*
16. *Má každý lineárně iterativní výpočetní proces lineární časovou složitost?*
17. *Co jsou a k čemu slouží čítače?*
18. *Co jsou a k čemu slouží střadače?*
19. *V jakých fázích probíhají jednotlivé rekurzivní výpočetní procesy?*
20. *Z jakého důvodu je v rekurzivních procedurách přítomná limitní podmínka?*
21. *Co máme na mysli pod pojmem jednorázová aplikace rekurzivní procedury?*

22. *Jakým způsobem lze vždy nahradit rekurzivní proceduru iterativní procedurou?*
23. *U kterých výpočetních procesů roste exponenciálně počet prostředí vzniklých během aplikace?*
24. *Který z rekurzivních výpočetních procesů je možné snadno „zastavit“ a „rozběhnout“?*

Cvičení

1. Naprogramujte proceduru `perrin` jednoho argumentu $\langle n \rangle$, která vrací $\langle n \rangle$-tý člen P_n Perinovy posloupnosti. Perinova posloupnost je definována následujícím předpisem:
$$P_n = \begin{cases} 3 & \text{pokud } n = 0, \\ 0 & \text{pokud } n = 1, \\ 2 & \text{pokud } n = 2, \\ P_{n-2} + P_{n-3} & \text{jinak.} \end{cases}$$
Proceduru implementujte
   (a) jako rekurzivní proceduru
   (b) jako iterativní proceduru
   (c) pomocí pojmenovaného `let`
2. Napište rekurzivní proceduru vracející aritmetický průměr čísel v seznamu.
3. Napište rekurzivní verzi procedury `list-indices`, kterou jsme definovali v programu 6.5.
4. Implementujte Euklidův algoritmus na výpočet největšího společného dělitele.
5. Naprogramuje proceduru `reverse` pomocí pojmenovaného `let`.
6. Doplňte sadu procedur pro práci s polynomy ze sekce 8.6 o:
   (a) predikát `poly?`, který zjišťuje, zda je jeho argument reprezentací seznamu
   (b) konstruktor `poly-roots` libovolného počtu argumentů vytvářející polynom podle výčtu všech jeho kořenů.
   (c) proceduru `poly-gcd`, která počítá největší společný dělitel dvou polynomů, a která bude implementaci Euklidova algoritmu na polygomech.

Úkoly k textu

1. Zamyslete se, proč první argument speciální formy `if` není v koncové pozici. Co by nefungovalo?
2. Určete složitosti procedur uvedených v sekci 8.6.
3. Napište proceduru `pascal`, která přijímá dva číselné argumenty $\langle x \rangle$, $\langle y \rangle$ a vrací prvek pascalova trojúhelníka na pozici ($\langle x \rangle$, $\langle y \rangle$). Viz příklady volání:
   `(pascal 0 0)` $\Longmapsto$1 `(pascal 5 1)` $\Longmapsto$5 `(pascal 5 3)` $\Longmapsto$10 `(pascal 7 3)` $\Longmapsto$35
   Určete složitost vašeho řešení.

Řešení ke cvičením

1.
```
(define perrin
  (lambda (n)
    (cond ((= n 0) 3)
          ((= n 1) 0)
          ((= n 2) 2)
          (else (+ (perrin (- n 2))
                   (perrin (- n 3)))))))
```

(a)
```scheme
(define perrin
  (lambda (n)
    (define iter
      (lambda (a b c i)
        (if (<= i 0)
            a
            (iter b c (+ a b) (- i 1)))))
    (iter 3 0 2 n)))
```

(b)
```scheme
(define perrin
  (lambda (n)
    (let iter ((a 3) (b 0) (c 2) (i n))
      (if (<= i 0)
          a
          (iter b c (+ a b) (- i 1))))))
```

2.
```scheme
(define arit-means
  (lambda numbers
    (if (null? numbers)
        #f
        (let iter ((l numbers)
                   (accum 0)
                   (nr 0))
          (if (null? l)
              (/ accum nr)
              (iter (cdr l) (+ accum (car l)) (+ nr 1)))))))
```

3.
```scheme
(define list-indices
  (lambda (l elem)
    (let find-occurrs
        ((l l)
         (i 0))
      (cond ((null? l) '())
            ((equal? elem (car l)) (cons i (find-occurrs (cdr l) (+ i 1))))
            (else (find-occurrs (cdr l) (+ i 1)))))))
```

4.
```scheme
(define gcd
  (lambda (a b)
    (if (= b 0)
        a
        (gcd b (modulo a b)))))
```

5.
```scheme
(define reverse
  (lambda (l)
    (let iter ((l l)
               (accum '()))
      (if (null? l)
          accum
          (iter (cdr l) (cons (car l) accum))))))
```

6. (a)
```scheme
(define poly?
  (lambda (p)
    (or (constant-poly? p)
        (and (pair? p)
             (let poly-test ((p p)
                             (last-zero? #f))
               (if (null? p)
                   (not last-zero?)
```

```
                        (if (and (pair? p) (number? (car p)))
                            (poly-test (cdr p) (= (car p) 0))
                            #f)))))))
```

(b)
```
(define poly-roots
  (lambda roots
    (if (null? roots)
        the-unit-poly
        (poly* (poly (- (car roots)) 1)
               (apply poly-roots (cdr roots))))))
```

(c)
```
(define poly-gcd
  (lambda (p1 p2)
    (let ((m (poly-modulo p1 p2)))
      (if (zero-poly? m)
          p2
          (poly-gcd p2 m)))))
```

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` umožňující 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 66 a 67 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 „větve“ 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.

Naším 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 ⟨argument1⟩ ⟨argument2⟩ ··· ⟨argumentn⟩))
```

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)))))
```

$\cdots$ *výrazy* $\cdots$
```
(append2 '(1 2 3) '(a b c))
```
$\cdots$ *výrazy* $\cdots$)

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

$$
\begin{array}{l}
\texttt{(letrec ((}\langle symbol_1\rangle\ \langle element_1\rangle\texttt{)} \\
\qquad\qquad\texttt{(}\langle symbol_2\rangle\ \langle element_2\rangle\texttt{)} \\
\qquad\qquad\ \ \vdots \\
\qquad\qquad\texttt{(}\langle symbol_n\rangle\ \langle element_n\rangle\texttt{))} \\
\quad\langle výraz_1\rangle \\
\quad\langle výraz_2\rangle \\
\quad\ \ \vdots \\
\quad\langle výraz_m\rangle\texttt{)},
\end{array}
$$

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

`letrec`-blok vyhodnocuje stejným způsobem jako výraz

$$
\begin{array}{l}
\texttt{(let ((}\langle symbol_1\rangle\ \mathit{undefined}\texttt{)} \\
\qquad\quad\texttt{(}\langle symbol_2\rangle\ \mathit{undefined}\texttt{)} \\
\qquad\quad\ \ \vdots \\
\qquad\quad\texttt{(}\langle symbol_n\rangle\ \mathit{undefined}\texttt{))} \\
\\
\quad\texttt{(define }\langle symbol_1\rangle\ \langle element_1\rangle\texttt{)} \\
\quad\texttt{(define }\langle symbol_2\rangle\ \langle element_2\rangle\texttt{)} \\
\quad\ \ \vdots \\
\quad\texttt{(define }\langle symbol_n\rangle\ \langle element_n\rangle\texttt{)} \\
\quad\langle výraz_1\rangle \\
\quad\langle výraz_2\rangle \\
\quad\ \ \vdots \\
\quad\langle výraz_m\rangle\texttt{)},
\end{array}
$$

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^5RS 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 $\langle element_1 \rangle, \langle element_2 \rangle, \ldots \langle element_n \rangle$ 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 9.2. 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 184 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 vlastností, 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. Implementuje některou proceduru provádějící hloubkovou rekurzi na seznamech (například `depth-count`, `depth-find` nebo `depth-accum`) pomocí y-kombinátoru. Existuje nějaký principiální rozdíl při použití y-kombinátoru pro hloubkovou a běžnou rekurzi? Vysvětlete proč ano či proč ne.

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
```
(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
```
(define depth-map
  (lambda (f l)
    (depth-accum list f l)))
```

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

Lekce 10: Kombinatorika na seznamech, reprezentace stromů a množin

Obsah lekce: Tato lekce obsahuje pokročilejší příklady sloužící k procvičení práce s hierarchickými daty. V první části lekce se budeme zabývat reprezentací stromů pomocí seznamů a jejich prohledáváním do hloubky a do šířky. Dále ukážeme reprezentaci množin pomocí uspořádaných seznamů a pomocí binárních vyhledávacích stromů. Závěr lekce je věnován kombinatorice na seznamech.

Klíčová slova: kombinatorika na seznamech, reprezentace množin, stromy.

10.1 Reprezentace n-árních stromů

První sada příkladů se týká n-árních stromů (dále v této sekci budeme používat pojmenování *strom*). Nejdříve popíšeme reprezentaci těchto struktur, poté nadefinujeme příslušný konstruktor a selektory. V závěru sekce pak budeme realizovat algoritmy pro průchod stromem do hloubky a do šířky.

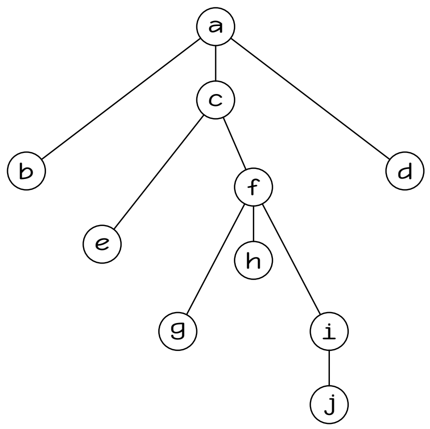
Za strom budeme považovat jakýkoli seznam, který, pokud je neprázdný, má za prvky ocasu další stromy. Hlavu seznamu přitom nazýváme *hodnota* a prvky ocasu nazýváme *podstromy* nebo též *větve stromu*. *Prvky stromu* rozumíme hodnotu stromu a prvky jeho větví. Stromy, které nemají podstromy nebo jsou všechny jejich podstromy prázdné, nazýváme *listy*.

Například strom nakreslený na obrázku 10.1 je reprezentován seznamem ve tvaru

```
(a (b) (c (e) (f (g) (h) (i (j)))) (d)).
```

Větve tohoto stromu jsou reprezentovány seznamy `(b)`, `(c (e) (f (g) (h) (i (j))))` a `(d)`. Hodnota stromu je `a`. Prvky stromu jsou `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i` a `j`.

Obrázek 10.1. Příklad n-árního stromu



Následuje definice konstruktoru a selektorů pro uvažovanou reprezentaci stromů:

```
(define make-tree
  (lambda (val . subtrees)
    (cons val subtrees)))

(define get-val car)

(define get-branches
  (lambda (x)
    (if (null? x) '() (cdr x))))
```

Dále uvádíme predikát `tree?` zjišťující zda je jeho argument reprezentace stromu. Predikát tedy zjistí, jestli se jedná o seznam a v případě že ano, zjistí jestli jsou jeho prvky (vyjma prvního) též reprezentace stromu. Implementace je velmi přímočará, využíváme v ní predikátu `forall`, který jsme zavedli v lekci 6.

```
(define tree?
  (lambda (x)
    (and (list? x)
         (forall tree? (cdr x)))))
```

Další procedury, které naimplementujeme v této sekci, jsou selektory vracející konkrétní větve. Přesněji selektor `left-branch` vracející k-tou větev zleva a selektor `right-branch` vracející k-tou větev zprava. Přitom budeme chtít, aby pro k, které bude záporné, nebo věší nebo rovno počtu větví vracel selektor prázdný strom (prázdný seznam). Následuje definice selektorů:

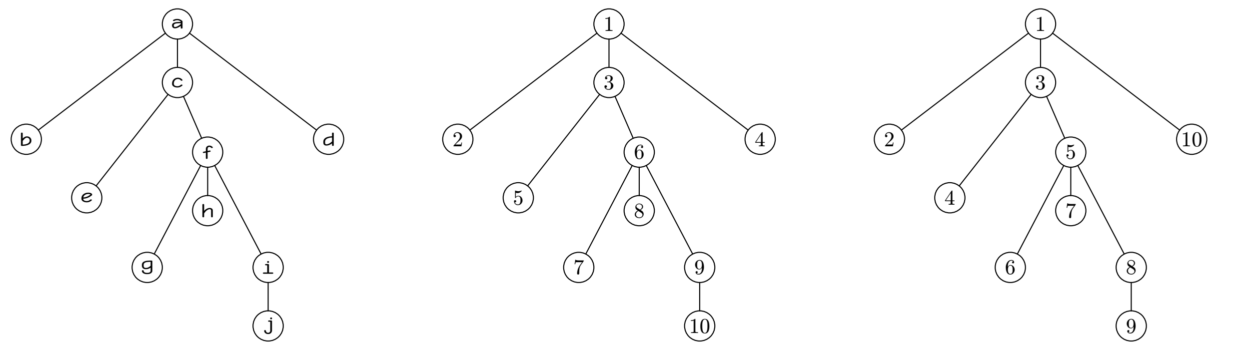
```
(define left-branch
  (lambda (tree k)
    (if (or (< k 0) (null? tree))
        '()
        (let iter ((branches (cdr tree))
                   (k k))
          (cond ((null? branches) '())
                ((= k 0) (car branches))
                (else (iter (cdr branches) (- k 1))))))))
```

Procedura `left-branch` nejprve ověří, zda nebylo zadáno záporné číslo a zda zadaný strom není prázdný. Pokud ano, vrací prázdný strom. V opačném případě pomocí iterativní procedury `iter`, postupně „odjímá“ podstromy, dokud nedojde na konec seznamu podstromů – pak vrací prázdný strom – nebo dokud nenajde požadovaný podstrom. Proceduru `right-branch` pak lze vytvořit jednoduše pomocí procedury `left-branch`. Viz její následující definici:

```
(define right-branch
  (lambda (tree k)
    (left-branch tree (- (length tree) k 2))))
```

Nyní budeme realizovat algoritmy průchodu stromu do hloubky a průchodu stromu do hloubky. Průchodem stromu přitom myslíme proceduru, která postupně „zpracuje“ všechny prvky stromu. Pod pojmem „zpracování prvků“ budeme mít na mysli konstrukci seznamu prvků v podstromech v tom pořadí, v jakém je navštěvujeme. Dále uvedené algoritmy se liší právě v pořadí, v jakém prvky stromu zpracovávají. Při průchodu stromu do hloubky je strom zpracováván „po větvích“, při průchodu stromu do šířky je strom zpracováván „po patrech“. Viz ilustraci na obrázku 10.2. Procedura `dfs` uvedená v programu 10.1 pro

Obrázek 10.2. Ukázka průchodu do šířky (uprostřed) a do hloubky (vpravo).



průchod stromu do hloubky vrací prázdný seznam pro prázdný strom. Jinak vezme větve stromu a projde je do hloubky (aplikací sebe sama), výsledné seznamy spojí a přidá k nim hodnotu stromu.

Procedura `bfs` uvedená v programu 10.2 pro průchod stromu do šířky využívá pomocnou proceduru `aux`, která si ve svém argumentu pamatuje seznam stromů určených k průchodu. Pokud je tento seznam prázdný, vrací prázdný seznam. Jinak vezme hodnoty za všech stromů v seznamu. V dalším kroku je pak tento seznam tvořen podstromy těchto stromů. Tím jakoby „sestoupíme o jedno patro“.

Program 10.1. Prohledávání stromu do hloubky.

```
(define dfs
  (lambda (tree)
    (if (null? tree) '()
        (cons (get-val tree)
              (apply append (map dfs (get-branches tree)))))))
```

Program 10.2. Prohledávání stromu do šířky.

```
(define  bfs
  (lambda (tree)
    (let aux ((tree (list tree)))
         (if (null? tree) '()
             (append (map get-val tree)
                     (aux (apply append (map get-branches tree))))))))
```

10.2 Reprezentace množin pomocí uspořádaných seznamů

V sekci 6.5 jsme ukázali, jak se dají reprezentovat konečné množiny pomocí seznamů neobsahujících vícenásobné výskyty prvků (duplicity). V této části předvedeme jinou možnost reprezentace množin. Za množiny budeme považovat seznamy bez vícenásobných výskytů prvků, které jsou navíc uspořádané. Tedy například seznam `(5 8 2 4 9)`, který byl v sekci 6.5 platnou reprezentací množiny, není uspořádaný (podle uspořádání daným predikátem `<=`), a proto není reprezentací množiny pro tuto sekci. Platnou reprezentací této množiny by pak byl seznam `(2 4 5 8 9)`.

Toto zpřísnění reprezentace nám umožní psát efektivnější procedury, než jsou ty ze sekce 6.5. Na druhou stranu ale musíme mít dané nějaké rozumné lineární uspořádání (rozumné ve smyslu složitosti – ověření, zda-li je jeden prvek menší nebo roven než druhý musí mít přijatelnou časovou a prostorovou složitost). Také kód procedur pro tuto reprezentaci bude mnohem méně přehledný, což může vést ke vzniku chyb při jejich implementaci. Proto je nutné provádět důsledné testování.

U následujících procedur (`in?`, `cons-set`, `union` a `inter`) budeme uvažovat jen číselné množiny a uspořádání dané predikátem `<=`. U každé z těchto procedur nejdříve naznačíme, jak pracují, uvedeme jejich definici s popisem kódu a příklady aplikace.

První procedura – predikát `in?` – bude využívat uspořádání seznamu k rychlejšímu zjištění nepřítomnosti daného prvku v množině. Tato procedura bude postupně procházet přes prvky seznamu dokud nenarazí na prvek, který je shodný s hledaným prvkem, dokud neprojde všechny prvky, nebo dokud nenarazí na prvek, který je větší než ten hledaný. Právě poslední případ zastavení je přínosem zpřísnění reprezentace množiny. Viz následující definici predikátu `in?`:

```
(define in?
  (lambda (x set)
    (cond ((or (null? set) (< x (car set))) #f)
          ((= x (car set)) #t)
          (else (in? x (cdr set))))))
```

V této rekurzívní proceduře zastavujeme rekurzi při zjištění, že je seznam prázdný, nebo že je první prvek seznamu větší než hledaný prvek. V tom případě je jisté, že hledaný prvek v množině není, a tak je vrácena nepravda `#f`. Dále zastavujeme rekurzi, pokud je první prvek seznamu shodný s hledaným prvek, a tehdy je vrácena pravda `#t`. Zastavení rekurze je v kódu zahrnuto v prvních dvou větvích speciální formy `cond`.

V případě, že není splněna ani jedna z těchto limitních podmínek, je predikát `in` rekurzivně aplikován na seznam bez prvního prvku. Následují příklady aplikace:

```
(in? 3 '())          ⟹  #f
(in? 3 '(1 2))       ⟹  #f
(in? 3 '(1 2 3 4))   ⟹  #t
(in? 3 '(1 2 4))     ⟹  #f
```

Druhou procedurou je konstruktor množiny `cons-set`. Ten má zatřídit zadaný element do dané množiny, pokud už tam tento element není. Ta bude procházet seznam reprezentující množinu podobným způsobem jako predikát `in?`. Přínosem zpřísnění reprezentace je opětně to, že můžeme dříve zjistit nepřítomnost prvku. Implementace této procedury se bude jen málo lišit od implementace predikátu `in?`. Viz definici:

```
(define cons-set
   (lambda (x set)
      (cond ((null? set) (list x))
            ((= x (car set)) set)
            ((<= (car set) x) (cons (car set) (cons-set x (cdr set))))
            (else (cons x set)))))
```

V těle této procedury zastavujeme rekurzi v případě, že je množina prázdná a v tom případě vracíme jednoprvkový seznam obsahující přidaný prvek. Dále zastavujeme rekurzi při zjištění, že je první prvek seznamu shodný s přidávaným elementem a tehdy vracíme původní seznam bez jakékoli změny, protože je v něm přidávaný prvek už obsažen. Poslední limitní podmínkou je skutečnost, že první prvek seznamu je větší, než přidávaný element. Pokud je tato podmínka splněna, znamená to, že jsme našli místo, kam má být prvek přidán. Pokud není splněna žádná z limitních podmínek, pak rozkládáme problém na přidání prvku do seznamu (konstruktorem `cons`), a přidávání prvku do menší množiny (tedy množiny reprezentované strukturálně jednodušším seznamem). Následují příklady použití procedury `cons-set`.

```
(cons-set 3 '())          ⟹  ()
(cons-set 3 '(1 2))       ⟹  (1 2 3)
(cons-set 3 '(1 2 3 4))   ⟹  (1 2 3 4)
(cons-set 3 '(1 2 4))     ⟹  (1 2 3 4)
```

Další dvě procedury uvedené v této sekci představují množinové operace sjednocení a průniku. Protože se jedná o složitější procedury, ukážeme nejdříve to, jak pracují na konkrétním případě a pak až se zaměříme na konkrétní implementaci.

U operace sjednocení množin zpracováváme současně dva seznamy reprezentující množiny. S těmito seznamy provádíme slévání, podobně jako tomu bylo u procedury `merge` v sekci 8.5. Jediným rozdílem je to, že sledujeme i možnost, že by při průběžném zpracování seznamů nastala skutečnost, že tyto seznamy mají stejný první prvek. V tom případě zařadíme tento prvek do výsledného seznamu a odebereme jej *z obou* seznamů. Tím zamezíme vzniku duplicit ve výsledném seznamu, které by nastaly při běžném slévání. Viz příklad:

seznam A	*seznam B*	*prvky zařazené do $A \cup B$*
`(2 4 6 7)`	`(1 2 3 4 5)`	`1`
`(2 4 6 7)`	`(2 3 4 5)`	`2`
`(4 6 7)`	`(3 4 5)`	`3`
`(4 6 7)`	`(4 5)`	`4`
`(6 7)`	`(5)`	`5`
`(6 7)`	`()`	`6 7`, *výsledek* $A \cup B$: `(1 2 3 4 5 6 7)`

Následuje definice procedury `union`:

```
(define union
   (lambda (set-A set-B)
      (cond ((null? set-A) set-B)
            ((null? set-B) set-A)
            ((= (car set-A) (car set-B))
```

```
             (cons (car set-A)
                   (union (cdr set-A) (cdr set-B))))
            ((<= (car set-A) (car set-B))
             (cons (car set-A)
                   (union (cdr set-A) set-B)))
            (else (cons (car set-B) (union set-A (cdr set-B)))))))
```

Limitní podmínkou je test na prázdnost alespoň jednoho ze seznamů. Pokud je tato podmínka splněna, vracíme druhý ze seznamů (ten, který není prázdný). To je vyjádřeno prvními dvěma větvemi speciální formy `cond`. Jinak rozlišujeme tyto možnosti:

- Oba seznamy začínají stejným prvkem. Pak tento společný prvek přidáme na začátek sjednocení množin bez tohoto prvku. To je zahrnuto v třetí větvi formy `cond`.

- Jeden seznam začíná menším prvkem než druhý seznam. V tom případě sjednotíme seznamy bez tohoto nejmenšího prvku, a tento prvek přidáme do výsledku. Tato možnost je vyjádřena posledními dvěma větvemi formy `cond`.

Viz příklady aplikace:

```
(union '() '())                     ⟾  ()
(union '() '(1 2 3 4 5))            ⟾  (1 2 3 4 5)
(union '(2 4 6 7) '())              ⟾  (2 4 6 7)
(union '(2 4 6 7) '(1 2 3 4 5))     ⟾  (1 2 3 4 5 6 7)
```

Podobně pracuje i procedura `inter` na výpočet průniku množin. Prvky do výsledné množiny ale zařazujeme jen v případě, že se vyskytují v obou seznamech. Takové se nám při průběžném zpracování dostanou na začátek seznamu. Viz následující příklad:

seznam A	*seznam B*	*prvky zařazené do $A \cap B$*
(2 4 6 7)	(1 2 3 4 5)	
(2 4 6 7)	(2 3 4 5)	2
(4 6 7)	(3 4 5)	
(4 6 7)	(4 5)	4
(6 7)	(5)	
(6 7)	()	*výsledek $A \cap B$*: (2 4)

Následuje implementace procedury `inter`:

```
(define inter
   (lambda (set-A set-B)
      (cond ((or (null? set-B) (null? set-A)) '())
            ((= (car set-A) (car set-B))
             (cons (car set-A)
                   (inter (cdr set-A) (cdr set-B))))
            ((<= (car set-A) (car set-B)) (inter (cdr set-A) set-B))
            (else (inter set-A (cdr set-B))))))
```

Procedura `inter` zastavuje rekurzi v případě, že alespoň jeden ze seznamů je prázdný. V takovém případě je výsledkem prázdná množina. Pokud oba seznamy začínají stejným elementem, přidáme tento společný element do průniku množin bez tohoto elementu. Jinak pouze „odjímáme“ nejmenší prvky ze seznamů.

Viz příklady aplikace:

```
(inter '() '())                     ⟾  ()
(inter '() '(1 2 3 4 5))            ⟾  ()
(inter '(2 4 6 7) '())              ⟾  ()
(inter '(2 4 6 7) '(1 2 3 4 5))     ⟾  (2 4)
```

10.3 Reprezentace množin pomocí binárních vyhledávacích stromů

V této sekci ukážeme další možnost, jak reprezentovat množiny. Jedná se o reprezentaci pomocí binárních vyhledávacích stromů. Binárním vyhledávacím stromem vzhledem k uspořádání $\leq$ rozumíme:

- prázdný strom
- strom, který má právě dva podstromy. Tyto podstromy nazýváme *pravý podstrom* a *levý podstrom*. Jsou-li tyto podstromy neprázdné, musí platit, že hodnota levého podstromu je menší vzhledem k uspořádání $\leq$ než hodnota stromu a hodnota pravého podstromu je větší vzhledem k uspořádání $\leq$ než hodnota stromu.

Binární vyhledávací stromy budeme reprezentovat stejně jako n-ární stromy uvedené v sekci 10.1. Tomu budou odpovídat i konstruktory a selektory pro binární strom.

```
(define make-tree
  (lambda (value left right)
    (list value left right)))

(define make-leaf
  (lambda (value)
    (make-tree value '() '())))

(define tree-value car)
(define tree-left cadr)
(define tree-right caddr)
```

Pomocí těchto struktur budeme reprezentovat množiny. Napíšeme predikát `in?` zjišťující, zda je zadaný prvek v množině, a konstruktor `cons-tree` přidávající prvek do množiny. Při implementaci těchto procedur budeme uvažovat stromy reprezentující množiny čísel a uspořádání dané predikátem `<=`.

Procedura `in?` zastavuje rekurzi v případě, že zkoumaný strom je prázdný – pak vrací nepravdu – nebo v případě, že hodnota stromu je shodná s hledaným prvkem – pak vrací pravdu. Pokud ani jedna z těchto limitních podmínek podmínek není splněna, pokračuje hledáním prvku v levém nebo v pravém podstromu, to podle porovnání hledaného prvku a hodnoty stromu.

```
(define in?
  (lambda (x tree)
    (cond ((null? tree) #f)
          ((= x (tree-value tree)) #t)
          ((< x (tree-value tree)) (in? x (tree-left tree)))
          (else (in? x (tree-right tree))))))
```

Procedura konstrukce množiny reprezentované stromem také zastavuje v případě, že je strom prázdný. Tehdy vrací jednoprvkový strom vytvořený procedurou `make-leaf`. Zastavuje také, pokud je hodnota stromu rovna přidávanému prvku. V tom případě je prvek už ve stromu přítomen a ten se jeho přidáním nezmění, a je tedy vrácen původní strom. Jinak je vytvořen nový strom do jehož levého nebo pravého podstromu (v závislosti na porovnání vkládaného prvku a hodnoty stromu) je vložen přidávaný prvek (použitím procedury `cons-tree`).

```
(define cons-tree
  (lambda (x tree)
    (cond ((null? tree) (make-leaf x))
          ((= x (tree-value tree)) tree)
          ((< x (tree-value tree))
           (make-tree (tree-value tree)
                      (cons-tree x (tree-left tree))
                      (tree-right tree)))
          (else
```

Program 10.3. Výpočet potenční množiny.

```
(define power-set
  (lambda (set)
    (if (null? set)
        '(())
        (append (map (lambda (x)
                       (cons (car set) x))
                     (power-set (cdr set)))
                (power-set (cdr set))))))
```

Program 10.4. Efektivnější výpočet potenční množiny.

```
(define power-set
  (lambda (set)
    (if (null? set)
        '(())
        (let ((power-rest (power-set (cdr set))))
          (append (map (lambda (x)
                         (cons (car set) x))
                       power-rest)
                  power-rest)))))
```

```
(make-tree (tree-value tree)
           (tree-left tree)
           (cons-tree x (tree-right tree)))))))
```

10.4 Kombinatorika na seznamech

V této poslední sekci předvedeme několik příkladů na výpočet kombinatorických úloh nad seznamy. Budeme například hledat všechny podmnožiny množiny reprezentované seznamem bez vícenásobných výskytů prvků, všechny permutace, všechny k-prvkové kombinace nebo k-prvkové kombinace prvků zadané množiny.

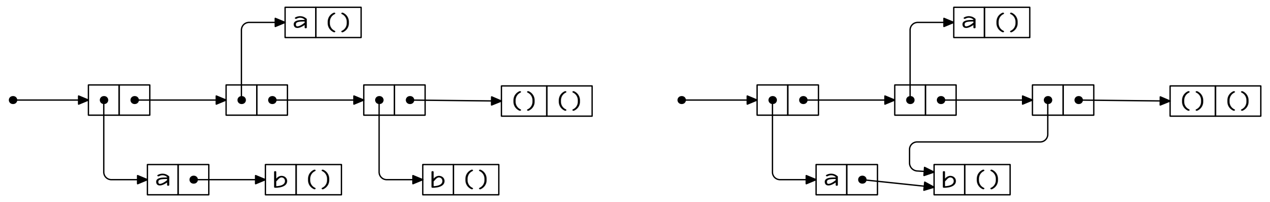
Výpočet potenční množiny 2^A (množiny všech podmnožin množiny A) vytvoříme podle tohoto rekurzivního přepisu:

$$2^A = \begin{cases} \{\emptyset\} & \text{pokud } A = \emptyset, \\ 2^{\{a_2,\dots\}} \cup \bigcup \{\{a_1\} \cup B | B \in 2^{\{a_2,\dots\}}\} & \text{pokud } A = \{a_1, a_2, \dots\} \text{ je neprázdná.} \end{cases}$$

Pokud zadaná množina $A = \{a_1, a_2, \dots\}$ není prázdná, vypočteme nejdříve potenční množinu její podmnožiny $\{a_2, \dots\}$. Do výsledné množiny pak zařadíme všechny prvky této potenční množiny „jakoby dvakrát". Jednou to budou přímo tyto množiny, podruhé tyto množiny s přidaným prvkem a_1. Program 10.3 je přímým přepisem právě uvedeného popisu. Všimněte si, že se v programu 10.3 vyskytuje dvakrát výraz `(power-set (cdr set))`. Výpočet můžeme zefektivnit pomocí lokální vazby tak, jak je uvedeno v programu 10.4. Důsledkem, že potenční množinu podmnožiny počítáme jen jednou, bude nejen kratší čas výpočtu, ale také struktura výsledného seznamu bude vypadat jinak (viz obrázek 10.3, vlevo je výsledek pro původní proceduru, vpravo je výsledek pro novou proceduru). Z čistě funkcionálního pohledu na seznamy, který jsme doposud uplatňovali, je to však principiálně jedno.

Dále se budeme zabývat výpočtem všech permutací n prvků. Budeme to provádět tak, že postupně projdeme všech n prvků seznamu. Pro každý vytvoříme seznam permutací majících na začátku právě tento prvek. Ty

Obrázek 10.3. Výsledek aplikace staré a vylepšené verze `power-set` pro argument `(a b)`.



Program 10.5. Výpočet všech permutací prvků množiny.

```
(define permutation
  (lambda (l)
    (if (null? l)
        '(())
        (let perm
            ((base l)
             (p-set (cdr l)))
          (if (null? base)
              '()
              (append
               (map (lambda (x)
                      (cons (car base) x))
                    (permutation p-set))
               (perm (cdr base)
                     (cdr (append p-set (list (car base))))))))))))
```

vytvoříme tak, ze najdeme (rekurzí) permutace zbývajících prvků a na jejich začátek přidáme uvažovaný prvek. Tyto seznamy spojíme. Následující schéma zachycuje hledání všech permutací prvků seznamu `(1 2 3)`:

uvažovaný prvek:	`1`		`2`		`3`	
zbývající prvky:	`(2 3)`		`(3 1)`		`(1 2)`	
permutace zbývajících prvků:	`(2 3)`	`(3 2)`	`(3 1)`	`(1 3)`	`(1 2)`	`(2 1)`
permutace začínající daným prvkem:	`(1 2 3)`	`(1 3 2)`	`(2 3 1)`	`(2 1 3)`	`(3 1 2)`	`(3 2 1)`

Program 10.5 obsahuje definici právě popsané procedury. Jednotlivé prvky zadaného seznamu `l` jsou procházeny pomocnou procedurou `perm`. Tato procedura si v argumentu `p-set` pamatuje seznam zbylých prvků (to jest prvků, které jsou různé od právě zpracovávaného). Z prvků tohoto seznamu vytvoří permutace (pomocí rekurzivního volání procedury `permutation`), připojí na jejich začátek aktuální prvek, a seznam takto vytvořených permutací připojí k permutacím začínajícím ostatními prvky (procedura `perm` pokračuje zpracováním dalšího prvku).

Můžeme také generovat přímo konkrétní permutace, aniž bychom museli vytvářet všechny. A to přes jejich index. K tomu použijeme takzvanou *faktoradickou soustavu čísel*. Faktoradická číselná soustava je soustava o proměnlivém základu založeném na faktoriálu:

základ:	`7`	`6`	`5`	`4`	`3`	`2`	`1`	`0`
hodnota pozice:	`7!`	`6!`	`5!`	`4!`	`3!`	`2!`	`1!`	`0!`
v dekadické soustavě:	`5040`	`720`	`120`	`24`	`6`	`2`	`1`	`1`

Prvních šest čísel faktoradické soustavy je ukázáno v prostředním sloupci tabulky 10.4.

Vztah faktoradických čísel a permutací n prvků je velice jednoduchý. My teď popíšeme, jak z faktoradického

Obrázek 10.4. Faktoradická čísla a permutace.

0	$0_2 0_1 0_0$	(0,1,2)
1	$0_2 1_1 0_0$	(0,2,1)
2	$1_2 0_1 0_0$	(1,0,2)
3	$1_2 1_1 0_0$	(1,2,0)
4	$2_2 0_1 0_0$	(2,0,1)
5	$2_2 1_1 0_0$	(2,1,0)

čísla získat permutaci:

Z n prvků vytvoříme tzv. seznam kandidátů. Číslice, která je v n-ciferné reprezentaci (případně doplněné počátečními nulami) faktoradického čísla nejvíce vlevo označuje index prvku v seznamu kandidátů, který dosadíme na nejlevější neobsazené pozici v n-tici permutace. Vybraný prvek odebereme ze seznamu kandidátů a z reprezentace faktoradického čísla odebereme nejlevější číslici. Zbytek pozic n-tice permutace vyplníme jako permutaci zbývajících kandidátů podle zkráceného faktoradického čísla.

Příklad 10.1. Předveďme popsaný postup na následujícím příkladě

$2_2 0_1 0_0$	$\{a, b, c\}$	$[c, \_, \_]$
$0_1 0_0$	$\{a, b\}$	$[c, a, \_]$
0_0	$\{b\}$	$[c, a, b]$

Permutace prvků a, b, c odpovídající faktoradickému číslu $2_2 0_1 0_0$ je permutace $[c, a, b]$. Další příklady jsou v tabulce 10.4.

Při implementaci budeme využívat procedur, které jsme naprogramovali dříve. Konkrétně se jedná o proceduru `fak` (viz programy 7.12, 8.3, 8.5 a 8.6 na stranách 187, 206, 208 a 212 a proceduru `remove` z programu6.3 na straně 147, respektive její efektivní implementaci).

Vyjdeme z jednoduché procedury na převod čísla `dec` v dekadické soustavě na číslo v `bas`-adické soustavě:

```
(define prevod
  (lambda (dec bas)
    (let iter ((dec dec)
               (res '()))
      (if (= dec 0) res
          (iter (quotient dec bas) (cons (modulo dec bas) res))))))
```

Nyní uděláme dvě úpravy (viz program 10.6):

1. na místo pevně zadaného základu bude procedura přijímat proceduru `basf`, která pro index (pozici, počítanou zleva) vrací její základ. To proto, abychom mohli převádět čísla z dekadické soustavy na soustavy s proměnlivým základem.

2. procedura bude vracet přesně `l` čísel (výsledek bude ořezán, nebo budou doplněny počáteční nuly). Číslo `l` bude předáváno proceduře jako další parametr.

Program 10.6 obsahuje definici procedury pro převod `prevod` s právě popsanými úpravami a definici procedury `index->perm` realizující výše popsané hledání permutace náležející k danému indexu.

Viz příklady aplikace:

```
(index->perm 0 '(1 2 3))  ⟹  (1 2 3)
(index->perm 1 '(1 2 3))  ⟹  (1 3 2)
(index->perm 2 '(1 2 3))  ⟹  (2 1 3)
(index->perm 3 '(1 2 3))  ⟹  (2 3 1)
(index->perm 4 '(1 2 3))  ⟹  (3 1 2)
(index->perm 5 '(1 2 3))  ⟹  (3 2 1)
```

Program 10.6. Výpočet permutace prvků množiny pomocí faktoradických čísel.

```
(define prevod
  (lambda (dec basf l)
    (let iter ((dec dec)
               (res '())
               (i 1))
      (if (> i l) res
          (let ((bas (basf i)))
            (iter (quotient dec bas)
                  (cons (modulo dec bas) res)
                  (+ i 1)))))))

(define index->perm
  (lambda (i set)
    (let perm ((set set)
               (fnum (prevod i n! (length set))))
      (if (null? set) '()
          (cons (list-ref set (car fnum))
                (perm (remove set (car fnum))
                      (cdr fnum))))))))
```

Program 10.7. Výpočet všech kombinací prvků množiny.

```
(define combination
  (lambda (k set)
    (cond ((null? set) '())
          ((= k 0) '(()))
          ((= k 1) (map list set))
          (else (append (map (lambda (x)
                               (cons (car set) x))
                             (combination (- k 1) (cdr set)))
                        (combination k (cdr set)))))))
```

Procedura `combination` přijímá dva argumenty – číslo k a množinu S – a vrací seznam všech k-prvkových kombinací prvků množiny S (to jest seznam všech různých k-tic, jejichž složky jsou vzájemně různé prvky množiny S). Pracuje tak, že prochází seznam reprezentující množinu S a pro každý prvek vytvoří dva seznamy kombinací:

1. kombinace obsahující tento prvek. Ty najde tak, že vytvoří $k-1$-prvkové kombinace doposud neuvažovaných prvků a pak do nich přidá právě zpracovávaný prvek.

2. kombinace neobsahující tento prvek. Hledají se tedy k-prvkové kombinace doposud neuvažovaných prvků.

Tyto seznamy se pak spojí dohromady. Viz program 10.7

Malou úpravou kódu z programu 10.7 dostaneme proceduru, která vrací seznam kombinací s opakováním. Program 10.8 se liší od definice procedury `combination-dup` jen v tom detailu, že při rekurzivním volání procedury, jehož výsledkem má být zbytek kombinace, neodebíráme už dosazený prvek z množiny kombinovaných prvků.

Program 10.8. Výpočet všech kombinací s opakováním.

```
(define combination-dup
  (lambda (k set)
    (cond ((null? set) '())
          ((= k 0) '(()))
          (else (append (map (lambda (x)
                                (cons (car set) x))
                              (combination-dup (- k 1) set))
                         (combination-dup k (cdr set)))))))
```

Shrnutí

Tato lekce obsahuje pokročilejší příklady sloužící k procvičení práce s hierarchickými daty. V první části lekce se budeme zabývat reprezentací stromů pomocí seznamů a jejich prohledáváním do hloubky a do šířky. Dále ukážeme reprezentaci množin pomocí uspořádaných seznamů a pomocí binárních vyhledávacích stromů. Závěr lekce je věnován kombinatorice na seznamech.

Lekce 11: Kvazikvotování a manipulace se symbolickými výrazy

Obsah lekce: Tato lekce obsahuje několik klasických příkladů zpracování symbolických dat se kterými se lze setkat i v jiné v literatuře. Nejprve se budeme zabývat kvazikvotováním, což je obecnější metoda kvotování umožňující programátorům snadněji definovat některé seznamy. Dále se budeme zabývat následujícími okruhy problémů: zjednodušování aritmetických výrazů, symbolická derivace, konverze mezi symbolickými výrazy zapsanými v prefixové, infixové, postfixové a polské bezzávorkové notaci. Nakonec ukážeme metodu vyhodnocování výrazů v polské bezzávorkové notaci.

Klíčová slova: kvazikvotování, infixová notace, postfixová notace, polská bezzávorková notace.

11.1 Kvazikvotování

V sekci 4.5 jsme ukázali speciální formu `quote`, která vracela svůj argument v nevyhodnocené podobě. Nyní představíme zobecnění této speciální formy – `quasiquote`. Speciální forma `quasiquote`, v porovnání se speciální formou `quote`, umožňuje navíc určit podvýrazy jejího argumentu, které budou vyhodnoceny běžným způsobem. Bez tohoto určení funguje speciální forma `quasiquote` stejným způsobem jako speciální forma `quote`. Viz následující příklady:

```
(quasiquote 15)              ⟾ 15
(quasiquote symbol)          ⟾ symbol
(quasiquote (x . 3))         ⟾ (x . 3)
(quasiquote (1 2 (3 4) 5))   ⟾ (1 2 (3 4) 5)
```

Zatímco speciální forma `quote` „slepě“ vrací svůj argument, speciální forma `quasiquote` argument projde a vyhledá v něm podvýrazy, které jsou jednoprvkové seznamy začínající symbolem `unquote`. Tyto podvýrazy jsou nahrazeny vyhodnocením jejich druhého prvku. Nejlépe to bude vidět na příkladech:

```
(define x 3)
(quasiquote (unquote x))                  ⟾ 3
(quasiquote (unquote (+ 1 2)))            ⟾ 3
(quasiquote (1 x (unquote (+ 1 x 2))))    ⟾ (1 x 6)
```

Příklad 11.1. (a) Někdy se symbol `unquote` nemusí nacházet viditelně na začátku seznamu. V následující ukázce tento symbol není prvním prvkem seznamu `(1 unquote (+ 1 2))`, ale až druhým. Tento seznam je tečkový pár `(1 . (unquote (+ 1 2)))`, jehož druhým prvkem je seznam začínající symbolem `unquote`:

```
(quasiquote (1 unquote (+ 1 2)))   ⟾ (1 . 3)
```

je totéž jako

```
(quasiquote (1 . (unquote (+ 1 2))))   ⟾ (1 . 3).
```

(b) Symbol `unquote` musí být prvním prvkem seznamu, nikoli jen páru. Navíc seznam, jehož první prvek je symbol `unquote`, musí obsahovat právě jeden další prvek. Následující výrazy neobsahují správné použití symbolu `unquote` v argumentu speciální formy `quasiquote`:

```
(quasiquote (1 (unquote . 2)))
(quasiquote (1 (unquote)))
(quasiquote (1 (unquote 2 3)))
```

Výsledky vyhodnocení těchto výrazů standard jazyka Scheme R^{5}RS neurčuje. Naopak přímo říká, že v takových případech dochází k nepředpovídatelnému chování. V našem abstraktním interpretu necháme vyhodnocení takových výrazů skončit chybou „CHYBA: Nekorektní argument speciální formy `quasiquote`“.

Další symbol se speciálním významem pro speciální formu `quasiquote` je symbol `unquote-splicing`. Pokud argument speciální formy `quasiquote` obsahuje seznam ve tvaru `(unquote-splicing` ⟨*arg*⟩`)`, je vyhodnocen výraz ⟨*arg*⟩. Předpokládá se, že se tento argument vyhodnotí na seznam. Celý výraz

`(unquote-splicing` ⟨*arg*⟩`)`

je pak nahrazen tímto výsledným seznamem bez otevírající a uzavírající závorky. Viz příklad:

```
(quasiquote (1 (unquote (build-list 3 +)) #t))                ⟹  (1 (0 1 2) #t)
(quasiquote (1 (unquote-splicing (build-list 3 +)) #t))       ⟹  (1 0 1 2 #t)
(quasiquote (1 (unquote (map - (list 1 2 3))) #t))            ⟹  (1 (-1 -2 -3) #t)
(quasiquote (1 (unquote-splicing (map - (list 1 2 3))) #t))   ⟹  (1 -1 -2 -3 #t)
```

Odstraněny jsou přitom jen vnější závorky. Pokud seznam vzniklý vyhodnocením výrazu ⟨*arg*⟩ má jako prvky další seznamy, nebudou k nim náležející výrazy odstraněny. Tedy například:

```
(quasiquote (1 2 (unquote-splicing (map list (list 1 2 3))))) ⟹ (1 2 (1) (2) (3))
```

V sekci 4.5 jsme zavedli použití uvozovky `'` jako syntaktický cukr pro speciální formu `quote`. Také pro speciální formu `quasiquote` je k dispozici zkrácený zápis. Namísto výrazu `(quasiquote` ⟨*arg*⟩`)` můžeme ekvivalentně psát jen `` ` ``⟨*arg*⟩. Dále namísto výrazů `(unquote` ⟨*arg*⟩`)` a `(unquote-splicing` ⟨*arg*⟩`)` je možno psát pouze `,`⟨*arg*⟩ a `,@`⟨*arg*⟩ (v tomto pořadí). Například:

```
`1                          ⟹  1
`symbol                     ⟹  symbol
`(x . 3)                    ⟹  (x . 3)
`(1 2 ,(+ 2 3) #f)          ⟹  (1 2 5 #f)
`(1 2 ,(build-list 3 +) #f) ⟹  (1 2 (0 1 2) #f)
`(1 2 ,@(build-list 3 +) #f) ⟹ (1 2 0 1 2 #f)
```

Podvýrazy argumentu speciální formy `quasiquote` mohou přirozeně opětně obsahovat použití této speciální formy:

```
`(1 2 ,(list `(3)))                   ⟹  (1 2 ((3)))
`(1 2 ,(map (lambda (x) `(,x)) `(1 2 3))) ⟹ (1 2 ((1) (2) (3)))
```

Nyní ukážeme několik příkladů s použitím kvotování a kvazikvotování. Všechny jsou uvedeny s použitím syntaktického cukru, i po jeho odstranění. Speciální forma `quote` svůj argument vrací bez jakékoli změny, bez ohledu na to, jestli se uvnitř něho vyskytují symboly `unquote` a `unquote`.

```
'`(1 2 3) = (quote (quasiquote (1 2 3)))
  ⟹ (quasiquote (1 2 3))
`'(1 2 3) = (quasiquote (quote (1 2 3)))
  ⟹ (quote (1 2 3))
``(1 2 3) = (quasiquote (quasiquote (1 2 3)))
  ⟹ (quasiquote (1 2 3))
'`(1 2 ,(+ 1 2)) = (quote (quasiquote (1 2 (unquote (+ 1 2)))))
  ⟹ (quasiquote (1 2 (unquote (+ 1 2))))
`'(1 2 ,(+ 1 2)) = (quasiquote (quote (1 2 (unquote (+ 1 2)))))
  ⟹ (quote (1 2 3))
```

Upozorněme, že na symboly `unquote` a `unquote-splicing` nejsou navázány ani procedury, ani speciální formy. Tím, že na tyto symboly navážeme nějakou hodnotu, nezměníme vyhodnocování speciální formy `quasiquote`. Viz následující příklady:

```
unquote                                       ⟹  „CHYBA: Symbol unquote nemá vazbu.“
(define unquote (lambda (x) (+ 1 x)))
`(,10) = (quasiquote (unquote 10))            ⟹  (10)
`(,,10) = (quasiquote (unquote (unquote 10))) ⟹  (11)
```

Standard jazyka Scheme R5RS neříká jakým způsobem má interpret zacházet se symboly `unquote` a `unquote-splicing`. Některé interprety, jako je například DrScheme, se odlišují od našeho abstraktního interpretru v aplikaci speciální formy `quasiquote`. Výše popsaným způsobem se chovají například interprety Bigloo nebo Guile. Například v DrScheme po navázání hodnoty na `unquote` přestane správně fungovat vyhodnocování kvazikvotovaných výrazů.

11.2 Zjednodušování aritmetických výrazů

V této a následujících sekcích uvedeme procedury pro zpracování symbolických výrazů. V této sekci půjde o vytvoření procedury `simplify`, která zjednodušuje aritmetické výrazy.

Aritmetické výrazy budeme v tomto případě reprezentovat S-výrazy které jsou čísla, symboly nebo dvouprvkové nebo tříprvkové seznamy, jejichž první prvek je jeden ze symbolů `+`, `-`, `*` a `/` označujících operaci a zbylé prvky jsou aritmetické výrazy. Čísla přitom reprezentují číselné hodnoty, které označují, symboly reprezentují „proměnné“, a výrazy ve tvaru seznamu reprezentují provedení operace sčítání, odčítání, násobení a dělení nad danými operandy. Například seznam `(+ (* 3 x) 2)` reprezentuje aritmetický výraz ve tvaru $3x + 2$.

Kód procedury `simplify` uvedený v programu 11.1 nejdříve popíšeme a poté se budeme věnovat úpravám tohoto řešení.

V těle procedury `simplify` definujeme několik pomocných procedur. První z nich je predikát `==`, který zjistí, zdali jsou oba jeho argumenty čísla a zdali jsou si rovna (při porovnání predikátem `=`). Dále definujeme proceduru `div-by-zero`, pomocí niž bude vrácen příznak dělení nulou a to v případě, že by měl být zjednodušován výraz, ve kterém se dělí nulou. V těle jsou dále uvedeny pomocné procedury zjednodušující určitý typ výrazu. V programu 11.1 máme uvedenu jen jednu z nich – proceduru `simplify-add` zajišťující zjednodušování výrazů, které jsou ve tvaru součtu. Další procedury, které by se v kódu nacházely na místě výpustky, jsou uvedeny v programu 11.2. Jedná se o procedury `simplify-min`, `simplify-mul` a konečně `simplify-div` zajišťující zjednodušování výrazů ve tvaru rozdílu, součinu a podílu.

Procedura `simplify` zjednodušuje výraz podle jeho charakteru. Jedná-li se o číslo nebo o symbol, jedná se o atomický výraz a zjednodušit už nejde. Pokud se jedná o seznam začínající symbolem operace, zjednoduší se jeho operandy rekurzivním voláním procedury `simplify`. V případě, že se po zjednodušení jedná o dvouprvkový seznam obsahující (mimo symbolu operace) číslo, je zjednodušením přímo výsledek vyhodnocení tohoto výrazu. Totéž platí pro tříprvkový seznam obsahující symbol operace a dvě čísla. V ostatních případech aplikujeme jednu ze zjednodušujících pomocných procedur (to jest procedur `simplify-add`,...,`simplify-div`) v závislosti na symbolu operace, kterým zjednodušovaný výraz začíná. Viz příklady aplikace:

```
(simplify '(+ 10))                          ⟹  10
(simplify '(+ 1 2))                         ⟹  3
(simplify '(+ (* 2 3) (+ y (+ 2 x))))       ⟹  (+ 6 (+ y (+ 2 x)))
(simplify '(- x 0))                         ⟹  x
(simplify '(- 0 x))                         ⟹  (- x)
(simplify '(/ 0 x))                         ⟹  0
(simplify '(/ x 0))                         ⟹  division-by-zero
(simplify '(/ x 1))                         ⟹  x
(simplify '(/ 1 x))                         ⟹  (/ x)
(simplify '(/ (+ 2 x) (* 2 0.5)))           ⟹  (+ 2 x)
```

Elegantnějšího řešení můžeme dosáhnout tak, že vytvoříme tabulku, ve které budeme mít uložené pomocné procedury na zjednodušování a k nim příslušné symboly operací. Implementovat bychom ji mohli například tak, jak je uvedeno v programu 11.3.

V proceduře `simplify` pak můžeme větvení `cond` zkrátit třeba takto:

```
(if (and (number? x) (number? y)) (eval expr)
    (apply (cdr (assoc op simplification-table)) expr))
```

Zde využíváme procedury `assoc`, kterou jsme doposud neuvedli. Tato procedura přijímá dva argumenty $\langle e \rangle$ a $\langle l \rangle$. První argument $\langle e \rangle$ je libovolný element a druhý argument $\langle l \rangle$ je seznam tečkových párů. Procedura vrací ten nejlevější tečkový pár ze seznamu $\langle l \rangle$, jehož první prvek je shodný s elementem $\langle e \rangle$. Viz ilustrativní příklady použití `assoc`:

```
(define s '((a . 10) (b . (a b c)) (20 30) (b . 666)))
(assoc 'a s)  ⟹  (a . 10)
```

Program 11.1. Procedura pro zjednodušování aritmetických výrazů.

```
(define simplify
  (lambda (expr)
    ;; zjisti, zda-li jsou oba argumenty stejna cisla
    (define ==
      (lambda (x y)
        (and (number? x) (number? y) (= x y))))

    ;; osetreni deleni nulou
    (define div-by-zero
      (lambda () 'division-by-zero))

    ;; zjednoduseni pro pricitani
    (define simplify-add
      (lambda (op x y)
        (cond ((== x 0) y)                  zjednodušení využívající 0 + y = y
              ((== y 0) x)                  zjednodušení využívající x + 0 = 0
              ((equal? x y) `(* 2 ,x))      zjednodušení využívající x + x = 2 · x
              (else (list op x y)))))
      ⋮
    (cond
      ((number? expr) expr)
      ((symbol? expr) expr)
      ((and (list? expr) (member (car expr) '(+ * - /)))
       (let* ((op (car expr))
              (expr (map simplify expr)))
         (if (null? (cddr expr))
             (if (number? (cadr expr))
                 (eval expr)
                 expr)
             (let ((x (cadr expr))
                   (y (caddr expr)))
               (cond ((and (number? x) (number? y)) (eval expr))
                     ((equal? op '+) (simplify-add op x y))
                     ((equal? op '*) (simplify-mul op x y))
                     ((equal? op '-) (simplify-min op x y))
                     (else (simplify-div op x y))))))))))
```

Program 11.2. Interní definice v proceduře pro zjednodušování výrazů.

```
;; zjednoduseni pro nasobeni
(define simplify-mul
  (lambda (op x y)
    (cond ((== x 0) 0)              zjednodušení využívající 0 · y = 0
          ((== y 0) 0)              zjednodušení využívající x · 0 = 0
          ((== x 1) y)              zjednodušení využívající 1 · y = y
          ((== y 1) x)              zjednodušení využívající x · 1 = y
          (else (list op x y)))))

;; zjednoduseni pro odcitani
(define simplify-min
  (lambda (op x y)
    (cond ((== x 0) (list op y))    zjednodušení využívající 0 − y = −y
          ((== y 0) x)              zjednodušení využívající x − 0 = x
          ((equal? x y) 0)          zjednodušení využívající x − x = 0
          (else (list op x y)))))

;; zjednoduseni pro deleni
(define simplify-div
  (lambda (op x y)
    (cond ((== x 0) 0)              zjednodušení využívající 0/y = 0
          ((== x 1) (list op y))    zjednodušení využívající 1/y = y^-1
          ((== y 0) (div-by-zero))  dělení nulou je nedefinované
          ((== y 1) x)              zjednodušení využívající x/1 = x
          ((equal? x y) 1)          zjednodušení využívající x/x = 1
          (else (list op x y)))))
```

Program 11.3. Tabulka procedur pro zjednodušování výrazů.

```
(define simplification-table
  ;; pomocne procedury == a div-by-zero
  (let ((== (lambda (x y)
              (and (number? x) (number? y) (= x y))))
        (div-by-zero (lambda () 'division-by-zero)))

   ;; vlastni tabulka
   `((+ . ,(lambda (op x y)
             (cond ((== x 0) y)
                   ((== y 0) x)
                   ((equal? x y) `(* 2 ,x))
                   (else (list op x y)))))

     (* . ,(lambda (op x y)
             (cond ((== x 0) 0)
                   ((== y 0) 0)
                         ⋮
```

Program 11.4. Procedura pro vyhledávání v asociačním seznamu.

```
(define assoc
  (lambda (key alist)
    (cond ((null? alist) #f)
          ((equal? key (caar alist)) (car alist))
          (else (assoc key (cdr alist))))))
```

```
(assoc 'b s)  ⟹  (b a b c)
(assoc 20 s)  ⟹  (20 30)
(assoc 'c s)  ⟹  #f
```

Proceduru `assoc` by bylo jednoduché implementovat, jak ukazuje program 11.4.

Nyní se zaměříme na zobecnění procedury `simplify` tak, aby bylo možné zjednodušovat i aritmetické výrazy obsahující použití `+` a `*` pro n operandů. Při sčítání (násobení) vyfiltrujeme všechna čísla a sečteme (vynásobíme) je. Zbylých, to jest nečíselných prvků, vytvoříme seznam. To provedeme například následovně s využitím procedur `filter` a `remove` (viz například programy 6.2 a 6.3 na straně 147.)

```
(let ((value (apply (eval op) (filter number? (cdr expr))))
      (compound (remove number? (cdr expr))))
  ...)
```

Tyto hodnoty pak zpracujeme následující procedurou `simplify-addmul`:

```
(define simplify-addmul
  (lambda (op compound value)
    (cond ((and (equal? op '*) (= value 0)) 0)
          ((null? compound) value)
          ((= value (eval `(,op))) (if (null? (cdr compound))
                                       (car compound)
                                       (cons op compound)))
          (else `(,op ,value ,@compound)))))
```

Procedura `simplify-addmul` vrací nulu, pokud se jedná o operaci `*` a číselná hodnota navázaná na formální argument `value` je nulová. A vrací přímo číselnou hodnotu, pokud je seznam složených výrazů (navázaný na formální argument `compound`), prázdný. Pokud je číselná hodnota rovna neutrálnímu prvku příslušné operace, což zjistíme vyhodnocením výrazu `(= value (eval `(,op))`, je vrácen původní výraz bez číselné hodnoty, nebo složený výraz ze seznamu `compound`, pokud je tento seznam jednoprvkový. V ostatních případech už nelze výraz zjednodušit.

Výsledný kód s uvedenými dvěma změnami by pak vypadal tak jak je to uvedeno v programu 11.5.

Následují příklady aplikace:

```
(simplify '(+))                              ⟹  0
(simplify '(+ 10))                           ⟹  10
(simplify '(+ 1 2))                          ⟹  3
(simplify '(+ 1 2 x 3 y 5 6))                ⟹  (+ 17 x y)
(simplify '(+ (* 2 3) y (+ 2 x)))            ⟹  (+ 6 y (+ 2 x))
(simplify '(- (* 1 x) (+ 2 -3 x 1)))         ⟹  0
(simplify '(- x 0))                          ⟹  x
(simplify '(- 0 x))                          ⟹  (- x)
(simplify '(/ 0 x))                          ⟹  0
(simplify '(/ x 0))                          ⟹  division-by-zero
(simplify '(/ x 1))                          ⟹  x
(simplify '(/ 1 x))                          ⟹  (/ x)
```

Program 11.5. Vylepšená procedura pro zjednodušování aritmetických výrazů.

```
;; tabulka pro zjednodusovani
(define simplification-table
  (let ((== (lambda (x y)
              (and (number? x) (number? y) (= x y))))
        (div-by-zero (lambda () 'division-by-zero))
        (simplify+* (lambda (op . rest)
                      (let ((value (apply (eval op) (filter number? rest)))
                            (compound (remove number? rest)))
                        (simplify-addmul op compound value)))))

    `((+ . ,simplify+*)
      (* . ,simplify+*)

      (- . ,(lambda (op x y)
              (cond ((== x 0) (list op y))
                    ((== y 0) x)
                    ((equal? x y) 0)
                    (else (list op x y)))))

      (/ . ,(lambda (op x y)
              (cond ((== x 0) 0)
                    ((== x 1) (list op y))
                    ((== y 0) (div-by-zero))
                    ((== y 1) x)
                    ((equal? x y) 1)
                    (else (list op x y))))))))

;; vlastni procedura provadejici zjednodusovani
(define simplify
  (lambda (expr)
    (cond
      ((number? expr) expr)
      ((symbol? expr) expr)
      ((list? expr)
       (let ((op (car expr))
             (expr (map simplify expr)))
         (if (forall number? (cdr expr))
             (eval expr)
             (let ((simplifier (assoc op simplification-table)))
               (if simplifier
                   (apply (cdr simplifier) expr)
                   (error "No record for such operation"))))))
      (else (error "Incorrect input expression")))))
```

```
(simplify '(/ (+ 2 x) (+ 1 x (* 2 1/2))))  ⟹  1
```

Procedura `simplify` samozřejmě ještě není (zdaleka) dokonalá. Na závěr sekce ukážeme několik příkladů aplikace procedury `simplify`, které ukazují její nedostatky:

```
(simplify '(+ x (- x)))              ⟹  (+ x (- x))         ideální zjednodušení: 0
(simplify '(* x (/ 1 x)))            ⟹  (* x (/ x))         ideální zjednodušení: 1
(simplify '(+ 1 (+ 1 (+ 1 x))))      ⟹  (+ 1 (+ 1 (+ 1 x))  ideální zjednodušení: (+ x 3)
(simplify '(+ x x x x))              ⟹  (+ x x x x)         ideální zjednodušení: (* x 4)
```

11.3 Symbolická derivace

V této sekci se budeme zabývat realizací procedury `diff` na nalezení symbolické derivace aritmetického výrazu (podle dané proměnné). Pro jednoduchost se omezíme jen na aritmetické výrazy s binárními operacemi.

Jak na to tedy půjdeme: Proměnné budeme derivovat na číslo 1, konstanty na číslo 0, složitější výrazy pak podle jejího derivačního předpisu. Tento předpis budeme vybírat na základě operace, kterou výraz začíná. Předpis přitom budeme reprezentovat procedurou dvou argumentů, kterými budou operandy derivovaného výrazu. Procedura předpisu bude vracet výraz odpovídající derivaci výrazu. Například pro operaci $+$, to bude procedura, která vznikne vyhodnocením λ-výrazu

```
(lambda (x y) `(+ ,(derive x) ,(derive y))),
```

která formalizuje známý derivační vztah $(f+g)' = f' + g'$. Operace a k nim příslušné předpisy budeme uchovávat v tabulce obdobně jako v sekci 11.2. Tato tabulka bude opět reprezentovaná seznamem řádků. Těmito řádky budou tečkové páry, tabulka tedy bude organizována jako asociační seznam. Tento přístup umožňuje rozšiřovat proceduru pro derivování pouhým přidáváním řádků do této tabulky. Například kdybychom chtěli derivovat i výrazy obsahující podvýrazy ve tvaru `(sin x)`, stačilo by přidat do kódu řádek

```
(sin . ,(lambda (x) `(* (cos ,x) ,(derive x))),
```

což formalizuje vztah $(\sin f)' = (\cos f') \cdot f'$ (nezapomeňte, že f nemusí být pouze proměnná). Definice procedury `diff` je uvedena v programu 11.6.

Procedura `diff` tedy pro aritmetický výraz a proměnnou vrací jeho derivaci podle této proměnné. Viz příklady aplikace této procedury:

```
(diff 'x 'x)                     ⟹  1
(diff 'x 'y)                     ⟹  0
(diff '(+ x x) 'x)               ⟹  (+ 1 1)
(diff '(+ x x) 'y)               ⟹  (+ 0 0)
(diff '(* x y) 'x)               ⟹  (+ (* x 0) (* 1 y))
(diff '(+ (* x 2) (* x x)) 'x)   ⟹  (+ (+ (* x 0) (* 1 2)) (+ (* x 1) (* 1 x)))
(diff '(* x (* x x)) 'x)         ⟹  (+ (* x (+ (* x 1) (* 1 x))) (* 1 (* x x)))
```

Výsledný výraz můžeme zjednodušit pomocí procedury `simplify`, kterou jsme naspali v předchozí sekci.

```
(simplify (diff 'x 'x))                     ⟹  1
(simplify (diff 'x 'y))                     ⟹  0
(simplify (diff '(+ x x) 'x))               ⟹  2
(simplify (diff '(+ x x) 'y))               ⟹  0
(simplify (diff '(* x y) 'x))               ⟹  y
(simplify (diff '(+ (* x 2) (* x x)) 'x))   ⟹  (+ (+ x x) 2)
(simplify (diff '(* x (* x x)) 'x))         ⟹  (+ (* x (+ x x)) (* x x))
```

Poznámka 11.2. Všimněte si, že v těle procedury `diff` v programu 11.6 jsme definovali pomocnou proceduru `derive` v jejímž těle je použita tabulka pravidel. Na druhou stranu ale taky platí, že procedury, které se nacházejí v tabulce pravidel, používají pomocnou proceduru `derive`. To je ale zcela v pořádku

Program 11.6. Procedura pro symbolickou derivaci.

```
(define diff
  (lambda (expr var)

    (define variable?
      (lambda (expr)
        (equal? expr var)))

    (define constant?
      (lambda (expr)
        (or (number? expr)
            (and (symbol? expr)
                 (not (variable? expr))))))

    (define table
      `((+ . ,(lambda (x y) `(+ ,(derive x) ,(derive y))))
        (- . ,(lambda (x y) `(- ,(derive x) ,(derive y))))
        (* . ,(lambda (x y) `(+ (* ,x ,(derive y)) (* ,(derive x) ,y))))
        (/ . ,(lambda (x y) `(/ (- (* ,(derive x) ,y) (* ,x ,(derive y)))
                                (* ,y ,y))))))

    (define derive
      (lambda (expr)
        (cond ((variable? expr) 1)
              ((constant? expr) 0)
              (else (apply (cdr (assoc (car expr) table))
                           (cdr expr))))))

    (derive expr)))
```

a kód je naprosto funkční. Někoho by možná mohlo zmást, že v těle procedury máme několik procedur, které se na sebe vzájemně odkazují. Mohla by se tedy nabídnout otázka, zdali je to vůbec přípustné a jestli k vůli tomu při aplikaci `diff` nemůže dojít k chybě. K chybě nedojde proto, že veškeré aplikace procedur `derive` a procedur v tabulce pravidel probíhají až v okamžiku, kdy jsou v lokálním prostředí procedury `diff` zavedeny všechny lokální vazby symbolů `variable?`, `constant?`, `table` a `derive`. To je důsledkem toho, že při vzniku procedury se nevyhodnocuje její tělo. Jinými slovy, během vzniku procedury vůbec nevadí, že v jejím těle je uveden výraz obsahující symbol, který je (zatím) bez vazby. Podstatné je, že vazby budou existovat v okamžiku její aplikace, což je v případě programu 11.6 splněno.

Pomocí procedury `diff` můžeme napsat proceduru vyššího řádu, která vrací proceduru jako proceduru jednoho argumentu reprezentující matematickou funkci. Provedeme to tak, že zkonstruujeme λ-výraz, jehož tělem je výsledek aplikace procedur `simplify` a `diff` na zadaný výraz. Ten vyhodnotíme procedurou `eval`.

```
(define diff-procedure
  (lambda (expr var)
    (eval `(lambda (,var)
             ,(simplify (diff expr var))))))
```

Procedura `diff-procedure` na rozdíl od procedury `derivace`, kterou jsme implementovali v programu 2.7 na straně 62 vrací přesnou reprezentaci derivované funkce. Viz následující příklad použití:

```
(define f (diff-procedure '(* x x) 'x))
(f 1)    ⟹  2
(f 10)   ⟹  20
(f 15)   ⟹  30
```

Na druhou stranu procedura `derivace` pro přibližnou derivaci z programu 2.7 je univerzálnější v tom, že jí můžeme předat jako argument rovnou proceduru. Procedura `diff` pracuje nad symbolickými výrazy a pokud by byla derivované funkce zastoupena procedurou obsahující operace, které nejsou v tabulce pravidel pro derivaci, museli bychom tabulku rozšířit.

> Programovací jazyky FORTRAN a LISP vznikly prakticky současně (FORTRAN je o něco málo starší). Přesto je filosofie obou jazyků naprosto odlišná a dá se říct, že oba dva jazyky předurčily vývoj mnoha dalších rodin programovacích jazyků. Zatímco FORTRAN byl programovací jazyk sloužící pro numerické výpočty, tedy jediná data, která bylo možné ve FORTRANu zpracovávat, byla čísla, LISP byl již od počátku jazykem zpracovávajícím *symbolická data*, viz [MC60]. Za symbolická data považujeme data reprezentující symbolické výrazy, tedy čísla, symboly a seznamy. To je výrazný posun proti pouhému „numerickému zpracování čísel“. Vznik LISPu byl motivován potřebou mít k dispozici programovací jazyk, ve kterém bude možné pracovat s procedurami reprezentujícími rekurzivní funkce nad symbolickými daty. Za zmínku stojí, že motivační příklad se symbolickými derivacemi byl představen již v prvním publikovaném dokumentu o jazyku LISP, kterým je článek [MC60].

11.4 Infixová, postfixová a bezzávorková notace

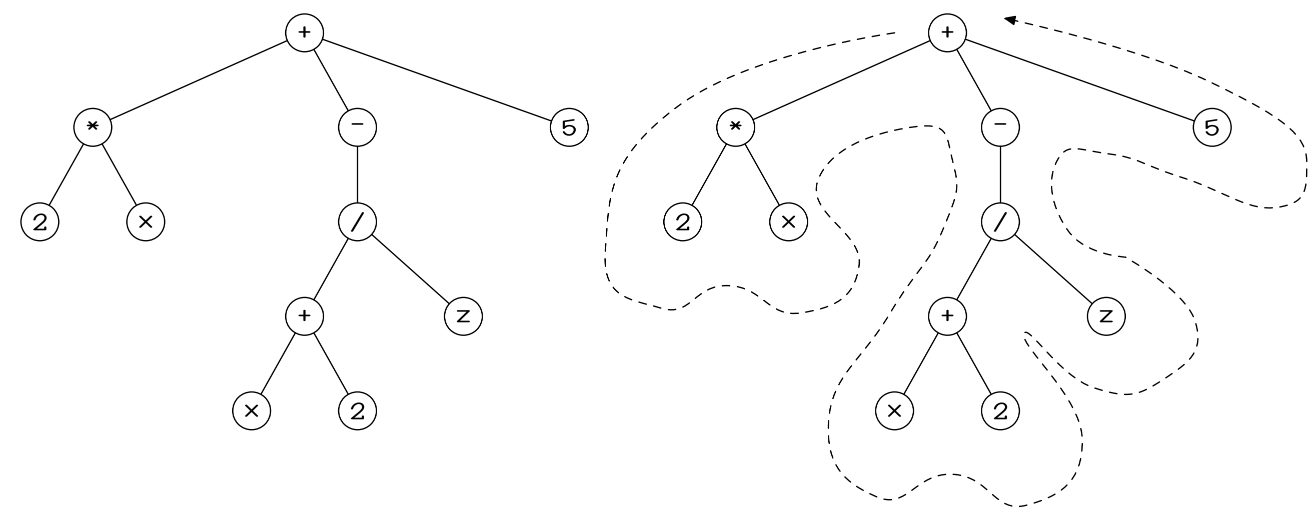
V této sekci se budeme zabývat problémem převodu výrazů zapsaných v různých notacích. Již v první lekci jsme konstatovali, že jazyk Scheme používá prefixovou notaci, ale běžně se také používají jiné notace. Nejčastější je infixová, méně časté jsou postfixová notace a postfixová bezzávorková notace (tak zvaná polská reverzní notace), viz sekci 1.2. Při psaní praktických aplikací se můžeme setkat s problémem převodu výrazů mezi těmito notacemi. Tím se tedy budeme zabývat nyní.

První a nejdůležitější je si vždy uvědomit strukturu výrazu. Máme-li například symbolický výraz

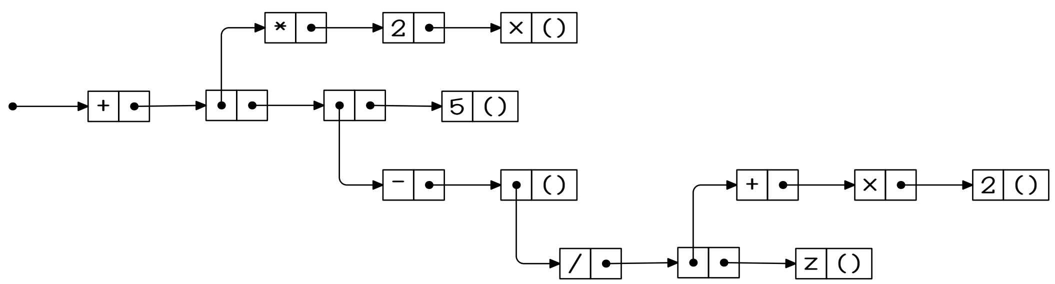
```
(+ (* 2 x) (- (/ (+ x 2) z)) 5),
```

pak jej z hlediska operací a operandů můžeme zachytit uzlově ohodnoceným stromem tak, jak je tomu v obrázku 11.1 vlevo. Z hlediska datové reprezentace je seznam `(+ (* 2 x) (- (/ (+ x 2) z)) 5)`

Obrázek 11.1. Struktura výrazu (+ (* 2 x) (- (/ (+ x 2) z)) 5)



Obrázek 11.2. Fyzická struktura seznamu (+ (* 2 x) (- (/ (+ x 2) z)) 5)



reprezentován strukturou párů, která je nakreslena v obrázku 11.2. Problém převodu prefixového výrazu do ostatních notací je vlastně problémem průchodu stromem naznačeným v obrázku 11.1, který je fyzicky reprezentovaný strukturou z obrázku 11.2. Pokud budeme tento strom procházet do hloubky, to jest ve směru tak, jak to naznačuje šípka v obrázku 11.1 vpravo, projdeme postupně všechny uzly reprezentující „podvýrazy" přitom pro daný výraz je vždy jako první zpracován jeho nejlevější podvýraz. Jelikož se prefixová, infixová a postfixová notace liší jen pozicí (symbolu) operace, převod můžeme provést jednoduše tak, že během průchodu strukturou budeme konstruovat její „kopii" až na to, že v každém nelistovém uzlu budeme vždy dávat symbol pro operaci na požadované místo: tedy buď před, mezi, nebo za všechny operandy. To v podstatě v grafové terminologii odpovídá *pre-order, in-order* a *post-order* zpracování nelistových uzlů stromu.

V programu 11.7 je uvedena procedura `prefix->postfix` pro převod výrazu v prefixové notaci do postfixové notace. Při splnění prvních tří podmínek v `cond`-výrazu je převod triviální. V případě, že vstupní výraz je seznam, je nejprve rekurzivně aplikována procedura `prefix->postfix`, což způsobí převod všech podvýrazů do postfixové notace. Z výsledku jejich převodu je vytvořen výsledný výraz připojením symbolu pro operaci za poslední z něj. Viz příklady použití procedury:

```
(prefix->postfix '(* 2 x))             ⟹  (2 x *)
(prefix->postfix '(- (/ (+ x 2) z)))   ⟹  (((x 2 +) z /) -)
```

Upravením procedury pro převod do postfixové notace můžeme vytvřit proceduru `prefix->polish` pro převod prefixových výrazů do polské reverzní bezzávorkové notace tak, jak je to ukázáno v programu 11.8. Jedinou změnou oproti myšlence použité v programu `prefix->postfix` je to, že nyní kromě připojení symbolu operace až za poselní operand navíc odstraňujeme z výrazu veškeré závorky (kromě vnějších). To v podstatě odpovídá operaci linearlizace, kterou jsme probrali v lekci 9, viz sekci 9.5. Vskutku, bezzávorko-

Program 11.7. Procedura pro převod výrazů do postfixové notace.

```
(define prefix->postfix
  (lambda (S-expr)
    (cond ((number? S-expr) S-expr)
          ((symbol? S-expr) S-expr)
          ((null? S-expr) S-expr)
          ((list? S-expr)
           (append (map prefix->postfix (cdr S-expr))
                   (list (car S-expr))))
          (else "incorrect expression"))))
```

Program 11.8. Procedura pro převod do postfixové (polské) bezzávorkové notace.

```
(define prefix->polish
  (lambda (S-expr)
    (cond ((number? S-expr) (list S-expr))
          ((symbol? S-expr) (list S-expr))
          ((null? S-expr) (list S-expr))
          ((list? S-expr)
           (append (apply append (map prefix->polish (cdr S-expr)))
                   (list (car S-expr))))
          (else "incorrect expression"))))
```

vou notaci bychom z prefixové mohli získat pouhou linearizací příslušného seznamu. Na druhou stranu, procedura z programu 11.8 pracuje efektivněji (jednoprůchodově). Viz příklady použití procedury:

```
(prefix->polish 'x)                   ⟾  (x)
(prefix->polish 20)                   ⟾  (20)
(prefix->polish '(* 2 x))             ⟾  (2 x *)
(prefix->polish '(- (/ (+ x 2) z)))   ⟾  (x 2 + z / -)
(prefix->polish '(* 2 (+ 3 5)))       ⟾  (2 3 5 + *)
(prefix->polish '(* (+ 2 3) 5))       ⟾  (2 3 + 5 *)
```

Na předchozím příkladu si povšimněte dvou věcí. Výsledný výraz je vždy ve tvaru jediného lineárního seznamu a to i v případě, že vstupní výraz byl atom, to je rozdíl oproti závorkované postfixové notaci. Na posledních dvou řádcích předchozího příkladu je vidět, jak se do bezzávorkové notace promítá jiné uzávorkování dvou aritmetických výrazů.

Analogicky, jako jsme naprogramovali procedury pro převod z prefixové do postfixových notací, bychom mohli naprogramovat i procedury pro opačné převody. Jejich naprogramování je více méně rutinní a necháme jej na laskavém čtenáři. Mnohem obtížnější je však manipulace s infixovými výrazy. Nejprve uveďme proceduru pro převod prefixových výrazů do infixu. Procedura `prefix->infix` vykonávající tento převod je uvedena v programu 11.9. Převod čísel a symbolů do infixu je opět triviální. V případě seznamů musíme ošetřit několik situací. Ve vnitřním `cond`-výrazu nejprve ošetřujeme situaci, kdy je převáděný seznam jednoprvkový, to jest obsahuje pouze symbol pro operaci a žádné operandy. V tomto případě je převod opět triviální. Dále ošetřujeme situaci, kdy máme vstupní operaci s jedním operandem. Ten převedeme do infixu, ale symbol pro operaci píšeme pořád před něj, to odpovídá například hodnotám -2, $\frac{1}{2}$ reprezentovaným seznamy `(- 2)` a `(/ 2)`, a podobně. V ostatních případech musíme za každý operand vložit symbol pro operaci. Navíc do sebe vnoříme závorky tak, aby každá operace měla pouze dva operandy. K tomuto účelu můžeme s výhodnou použít proceduru `foldl`, viz lekci 7 věnující se akumulaci. Následující příklady ukazují použití procedury.

Program 11.9. Procedura pro převod výrazů do infixové notace.

```
(define prefix->infix
  (lambda (S-expr)
    (cond ((null? S-expr) S-expr)
          ((number? S-expr) S-expr)
          ((symbol? S-expr) S-expr)
          ((pair? S-expr)
           (let* ((op (car S-expr))
                  (tail (cdr S-expr))
                  (len (length tail)))
             (cond ((= len 0) S-expr)
                   ((= len 1) (list op (prefix->infix (car tail))))
                   (else (foldl (lambda (expr collected)
                                  (list collected op expr))
                                (prefix->infix (car tail))
                                (map prefix->infix (cdr tail)))))))
          (else "incorrect expression"))))
```

```
(prefix->infix '(-))                   ⟹  (-)
(prefix->infix '(- 2))                 ⟹  (- 2)
(prefix->infix '(- 2 3))               ⟹  (2 - 3)
(prefix->infix '(- 2 3 4))             ⟹  ((2 - 3) - 4)
(prefix->infix '(- (/ (+ x 2) z)))     ⟹  (- ((x + 2) / z))
```

Převod výrazu z obrázku 11.1 by dopadl takto:

```
(prefix->infix '(+ (* 2 x) (- (/ (+ x 2) z)) 5))
  ⟹  (((2 * x) + (- ((x + 2) / z))) + 5)
```

Poznámka 11.3. Podotkněme, že procedura `prefix->infix` není zdaleka dokonalá, nerespektuje například přirozenou asociativitu operací. Všechny výrazy ve tvaru $x_1 \circledcirc \cdots \circledcirc x_n$ chápe jako výrazy tvaru

$$(\cdots((x_1 \circledcirc x_2) \circledcirc x_3) \cdots \circledcirc x_n),$$

což nemusí být vždy žádoucí. V některých případech vyžadujeme závorkování opačně, někdy jej lze úplně vynechat. Úpravy procedury ponecháváme na čtenáři.

Na závěr sekce poznamenejme, že pokud jsme řekli, že převod do infixové notace je obtížnější než převod na postfixovou notaci, který je více méně rutinní, pak musíme říct, že *převod z infixové notace* (třeba do prefixové) je ještě výrazně složitější. Zvlášť je tomu tak právě v případě, kdy se vynechávají závorky z důvodu *asociativity operací*, nebo když chceme do našich úvah započíst pravidla pro *priority operací* (pravidla typu „násobení“ má přednost před „sčítáním“, tedy infixový výraz `(2 * 3 + 5)` znamená v prefixové notaci `(+ (* 2 3) 5)` a nikoliv `(* 2 (+ 3 5))`). Problém rozpoznávání struktury výrazů je obecně obtížný. V informatice se tímto obecným problémem zabývá samotná disciplína – teorie *formálních jazyků a automatů*, která je přednášena jako jeden ze základních kursů na všech informatických oborech vysokých škol. Proto ponecháme převody z infixové notace zatím stranou. Co bychom si ale z této sekce měli odnést je poznatek, že jednou z hlavních výhod prefixové notace je její jednoduchost a tím pádem snadná strojová zpracovatelnost. Zvolení prefixové notace symbolických výrazů jako základní notace pro dialekty LISPu lze tedy bez nadsázky označit jako geniální tah.

11.5 Vyhodnocování výrazů v postfixové a v bezzávorkové notaci

V této sekci se budeme snažit pro výrazy používané v předchozí sekci sestavit vhodné evaluátory. Pro výrazy v prefixovém tvaru to činit nemusíme, protože evaluátor již je nám k dispozici v podobě procedury

Program 11.10. Procedura vyhodnocující postfixové výrazy.

```
(define postfix-eval
  (lambda (expr)
    (cond
     ((number? expr) expr)
     ((symbol? expr) (eval expr))
     (else
      (let iter ((expr expr)
                 (args '()))
        (cond ((null? expr) '())
              ((null? (cdr expr))
               (apply (postfix-eval (car expr))
                      (reverse args)))
              (else (iter (cdr expr)
                          (cons (postfix-eval (car expr)) args))))))))))
```

`eval` a jedná se samotný evaluátor jazyka Scheme. Zaměříme se tedy na konstrukci procedur provádějící vyhodnocování výrazů v postfixové a bezzávorkové reverzní notaci. Infixovou notací se opět kvůli složitosti nebudeme zabývat (studenti se s problematikou bíže setkají také v kursu *překladačů*).

Vyhodnocování závorkovaných postfixových výrazů bychom mohli provést analogicky jako vyhodnocování výrazů v prefixové notaci, pouze musíme počítat s tím, že výraz, jenž se má vyhodnotit na proceduru (nebo na speciální formu), stojí v seznamu jako poslední. Neformálně můžeme popsat vyhodnocování výrazů v postfixové notaci následovně:

- číslo se vyhodnotí na svou hodnotu,
- symbol se vyhodnotí na svou vazbu,
- je-li daný výraz seznam, tak se začnou vyhodnocovat jeho prvky jeden po druhém, až se vyhodnotí poslední z nich, pak se ověří, jestli se (poslední) vyhodnotil na proceduru. Pokud ano, je procedura aplikována s argumenty jimiž jsou elementy vzniklé vyhodnocením předchozích prvků seznamu.

Postup můžeme formalizovat procedurou `postfix-eval` uvedenou v programu 11.10. Tato procedura obsahuje pomocnou koncově rekurzivní proceduru `iter`, která je v jejím těle jednorázově aplikována (pomocí pojmenovaného `let`). Tato pomocná procedura se stará o postupné procházení prvků v seznamu, které jsou během průchodu postupně vyhodnocovány. Výsledky jejich vyhodnocení se postupně akumulují v seznamu navázaném na `args`. Při dosažení posledního prvku je aplikována procedura získaná vyhodnocením posledního elementu. Argumenty předané při aplikaci procedury jsou právě elementy, které byly akumulovány v seznamu navázaném na `args`.

```
(postfix-eval '(60 (10 20 +) /))   ⟹  2
(postfix-eval '((10 20 +) 60 /))   ⟹  1/2
(postfix-eval '(10 20 +))          ⟹  30
```

Je samozřejmě možné vyhodnocovat i výrazy obsahující procedury pro manipulaci s páry:

```
(postfix-eval '((10 20 cons) (30 40 cons) list))  ⟹  ((10 . 20) (30 . 40))
```

Upozorněme na fakt, že vyhodnocovací proces implementovaný v programu 11.10 je oproti vyhodnocovacímu procesu jazyka Scheme silně zjednodušený. Vůbec se například nepočítá s použitím speciálních forem. Viz příklad:

`(postfix-eval '(20 ((x) x lambda)))` ⟹ „CHYBA: Symbol `x` nemá vazbu"

Důvodem chyby je právě pořadí, v jakém vyhodnocujeme argumenty. Pokud postupujeme v seznamu od jeho počátku až ke konci, právě až na konci zjistíme, zda-li se daný výraz vyhodnotil na proceduru nebo na speciální formu. To jest zavedení speciálních forem v našem modelu vyhodnocování by nemělo smysl, protože ještě před tím, než zjistíme typ aplikovaného elementu, jsou již vyhodnoceny všechny argumenty.

Nyní obrátíme naší pozornost na vyhodnocování bezzávorkových výrazů. To by mohlo být na první pohled složitější, protože ve výrazech chybějí závorky explicitně určující „strukturu výrazů“. I v tomto případě lze ale navrhnout jednoznačný vyhodnocovací proces a implementovat jej.

Před tím, než začneme konstruovat evaluátor výrazů, si musíme objasnit jeden nepříjemný rys, který souvisí s odstraněním závorek z postfixových výrazů. V bezzávorkové notaci totiž již obecně neexistuje jednoznačný přepis prefixových výrazů, pokud bereme v potaz symboly pro operace s libovolným počtem operandů, viz příklad:

```
(+ 2 3 (* 4 5))                       ⟹  25
(prefix->polish '(+ 2 3 (* 4 5)))     ⟹  (2 3 4 5 * +)
(+ 2 (* 3 4 5))                       ⟹  62
(prefix->polish '(+ 2 (* 3 4 5)))     ⟹  (2 3 4 5 * +)
```

Zde vidíme, že dva různé výrazy mají stejnou reverzní bezzávorkovou reprezentaci. Než přistoupíme ke konstrukci vyhodnocovacího procesu, musíme tuto situaci vyřešit. Jinak bychom nevěděli, zda-li vyhodnotit výraz `(2 3 4 5 * +)` na hodnotu `25` nebo na hodnotu `62` (nebo ještě nějak úplně jinak). Jelikož problém nejednoznačnosti spočívá v tom, že neznáme počet operandů pro operace, nabízí se dvě metody řešení:

(i) Každou operaci uvažovat vždy pouze *s pevnou aritou*, to jest s pevným počtem operandů. Toto řešení je jednoduché a my se jej přidržíme. Zavedeme tabulku symbolů, ve které budeme mít pro každou operaci zaznamenán počet jejích operandů.

(ii) Do výrazů budeme před každý symbol operace vždy zapisovat počet operandů, na které se vztahuje. Například ve výrazu `(+ 2 3 (* 4 5))` má „`+`“ tři operandy a „`*`“ pouze dva. V reverzní bezzávorkové notaci bychom tedy výraz zapsali ve tvaru `(2 3 4 5 2 * 3 +)`, přitom červeně jsou zdůrazněny hodnoty reprezentující počty operandů. Na druhou stranu výraz `(+ 2 (* 3 4 5))` by byl reprezentován seznamem `(2 3 4 5 3 * 2 +)`. Oba výrazy jsou tedy různé a k nejednoznačnostem nedochází. Výrazy kódované tímto způsobem jsou pochopitelně delší, ale zase v nich můžeme používat operace s libovolnými operandy.

Nyní vezmeme v úvahu úmluvu (i) a budeme se soustředit na implementaci. Jako první budeme definovat tabulku symbolů, viz definici v programu 11.11. V této tabulce je pro každý symbol uvažované operace

Program 11.11. Jednoduchá tabulka s prostředím vazeb pro postfixový evaluátor.

```
(define env
  `((+ 2 ,+)
    (- 2 ,-)
    (/ 2 ,/)
    (* 2 ,*)
    (n 1 ,-)
    (^ 2 ,expt)))
```

tříprvkový záznam ve formě seznamu `(`⟨*symbol*⟩ ⟨*arita*⟩ ⟨*procedura*⟩`)`, kde ⟨*symbol*⟩ je jméno operace, ⟨*arita*⟩ je číslo určující počet operandů dané operace a ⟨*procedura*⟩ je procedura používající se při „aplikaci operace“. Všimněte si, že v tabulce 11.11 jsme definovali unární operaci označenou `n`, což je unární minus. Pro unární minus již nemůžeme zvolit symbol „`-`“, který je vyhrazen pro odčítání dvou čísel (uvědomte si, že arita „`-`“ je dána pevně na jedinou hodnotu).

Vyhodnocování postfixových výrazů budeme provádět pomocí dodatečné datové struktury – zásobníku. Samotná reprezentace zásobníku je z našeho pohledu primitivní, protože jej můžeme reprezentovat přímo seznamem (`cons` „přidává“ prvek na vrchol zásobníku, `car` vrací prvek na vrcholu, a `cdr` „odebírá“ prvek z vrcholu zásobníku). Během vyhodnocování tedy budeme mít k dispozici *vstupní výraz* reprezentovaný lineárním seznamem čísel a symbolů a *zásobník*. V každém kroku se ze vstupního výrazu odebere nejlevější „slovo" (číslo nebo symbol) a zpracuje se. Zásobník je na počátku prázdný. Podle typu slova na vstupu budeme rozlišovat dvě situace:

(i) Na vstupu je číslo. V tomto případě přesuneme číslo na vrchol zásobníku a zpracujeme následující vstupní slovo.

(ii) Je-li na vstupu symbol f označující operaci, odstraníme jej ze vstupu a vyzvedneme ze zásobníku tolik prvků, jaká je arita symbolu f (aritu nejdeme v tabulce symbolů, kterou jsme již zavedli) a aplikujeme příslušnou proceduru s těmito argumenty (procedura je opět k nalezení v tabulce). Výsledek vyhodnocení dáme na vrchol zásobníku.

Vyhodnocování končí vypotřebováním vstupu. V tom případě je výsledek uložen na vrcholu zásobníku (nebo za výsledek vyhodnocování můžeme považovat celý zásobník).

Příklad 11.4. (a) Uvažujme výraz `(10 20 +)`. Průběh jeho vyhodnocování daný předchozím postupem, můžeme zaznamenat v tabulce se dvěma sloupci. První sloupec ukazuje stav vstupního výrazu, ze kterého jsou postupně zepředu odebírány prvky, a druhý sloupec představuje aktuální stav zásobníku. Každý řádek tabulky pak koresponduje s jedním elementárním krokem výpočetu. V případě našeho výrazu bude výpočet vypadat takto:

vstup	*zásobník*
`(10 20 +)`	`()`
`(20 +)`	`(10)`
`(+)`	`(20 10)`
`()`	`(30)`

Výsledek vyhodnocení je tedy `30`.

(b) V případě výrazu `'(10 20 30 + *)` probíhá vyhodnocování takto:

vstup	*zásobník*
`(10 20 30 + *)`	`()`
`(20 30 + *)`	`(10)`
`(30 + *)`	`(20 10)`
`(+ *)`	`(30 20 10)`
`(*)`	`(50 10)`
`()`	`(500)`

Výsledek vyhodnocení je `500`.

(c) Konečně, v případě `'(10 20 + 30 *)` probíhá vyhodnocování takto:

vstup	*zásobník*
`(10 20 + 30 *)`	`()`
`(20 + 30 *)`	`(10)`
`(+ 30 *)`	`(20 10)`
`(30 *)`	`(30)`
`(*)`	`(30 30)`
`()`	`(900)`

Výsledek vyhodnocení je `900`.

Proceduru provádějící vyhodnocování pomocí zásobníku můžeme naprogramovat tak, jak je to uvedeno v programu 11.12. Zde je uvedena procedura `polish-eval`, která jako argumenty akceptuje daný výraz, který bude vyhodnocen, a tabulku symbolů. V našem případě budeme vždy používat tabulku z programu 11.11. Nic však nebrání tomu, abychom zavedli jinou tabulku. Procedura ve svém těle používá pomocnou interní iterativní proceduru `iter`, která má dva argumenty: seznam reprezentující vstupní výraz a seznam reprezentující zásobník (na počátku prázdný). Ve svém těle procedura dělá přesně to, co jsme řekli v předchozích paragrafech. Pomocná procedura `list-pref` uvedená v programu 11.11 slouží k získání prvních n prvků ze seznamu: procedura slouží k vyzvednutí více hodnot ze zásobníku před aplikací procedury odpovídající symbolu operace.

Program 11.12. Procedura vyhodnocující výrazy v reverzní bezzávorkové notaci.

```
(define list-pref
  (lambda (n l)
    (if (<= n 0)
        '()
        (cons (car l)
              (list-pref (- n 1) (cdr l))))))

(define polish-eval
  (lambda (expr env)
    (let iter ((input expr)
               (stack '()))
      (if (null? input)
          stack
          (let ((word (car input))
                (tail (cdr input)))
            (if (not (symbol? word))
                (iter tail (cons word stack))
                (let ((func (assoc word env)))
                  (if (not func)
                      (error "Symbol not bound")
                      (let ((arity (cadr func))
                            (proc (caddr func)))
                        (iter tail
                              (cons (apply proc
                                           (reverse (list-pref arity stack)))
                                    (list-tail stack arity))))))))))))
```

```
(polish-eval '(10 20 +) env)        ⟹  (30)
(polish-eval '(10 20 30 +) env)     ⟹  (50 10)
(polish-eval '(10 20 30 + *) env)   ⟹  (500)
(polish-eval '(10 20 30 * +) env)   ⟹  (610)
(polish-eval '(10 20 + 30 *) env)   ⟹  (900)
(polish-eval '(10 20 * 30 +) env)   ⟹  (230)
(polish-eval '(10 30 n +) env)      ⟹  (-20)
(polish-eval '(10 n 30 +) env)      ⟹  (20)
(polish-eval '(10 n 30 n +) env)    ⟹  (-40)
(polish-eval '(10 n 30 n + n) env)  ⟹  (40)
(polish-eval '(10 8 2 / ^) env)     ⟹  (10000)
```

Zásobníkové vyhodnocování postfixových bezzávorkových výrazů se používá daleko častěji, než jak bychom možná intuitivně čekali. Na tomto stylu vyhodnocování výrazů je založeno celé jedno minoritní paradigma – *zásobníkové paradigma* (dost často se však za samostatné paradigma nepovažuje). Typickým zástupcem zásobníkového jazyka je FORTH. Jazyk PostScript, který umí interpretovat každá trochu lepší tiskárna, a který je v současnosti de facto standardem, pokud jde přes přenositelné formáty popisu tiskové strany, je rovněž dialektem jazyka FORTH upraveným právě pro použití popisu obsahu tiskové strany. Virtuální stroje zpracovávající programy v bajtkódu (viz první lekci) jsou vesměs naprogramovány jako zásobníkové vyhodnocovací programy zpracovávající programy v zásobníkových jazycích. Existují samozřejmě i hardwarové zásobníkové procesory, které jsou součástí malých počítačů a tiskáren.

Shrnutí

V této lekci jsme se zabývali zpracováním symbolických výrazů reprezentovaných seznamy skládajícími se z čísel, symbolů a dalších seznamů reprezentující symbolické výrazy. Nejprve jsme pro zjednodušení práce zavedli rozšíření kvotování – tak zvané kvazikvotování, což je obecnější metoda kvotování umožňující programátorům snadněji definovat některé seznamy. V lekci jsme se potom věnovali zjednodušování aritmetických výrazů, tyto aritmetické výrazy byly reprezentovány seznamy. Ukázali jsme několik procedur pro zjednodušování s různou vnitřní strukturou a s různými schopnostmi. Dalším příkladem bylo počítání symbolických derivací. Potom jsme naši pozornost přesunuli na reprezentaci výrazů v infixové, postfixové, a reverzní bezzávorkové notaci. Naprogramovali jsme řadu procedur pro konverzi výrazů v prefixové notaci na ostatní notace a zpět. Poukázali jsme na fakt, že infixová notace je z hlediska strojového zpracování dost komplikovaná. V poslední sekci jsme se věnovali konstrukcí procedur vyhodnocující výrazy v postfixové a reverzní bezzávorkové notaci. Navrhli jsme několik modelů jejich vyhodnocování a ukázali jejích silné a slabé stránky. Vyhodnocování reverzních bezzávorkových výrazů jsme naprogramovali pomocí manipulace s dodatečným zásobníkem, který byl fyzicky reprezentován seznamem.

Pojmy k zapamatování

- kvazikvotování

Nově představené prvky jazyka Scheme

- speciální forma `quasiquote`
- procedura `assoc`

Kontrolní otázky

1. *Co je kvazikvotování? jak se liší od kvotování?*
2. *Jak jsme naprogrogramovali zjednodušování aritmetických výrazů?*
3. *Jak jsme naprogrogramovali symbolickou derivaci?*

4. *Jak jsme naprogrogramovali převod mezi jednotlivými notacemi?*

Cvičení

1. Bez použití interpretru určete výsledky vyhodnocení následujících výrazů:
```
(quasiquote symbol)
`(symbol)
(car ``symbol)
`(+ 1)
`(,+ 1 2)

`(1 ,1)
`(1 ,,1)
`(,+ . ,-)
`(1 + 2 ,@3)
(quote (,@()))

(quasiquote quasiquote)
(quasiquote (+ 1 (unquote +)))
(quasiquote (1 2 ,(+ 1 2)))
`,(+ 1 2)
`,`,(+ 1 2)
`(1 2 ,@(build-list 5 (lambda (x) (* x x))) 3)

(quasiquote (1 2 (unquote-splicing (map list '(1 2 3))) 5))
unquote-splicing
```()
`'+
(quote unquote)
```
2. Upravte kód procedury `diff` v programu 11.6 tak, aby bylo možné derivovat výrazy, ve kterých je součet a součin použit na libovolné množství argumentů.

---

**Úkoly k textu**

1. Popište, jak nahradit použití speciální formy `quasiquote`.
2. Rozšiřte proceduru `simplify` tak, aby neměla nedostatky uvedené na konci sekce 11.2.
3. Naprogramujte proceduru vyhodnocování pro infixové výrazy s pevně daným počtem argumentů (nejvýše dva).

---

## Řešení ke cvičením

1. symbol, (symbol), quasiquote, (+ 1), („procedura sčítání“ 1 2)
(1 1) chyba („procedura sčítání“ . „procedura odčítání“) chyba ((unquote-splicing ()))
quasiquote (+ 1 „procedura sčítání“) (1 2 3) 3 3 (1 2 0 1 4 9 16 3)
(1 2 (1) (2) (3) 5) chyba (quasiquote (quasiquote ())) '+ unquote
2. Oproti programu 11.6 stačí změnit lokální definici tabulky takto:
```
(define table
  (let ((2*->* (lambda (x) (if (and (pair? x) (equal? '*2 (car x)))
                               `(* ,@(cdr x))
                               x))))
```
```

```
`((+ . ,(lambda l `(+ ,@(map (lambda(x) (derive x)) l))))
  (- . ,(lambda (x y) `(- ,(derive x) ,(derive y))))
  (* . ,(lambda l (derive (foldr (lambda (x a) `(*2 ,x ,a)) 1 l))))
  (*2 . ,(lambda (x y) `(+ (* ,(2*->* x) ,(derive y)) (* ,(derive x) ,(2*->* y)))))
  (/ . ,(lambda (x y) `(/ (- (* ,(derive x) ,y) (* ,x ,(derive y))) (* ,y ,y)))))
```

Lekce 12: Čistě funkcionální interpret Scheme

Obsah lekce: V této lekci se nejprve budeme zabývat automatickým přetypováním a generickými procedurami. Pro generické procedury zavedeme jednoduchou metodu jejich aplikace prostřednictvím vyhledávání procedur pomocí vzorů uvedených v tabulkách. Ve zbytku lekce ukážeme implementaci čistě funkcionální podmnožiny jazyka Scheme. Zaměříme se na implementaci datové reprezentace elementů jazyka pomocí manifestovaných typů a na implementaci vyhodnocovacího procesu.

Klíčová slova: generická procedura, koerce, manifestovaný typ, přetypování, tabulka generických procedur.

12.1 Automatické přetypování a generické procedury

V úvodní sekci této lekce uděláme malou odbočku od hlavního zaměření lekce jímž bude konstrukce interpretu jazyka Scheme. V této sekci se budeme zabývat *generickými procedurami.* Většina procedur, které jsme v jazyku Scheme používali, byly těsně svázané s konkrétním datovým typem. Například procedura `append` prováděla spojování seznamů a nebylo ji možné použít s argumenty jiných typů než jsou seznamy. Toto chování je v mnoha situacích žádoucí.

Někdy je ale potřeba jednoduše používat tutéž proceduru s argumenty různých datových typů. Například procedura pro sčítání `+` je schopna pracovat s čísly v přesné reprezentaci (racionální zlomky) a s čísly v přibližné reprezentaci (čísla s pohyblivou desetinnou tečkou). Proceduru sčítání je dokonce možné použít s argumenty různých typů současně, to jest s některými argumenty (čísly) v přesné reprezentaci a s některými argumenty (čísly) v přibližné reprezentaci. Procedura pro sčítání tedy musí provádět dílčí operace, které jsou podmíněné typem elementů. V některých případech tedy musí provést před samotným součtem jistou „konverzi datových typů“, aby mohla aritmetickou operaci provést.

Procedurám, které svou činnost řídí podle typů svých argumentů, říkáme *generické procedury.* Přesněji řečeno, za generické procedury považujeme ty procedury, které podle typů argumentů provádějí aplikace jiných procedur a provádějí případně dodatečnou konverzi argumentů (elementů) na elementy vyžadovaných typů. Například procedura sčítání v jazyku Scheme je generická procedura. Pokud je sčítání aplikováno s racionálními argumenty, provede se sčítání přímo v přesná reprezentaci a výsledkem je opět číslo v přesné reprezentaci. Pokud by byl byt'jen jeden z argumentů v přibližné reprezentaci, pak procedura provede nejprve konverzi všech argumentů do přibližné reprezentace a provede sečtení v přibližné reprezentaci. Výsledkem takového sečtení je číslo v přibližné reprezentaci. To by nás nemělo překvapit, protože při práci s interpretem jazyka Scheme jsme si mohli všimnout následujícího chování `+`:

```
(+ 1/2 10)      ⟾  21/2
(+ 0.5 10)      ⟾  10.5
(+ 1/2 2/3)     ⟾  7/6
(+ 0.5 0.666)   ⟾  1.166
```

Automatickým konverzím na elementy jiných typů, ke kterým může docházet během používání generických procedur, se říká *koerce.* Koerce je tedy obecně řečeno *implicitní přetypování.* Skoro ve všech programovacích jazycích jsou k dispozici nějaké prostředky pro *explicitní přetypování,* to jest programátorem vynucenou změnu typu elementu. Příkladem může být třeba *převod čísla na řetězec znaků.* V jazyku Scheme, jako ve většině programovacích jazyků, existuje datový typ „řetězec znaků“. S řetězci lze dělat běžné operace, jimiž se podrobně nebudeme zabývat, protože to není naším hlavním cílem (zájemce odkazuji na specifikaci R^5RS jazyka Scheme, viz [R5RS]). Jednou z operací s řetězci je jejich spojování. K tomu slouží procedura `string-append`. Spojování řetězců je demonstrováno následujícími příklady:

```
(string-append)                 ⟾  ""
(string-append "Ahoj" "svete")  ⟾  "Ahojsvete"
(string-append "Ahoj " "svete") ⟾  "Ahoj svete"
(string-append "a" "b" "c")     ⟾  "abc"
```

Převod čísel na řetězce se provádí pomocí procedury `number->string`, které pro dané číslo vrací řetězec znaků obsahující *externí reprezentaci čísla.* Pro ilustraci viz následující ukázky použití procedury:

```
(number->string 10.2)         ⟹  "10.2"
(number->string (/ -1 2))     ⟹  "-1/2"
(number->string (sqrt -1))    ⟹  "0+1i"
```

Předchozí operaci bychom de facto mohli chápat jako *explicitní přetypování*. Na programátorovu žádost byl element číslo „převeden“ na element řetězec, se kterým se dál může pracovat. Naproti tomu výše uvedené *implicitní přetypování* (*koerci*) provádějí automaticky některé (generické) procedury. Je zajímavé, že koerce je někdy chápána jako potenciální zdroj chyb a některé jazyky koerci vůbec neumožňují. Jedním z takových jazyků je například funkcionální jazyk ML.

Ve zbytku této sekce si ukážeme modelový příklad, jak lze naprogramovat uživatelsky definované generické procedury. Vyjdeme z nám dobře známé operace sčítání čísel a obohatíme ji tak, aby mohla sloužit i ke spojování řetězců. Umožníme navíc, abychom mohli tuto novou verzi generického sčítání používat s argumenty různých typů (tedy s čísly i s řetězci). Toto zobecnění sčítání je ve skutečnosti docela praktické. Umožňuje nám snadno vytvářet textový výstup, v němž jsou některé hodnoty dopočtené, viz ukázku:

```
(let ((x 10)
      (y 20))
  (+ "Soucin " x " a " y
     " je " (* x y) "."))  ⟹  "Soucin 10 a 20 je 200."
```

Jako první vytvoříme pomocný predikát `match-type?` sloužící k testování typů argumentů. Predikát je

Program 12.1. Porovnávání datového typu argumentu se vzorem.

```
(define match-type?
  (lambda (preds args)
    (or (and (null? preds) (null? args))
        (and (pair? preds)
             (pair? args)
             ((car preds) (car args))
             (match-type? (cdr preds) (cdr args))))))
```

uveden v programu 12.1. Prvním argumentem procedury `match-type?` je seznam predikátů a druhým argumentem je seznam elementů. Výsledek aplikace `match-type?` pro dané argumenty je „pravda“, právě když jsou oba dva seznamy předané jako argumenty stejně dlouhé a elementy z druhého seznamu odpovídají po řadě typům daným predikáty v prvním seznamu. Použití `match-type?` je demonstrováno následujícím příkladem.

```
(match-type? `(,number? ,number?) '(10.2 "Ahoj"))  ⟹  #f
(match-type? `(,number? ,string?) '(10.2 "Ahoj"))  ⟹  #t
(match-type? `(,string? ,number?) '(10.2 "Ahoj"))  ⟹  #f
(match-type? `(,string? ,string?) '(10.2 "Ahoj"))  ⟹  #f
```

Predikát `match-type?` bude použit při testování shody argumentů se vzorem v tabulce určující chování generické procedury. Pro každou generickou proceduru budeme vždy uvažovat *tabulku metod*[18], což bude tabulka ve speciálním tvaru určujícím vzory a procedury pro aplikaci, pokud argumenty budou odpovídat danému vzoru.

Pokud se nyní pro ilustraci zaměříme pouze na dva argumenty, můžeme nově implementované generické sčítání popsat tabulkou uvedenou v programu 12.2. Po vyhodnocení výrazu z programu 12.2 dojde k navázání seznamu párů na symbol `table-generic`. Seznam se skládá z párů jejichž první prvek lze chápat jako identifikátor typu konkrétní operace a druhý prvek je samotná operace, která se má provést. Například první řádek bychom mohli číst tak, že v případě sčítání dvou čísel je použita původní operace „sčítání čísel“

[18] Pojem „metoda“ je často používán v objektovém programování. My budeme za metody považovat procedury spolu se vzorem jejich použití, které budou součástí tabulky určující činnost generické procedury.

Program 12.2. Konkrétní tabulka metod generické procedury.

```
(define table-generic
  (let ((+ +))
    `(((+ ,number? ,number?) . ,+)
      ((+ ,number? ,string?) . ,(lambda (x y)
                                   (string-append (number->string x) y)))
      ((+ ,string? ,number?) . ,(lambda (x y)
                                   (string-append x (number->string y))))
      ((+ ,string? ,string?) . ,string-append))))
```

(všimněte si vazby + v let-výrazu, pro připomenutí viz lekci 3). Na druhém řádku tabulky je uvedeno, že v případě sčítání čísla a řetězce se provede konverze čísla na řetězec a výsledek se spojí s předaným řetězcem. Při aplikaci poslední jmenované procedury vlastně z hlediska uživatele generické procedury (kterou vytvoříme dále) dochází ke koerci (přetypování čísla na řetězec).

Program 12.3. Vyhledání příslušné operace v tabulce metod generické procedury.

```
(define table-lookup
  (lambda (table op args)
    (cond ((null? table) #f)
          ((not (equal? (caaar table) op)) (table-loopkup (cdr table) op args))
          ((match-type? (cdaar table) args) (cdar table))
          (else (table-lookup (cdr table) op args)))))
```

Procedura table-lookup uvedená v programu 12.3 má na starost vyhledávat příslušnou operaci v tabulce metod generické procedury. Procedura table-lookup bere jako argumenty tabulku metod generické procedury, symbol identifikující generickou proceduru (v jedné tabulce metod mohou být záznamy pro více generických operací), a seznam argumentů, podle kterých se v tabulce vyhledává. Procedura iterativně prochází tabulku a při nalezení první shody vzoru metody s argumenty (zde se používá predikát match-type?) je vrácena procedura jež je součástí metody.

```
(table-lookup table-generic '+ '(10 20))          ⟹  primitivní procedura „sčítání čísel“
(table-lookup table-generic '+ '(10 "svete"))     ⟹  uživ. def. procedura z 2. řádku tabulky
(table-lookup table-generic '+ '("Ahoj" 20))      ⟹  uživ. def. procedura z 3. řádku tabulky
(table-lookup table-generic '+ '("Ahoj" "svete")) ⟹  primitivní procedura „spojení řetězců“
(table-lookup table-generic '+ '(10 #t))          ⟹  #f
```

Aplikace generických procedur bude prováděna pomocí procedury apply-generic, viz program 12.4. Procedura apply-generic provede vyhledání metody v tabulce navázané na table-generic podle vzoru

Program 12.4. Aplikace generické procedury.

```
(define apply-generic
  (lambda (op . args)
    (let ((proc (table-lookup table-generic op args)))
      (if proc
          (apply proc args)
          (error "No method for these types")))))
```

a provede následnou aplikaci procedury jež je součástí shodující se metody. Pro větší pohodlí při aplikaci generické procedury provedeme navázání nově vytvořené procedury na symbol `+`. Kód provádějící tuto vazbu je uveden v programu 12.5. Pomocná procedura `foldr1` pracuje stejně jako `foldr` s jedním

Program 12.5. Generická procedura pro sčítání.

```
(define foldr1
  (lambda (f term l)
    (cond ((null? l) term)
          ((null? (cdr l)) (car l))
          (else (f (car l) (foldr1 f term (cdr l)))))))

(define +
  (lambda args
    (foldr1 (lambda (x y)
              (apply-generic '+ x y))
            0
            args)))
```

seznamem, pouze s tím rozdílem, že hodnota navázaná na symbol `term` je vrácena pouze v případě, že seznam předaný `foldr1` jako třetí argument je prázdný. V případě, že seznam je jednoprvkový, je vrácen tento prvek – to je jediný rozdíl oproti původní verzi `foldr`. V programu 12.5 jsme použili `foldr1` k naprogramování nové procedury libovolných argumentů, která je posléze navázána na symbol `+`. Procedura `foldr1` je zde použita k zobecnění aplikace generické procedury ze dvou argumentů na libovolné množství argumentů. Touto problematikou jsme se již zabývali v sekci 7.3. Novou generickou proceduru `+` je nyní možné používat následujícími způsoby:

```
(+)                  ⟾  0
(+ 10)               ⟾  10
(+ 10 20)            ⟾  30
(+ 10 20 30)         ⟾  60
(+ 10 "x" 30)        ⟾  "10x30"
(+ 10 20 30 "")      ⟾  "102030"
(+ 10 20 "" 30)      ⟾  "102030"
(+ 10 "" 20 30)      ⟾  "1050"
(+ "" 10 20 30)      ⟾  "60"
```

Všimněte si, že pokud uvádíme jako argumenty pouze čísla, generická procedura `+` se chová stejně jako původní procedura sčítání. Zbývá odpovědět na otázku, proč jsme při vytvoření generické procedury `+` použili `foldr1` místo klasického `foldr`. Je to z důvodu praktičnosti. Při použití `foldr` místo `foldr1` bychom totiž dostali nepřirozený výsledek v situaci, kdy je poslední ze sčítaných elementů řetězec:

```
(+ "")                                ⟾  "0"
(+ "Ahoj " "svete")                   ⟾  "Ahoj svete0"
(+ "Faktorial " 4 " je " 24 ".")      ⟾  "Faktorial 4 je 24.0"
```

Naproti tomu verze se `foldr1` se chová přirozeně:

```
(+ "")                                ⟾  ""
(+ "Ahoj " "svete")                   ⟾  "Ahoj svete"
(+ "Faktorial " 4 " je " 24 ".")      ⟾  "Faktorial 4 je 24."
```

12.2 Systém manifestovaných typů

Jelikož nám v této lekci jde o konstrukci interpretu jazyka Scheme, jako jeden z prvních problémů musíme vyřešit, jak budeme *reprezentovat jednotlivé elementy jazyka*: čísla, pravdivostní hodnoty, symboly, páry, procedury (primitivní a uživatelsky definované), speciální formy a další. V této sekci nejprve naznačíme obecnou reprezentaci elementů při níž užijeme tak zvanou *manifestaci typů*. Každý element se bude skládat ze dvou základních částí:

(i) *identifikátor typu elementu,*

(ii) *data charakterizující element.*

Identifikátor typu bude sloužit k jednoznačnému určení o jaký element se jedná. Pomocí tohoto identifikátoru tedy rozlišíme, zda-li například daný element reprezentuje pár nebo proceduru. Jako identifikátory typů budeme používat symboly jejichž jména budou typ elementu označovat. Druhou částí každého elementu jsou data představující „hodnotu elementu“.

Poznámka 12.1. (a) Pokud si například představíme dvě čísla `-13` a `27`, pak jejich vnitřní reprezentace (v našem konstruovaném interpretu Scheme) budou obsahovat shodný identifikátor typu „číslo“. Oba elementy budou mít ale různou datovou část – v prvním případě bude datová část uchovávat číselnou hodnotu -13, v druhém případě 27.

(b) Otázkou je, proč v elementech potřebujeme identifikátory jejich typů a zda-li bychom se bez nich mohli obejít. Identifikátory skutečně potřebujeme, abychom mohli správně rozlišovat datové typy elementů (a v důsledku abychom byli schopni správně vyhodnocovat elementy). V některých případech totiž pouze na základě znalosti „datové části elementu“ nejsme schopni určit jeho typ. Vezměme si například tečkový pár `(2 . 3)`. Mohli bychom se na něj dívat jako na dvojici číselných hodnot. Ta může reprezentovat třeba zlomek $\frac{2}{3}$, nebo komplexní číslo $2 + 3i$. Z pohledu syntaxe a sémantiky jazyka, viz sekci 1.2, je identifikátor typu *sémantická informace* určující *význam datové části elementu* (datová část elementu by sama o sobě neměla žádný význam).

Pod pojmem *manifestovaný typ* máme tedy na mysli přítomnost identifikátoru typu „v datech“. Při reprezentaci elementů budeme používat manifestaci typů, jak jsme to naznačili v úvodu této sekce.

Pro práci s elementy a manifestovanými typy si vytvoříme sadu pomocných procedur, které jsou uvedeny v programu 12.6. Procedura `curry-make-elem` slouží k vytváření *konstruktorů elementů* s manifestovaným

Program 12.6. Systém manifestovaných typů.

```
(define curry-make-elem
  (lambda (type-tag)
    (lambda (data)
      (cons type-tag data))))

(define get-type-tag car)

(define get-data cdr)

(define curry-scm-type
  (lambda (type)
    (lambda (elem)
      (equal? type (get-type-tag elem)))))
```

typem. Procedura `curry-make-elem` bere jako argument symbol (formální argument `type-tag`) označující daný typ. Symbolům označujícím typy se někdy říká „visačky“ nebo „tagy“ (z anglického *tags*). Procedura `curry-make-elem` vrací konstruktor jímž je procedura jednoho argumentu. Tento argument představuje hodnotu, kterou chceme vložit do datové části elementu. Na fyzické úrovni je každý element reprezentován párem, jehož první složka je visačka a druhá složka je tvořena hodnotou elementu. Všimněte si, že procedura `curry-make-elem` pouze rozkládá proceduru dvou argumentů `cons` na dvě procedury jednoho argumentu, tento princip jsme popsali v sekci 2.4 jako *currying*. Procedury `get-type-tag` a `get-data` pro daný element vrací jeho visačku respektive jeho datovou složku. Poslední procedura v programu 12.6 je `curry-scm-type`. Tato procedura pro daný identifikátor typu vrací predikát testující zda-li daný element je tohoto typu či nikoliv. Viz následující příklad použití.

Nejprve vytvoříme konstruktory a predikáty testující typ pro dva různé datové typy (racionální a komplexní čísla):

```
(define make-frac (curry-make-elem 'fraction))
(define make-cplx (curry-make-elem 'complex-number))
(define frac? (curry-scm-type 'fraction))
(define cplx? (curry-scm-type 'complex-number))
```

Předchozí procedury můžeme používat následujícím způsobem:

```
(define a (make-frac '(2 . 3)))
(define b (make-cplx '(2 . 3)))
a               ⟹  (fraction 2 . 3)
b               ⟹  (complex-number 2 . 3)
(get-data a)    ⟹  (2 . 3)
(get-data b)    ⟹  (2 . 3)
(frac? a)       ⟹  #t
(frac? b)       ⟹  #f
(cplx? a)       ⟹  #f
(cplx? b)       ⟹  #t
```

Elementy jazyka Scheme budeme reprezentovat jako hodnoty s *manifestovaným typem.* Typ elementu je manifestován pomocí *visačky,* což je identifikátor typu. Typ elementu je potřeba rozeznávat, protože samotná datová část elementu jednoznačně neurčuje typ (sémantiku) elementu. V další sekci uvidíme, že manifestace typu nám umožní rozlišovat od sebe například vnitřní reprezentaci párů a uživatelsky definovaných procedur. Samotný princip manifestace typů je samozřejmě použitelný i při řešení jiných problémů než je reprezentace elementů jazyka Scheme.

12.3 Datová reprezentace elementů jazyka Scheme

V této sekci ukážeme implementaci jednotlivých elementů jazyka Scheme, které budeme potřebovat při vytvoření jeho interpretu. Budeme postupovat od nejjednodušších elementů ke složitějším.

Před tím, než začneme, je potřeba udělat několik terminologických poznámek. Jelikož se zabýváme implementací interpretu jazyka Scheme *v jazyku Scheme,* pracujeme vlastně se dvěma interprety současně. Prvním z interpretů je pro nás ten interpret, který používáme při vývoji programu. Tímto programem je (druhý) interpret jazyka Scheme. V této sekci se tedy budeme zabývat reprezentací elementů nově vytvářeného interpretu Scheme, nikoliv reprezentací elementů v interpretu, který při vytváření používáme. Abychom zjednodušili terminologii, zavedeme nyní pojmy *metainterpret* a *interpret* jazyka Scheme:

- *metainterpret jazyka Scheme* je již existující interpret, který používáme pro vytváření dalších programů (mimo jiné našeho nového *interpretu* jazyka Scheme),
- *interpret jazyka Scheme* je (meta)program pro metainterpret jazyka Scheme, který provádí interpretaci jisté podmnožiny jazyka Scheme.

Podobně jako rozlišujeme pojmy metainterpret a interpret můžeme odlišovat další pojmy, se kterými jsme se doposud setkali. Tak třeba *metajazyk* (jazyk interpretovaný *metainterpretem*) a *jazyk* (jazyk interpretovaný *interpretem*), *metaelement* (element *metajazyka*) a *element* (element *jazyka*), *metaprogram* (program v *metajazyku*) a *program* (program v *jazyku*). Abychom situaci ještě zjednodušili, budeme někdy předponu „meta“ vynechávat, a to v případě, kdy bude jasné, že se bavíme o původním interpretu jazyka Scheme nebo o souvisejících pojmech.

Pokud budeme hovořit o „metapojmech“, budeme tím mít vždy na mysli pojmy vztažené k metainterpretu jazyka Scheme, to jest k interpretu, který používáme k vývoji našeho nového interpretu podmnožiny jazyka Scheme. Na programy, které dosud vytváříme se taky musíme dívat ze dvou úhlů pohledu. Metaprogramy jsou programy pro výchozí metainterpret, tedy náš nový interpret jazyka Scheme je sám o sobě metaprogram. Programy pro nově vytvářený interpret nazýváme v souladu s předchozí úmluvou pouze „programy“.

Nyní již obrat'me naši pozornost k reprezentaci elementů jazyka Scheme. Mezi nejjednodušší elementy patří bezpochyby *čísla*. Při jejich implementaci využijeme toho, že se snažíme vytvořit „Scheme ve Scheme“, tedy nový interpret Scheme v již existujícím metainterpretu Scheme. Díky tomu můžeme de facto převzat veškeré aritmetické (meta) procedury, jak uvidíme dále. Při implementaci také nebudeme rozlišovat jednotlivé typy čísel (přesnou a nepřesnou reprezentaci čísel, racionální čísla, komplexní čísla a tak dále). Pro čísla tedy zavedeme pouze jejich konstruktor a predikát testující typ „číslo“ následovně:

```
(define make-number (curry-make-elem 'number))
(define scm-number? (curry-scm-type 'number))
```

Analogicky jednoduchá bude reprezentace *symbolů*. Hodnotou symbolu (jakožto elementu jazyka) je pro nás jeho „jméno“, které můžeme ztotožnit s řetězcem znaků. Abychom situaci ještě více zjednodušili, nebudeme k označení jmen symbolů používat řetězce znaků, ale metasymboly dostupné v metainterpretu jazyka Scheme. Konstruktory symbolů a predikát testující typ tedy zavedeme:

```
(define make-symbol (curry-make-elem 'symbol))
(define scm-symbol? (curry-scm-type 'symbol))
```

Nyní se zaměříme na speciální elementy jazyka, které se vyhodnocovaly na sebe sama. Jednalo se o *pravdivostní hodnoty*, *prázdný seznam*, a element zastupující *nedefinovanou hodnotu*. Pro tyto elementy nepotřebujeme vytvářet konstruktory, protože pravdivostní hodnoty jsou pouze dvě, prázdný seznam je pouze jeden a stejně tak pouze jeden je element zastupující nedefinovanou hodnotu.

V případě pravdivostních hodnot tedy vytvoříme dva nové elementy zastupující nepravdu a pravdu. Tyto elementy navážeme na symboly `scm-false` a `scm-true`. Dále vytvoříme predikát testující typ „pravdivostní hodnota“. Viz následující kód.

```
(define scm-false ((curry-make-elem 'boolean) #f))
(define scm-true  ((curry-make-elem 'boolean) #t))
(define scm-boolean? (curry-scm-type 'boolean))
```

Prázdný seznam bude nově vytvořený element navázaný na `the-empty-list`:

```
(define the-empty-list ((curry-make-elem 'empty-list) '()))
(define scm-null? (lambda (elem) (equal? elem the-empty-list)))
```

A konečně stejným způsobem vytvoříme i element zastupující nedefinovanou hodnotu:

```
(define the-undefined-value ((curry-make-elem 'undefined) '()))
(define scm-undefined? (lambda (elem) (equal? elem the-undefined-value)))
```

Všimněte si toho, že předchozí predikáty `scm-null?` a `scm-undefined?` využívají toho, že prázdný seznam a nedefinovaná hodnota jsou unikátní elementy daného typu. V tomto případě je tedy možné naprogramovat predikáty testující typ jako predikáty testující rovnost s daným elementem.

Nyní se budeme zabývat tečkovými páry. Nejprve podotkněme, že pro to, abychom v našem interpretu mohli uvažovat seznamy, není nutné (a ani vhodné) vytvářet elementy jazyka typu „seznam“. Plně si vystačíme s dále navrženými páry a prázdným seznamem, který jsme definovali výše. To je zcela v souladu s tím, jak jsme zavedli seznamy pomocí párů v lekci 5. Páry pro nás tedy budou speciální elementy typu „pár“, jejichž datovou složkou budou tvořit dva elementy v pevně daném pořadí. Fyzicky budeme páry reprezentovat pomocí metapárů ve tvaru

`(pair . (`⟨*první*⟩ `.` ⟨*druhý*⟩`))`,

kde ⟨*první*⟩ je element představující první složku páru a ⟨*druhý*⟩ je element představující druhou složku páru. Pro páry vytvoříme jejich konstruktor, dva selektory a predikát testující typ „pár“. V programu 12.7 je uveden konstruktor páru `make-pair`. Jedná se o proceduru dvou argumentů, která vznikla v prostředí

Program 12.7. Reprezentace tečkových párů.

```
(define make-pair
  (let ((make-physical-pair (curry-make-elem 'pair)))
    (lambda (head tail)
      (make-physical-pair (cons head tail)))))

(define scm-pair? (curry-scm-type 'pair))

(define pair-car
  (lambda (pair)
    (if (scm-pair? pair)
        (car (get-data pair))
        (error "CAR: argument must be a pair"))))

(define pair-cdr
  (lambda (pair)
    (if (scm-pair? pair)
        (cdr (get-data pair))
        (error "CDR: argument must be a pair"))))
```

v němž je na symbol `make-physical-pair` navázána procedura vytvářející element s manifestovaným typem „pár“. Samotná procedura `make-pair` provádí pouze jednu aplikaci `make-physical-pair` při níž jsou obě dvě složky spojeny do metapáru pomocí `cons`. Opět jsme tedy zvolili strategii, že k reprezentaci párů nám slouží metapáry a konstruktor páru `make-pair` je vytvořen pomocí konstruktoru metapáru `cons`. Pro objasnění uveďme následující příklady použití `make-pair`:

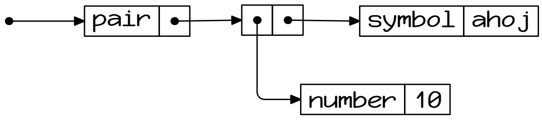
```
(define a (make-number 1))
(define b (make-number 2))
(make-pair a b)                ⟹  (pair (number . 1) number . 2)
(make-pair a the-empty-list)   ⟹  (pair (number . 1) empty-list)
(make-pair the-empty-list b)   ⟹  (pair (empty-list) number . 2)
```

V programu 12.7 je dále uveden predikát `scm-pair?` testující, zda-li je daný element typu „pár“. Dále jsou zde uvedeny selektory `pair-car` a `pair-cdr` sloužící k přístupu k první, případně druhé, složce párů. Jejich implementace je přímočará.

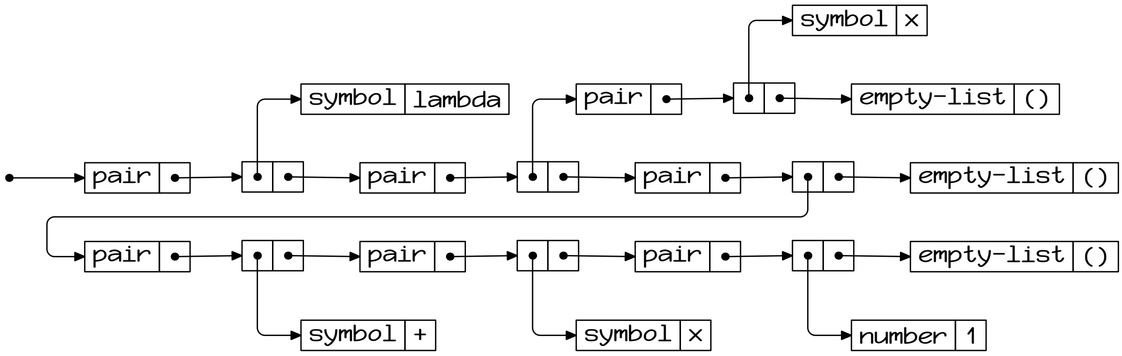
Příklad 12.2. Nyní si již můžeme udělat základní představu o tom, jak bude vypadat reprezentace složitějších elementů jazyka pomocí metaelementů. Například pár `(10 . ahoj)`, to jest pár, jehož první složkou je číslo a druhou složkou je symbol bude v metainterpretu fyzicky reprezentován metaelementem vyobrazeným na obrázku 12.1. Dále například λ-výraz `(10 . ahoj)` bude v metainterpretu reprezentován tak, jak ukazuje obrázek 12.2. Jak je na první Pohled zřejmé, reprezentace tohoto relativně malého seznamu je dost velká. To je jakási daň, kterou musíme zaplatit za visačky jednoznačně určující typy elementů.

Další elementy jazyka, jejichž reprezentaci popíšeme, jsou *prostředí.* Koncept prostředí byl představen již v lekci 1 a dále zpřesněn v lekci 2. Prostředí potřebujeme k udržování vazeb mezi symboly a elementy a kvůli schůdné implementaci lexikálního rozsahu platnosti: každé prostředí, kromě globálního, má ukazatele na svého lexikálního předka (prostředí svého vzniku). Datová část prostředí tedy musí obsahovat jednak tabulku vazeb mezi symboly a elementy a jednak (ukazatel na) další prostředí. Elementy typu prostředí tedy budeme reprezentovat metapáry ve tvaru

Obrázek 12.1. Fyzická reprezentace páru `(10 . ahoj)` pomocí metaelementů.



Obrázek 12.2. Fyzická reprezentace seznamu `(lambda (x) (+ x 1))` pomocí metaelementů.



`(environment . (`⟨*předek*⟩ `.` ⟨*tabulka*⟩`))`,

kde ⟨*předek*⟩ je buďto element prostředí nebo element „nepravda“ (dané prostředí nemá předka) a ⟨*tabulka*⟩ je tabulka vazeb mezi symboly a elementy udržovaná jako interní reprezentace seznamu ve tvaru

`((`⟨*symbol*$_1$⟩ ⟨*element*$_1$⟩`)`
` (`⟨*symbol*$_2$⟩ ⟨*element*$_2$⟩`)`
⋮
` (`⟨*symbol*$_n$⟩ ⟨*element*$_n$⟩`))`.

Pro práci s prostředími budeme potřebovat několik základních procedur. V první řadě to bude konstruktor prostředí `make-env` a dva selektory `get-pred` a `get-table`, které pro dané prostředí vracejí prostředí předka nebo tabulku vazeb. Tyto procedury jsou uvedeny v programu 12.8. V tomto programu je dále uveden predikát testující typ elementu „prostředí“. Všimněte si, že základní konstruktory a selektory prostředí jsou de facto stejné jako konstruktory a selektory párů, viz program 12.7. V případě prostředí ale platí, že jeho datové složky nejsou libovolné elementy, ale přesně vymezené elementy (první element je předek, tedy „nepravda“ nebo opět prostředí a druhý element je vždy tabulka vazeb).

Pro pohodlnou manipulaci s prostředím zavedeme další procedury. V našem novém interpretu totiž musíme mít nějak definováno prostředí počátečních vazeb, musíme mít tedy k dispozici procedury, kterými prostředí vytvoříme. Další prostředí již budou vznikat při aplikaci uživatelsky definovaných procedur tak, jak jsme to vysvětlili v lekci 2. V programu 12.9 je uvedeno několik procedur, které využijeme při definici počátečních prostředí. Predikát `global?` je pro daný element pravdivý, právě když je element prostředí, které nemá předka. Jedná se tedy o predikát testující, zda-li je předaný element globální prostředí. Pomocná procedura `assoc->env` provede převod *asociačního metaseznamu* na tabulku vazeb realizovanou *asociačním seznamem.* Při bližším pohledu je vidět, že `assoc->env` je v podstatě jen jednoduchá rekurzivní procedura, která převádí metaseznam metapárů na seznam párů ve vnitřní reprezentaci (nového interpretu), přitom také provádí vytváření nových symbolů z metasymbolů. Pro objasnění ještě uveďme příklad jejího použití:

```
(define s `((a . ,(make-number 10)) (b . ,(make-number 20))))
(assoc->env s)  ⟹  (pair (pair (symbol . a) number . 10)
                         pair (pair (symbol . b) number . 20)
                              empty-list)
```

Konečně procedura `make-global-env` z programu 12.9 slouží k vytvoření globálního prostředí. Její použití uvidíme v jedné z dalších sekcí.

Program 12.8. Reprezentace prostředí.

```
(define make-env
  (let ((make-physical-env (curry-make-elem 'environment)))
    (lambda (pred table)
      (make-physical-env (cons pred table)))))

(define scm-env? (curry-scm-type 'environment))

(define get-table
  (lambda (elem)
    (if (scm-env? elem)
        (cdr (get-data elem))
        (error "GET-TABLE: argument must be an environment"))))

(define get-pred
  (lambda (elem)
    (if (scm-env? elem)
        (car (get-data elem))
        (error "GET-PRED: argument must be an environment"))))
```

Při implementaci vyhodnocovacího procesu budeme potřebovat hledat vazby v prostředích. K tomuto účelu naprogramujeme dvě pomocné procedury. Procedura `scm-assoc` z programu 12.10 provádí prakticky totéž co standardní procedura `assoc` (viz standard R^{5}RS jazyka Scheme [R5RS]). Jediný rozdíl je v tom, že `scm-assoc` pracuje s asociačními seznamy (a nikoliv metaseznamy) v jejich interní reprezentaci. Procedura pro daný klíč a asociační seznam prohledává daný asociační seznam a vrací první pár, jehož první složka odpovídá klíči. Pokud žádný takový pár neexistuje, je vrácena „nepravda“. Proceduru `scm-assoc` tedy můžeme použít k vyhledání vazby v tabulce vazeb nějakého prostředí. Druhá procedura v programu 12.10 je procedura `lookup-env`, což je komplexnější procedura, která vyhledává vazbu symbolů v prostředí nebo vrací element navázaný na symbol `not-found`, pokud není vazba nalezena. Formální argument pojmenovaný `search-nonlocal?` slouží jako přepínač, kterým lze ovlivňovat, co se má stát, pokud vazba není nalezena v tabulce aktuálního prostředí. Pokud je při aplikaci `lookup-env` na `search-nonlocal?` navázána hodnota „pravda“, pak se při nenalezení vazby v tabulce lokálního prostředí postupuje k nadřazenému prostředí. V případě, že je na `search-nonlocal?` navázána „nepravda“, pak je při nenalezení vazby v tabulce lokálního prostředí okamžitě vrácena hodnota navázaná na `not-found`. Procedura `lookup-env` bude používána k hledání vazeb symbolů přímo během vyhodnocování elementů.

Nyní se budeme zabývat reprezentací procedur a speciálních forem. Reprezentace *primitivních procedur*, tedy procedur, které budou přímo zabudované v našem novém interpretu, bude jednoduchá. Vytvoříme jejich konstruktor a predikát testující typ „primitivní procedura“ následovně:

```
(define make-primitive (curry-make-elem 'primitive))
(define scm-primitive? (curry-scm-type 'primitive))
```

Datovou složkou primitivní procedury bude *metaprocedura*, tedy nějaká procedura, která je přímo součástí konstruovaného interpretu a která bude při aplikaci prováděna metainterpretem jazyka Scheme. V tuto chvíli připomeňme, že už v první lekci, kdy jsme primitivní procedury zavedli, jsem upozornili na fakt, že se nebudeme zabývat tím, jak jsou vytvořené. Obecně můžeme říct, že primitivní procedury jsou vždy vytvořeny pomocí nějakých metaprocedur. Primitivní metaprocedury (primitivní procedury v metainterpretu) jsou rovněž vytvořeny pomocí nějakých „metametaprocedur“, které jsou, v případě interpretu jazyka Scheme vzniklého kompilací, zapsané v kódu stroje (a zdrojové kódy těchto „metametaprocedur“ budou naprogramovány nejspíš v nějakém vyšším programovacím jazyku, třeba v jazyku C, což je oblíbený jazyk pro tvorbu interpretů a překladačů).

Program 12.9. Konstruktor pro globální prostředí.

```
(define assoc->env
  (lambda (l)
    (if (null? l)
        the-empty-list
        (make-pair (make-pair (make-symbol (caar l))
                              (cdar l))
                   (assoc->env (cdr l))))))

(define make-global-env
  (lambda (alist-table)
    (make-env scm-false (assoc->env alist-table))))

(define global?
  (lambda (elem)
    (and (scm-env? elem)
         (equal? scm-false (get-pred elem)))))
```

Ať tak či onak, nyní se *musíme* zabývat tím, jak primitivní procedury vytvářet, protože se zabýváme konstrukcí interpretu jazyka. Některé primitivní procedury budeme programovat jako *uživatelsky definované metaprocedury.* Řadu důležitých primitivních procedur, například aritmetické procedury, ale můžeme vytvořit jednoduchým „trikem“ a to tak, že pouze zabalíme metaproceduru, která je protějškem dané procedury, do pomocného programu, který z daných elementů vyzvedne jejich datovou část, aplikuje metaproceduru s takto získanými hodnotami a nakonec výsledkem aplikace (tedy metaelement) převede do interní reprezentace. Na tuto problematiku se blíže podíváme v dalších sekcích.

V lekci 2 jsme uvedli, že *uživatelsky definované procedury* chápeme jako trojice hodnot $\langle\langle \textit{parametry}\rangle, \langle \textit{tělo}\rangle, \mathcal{P}\rangle$, kde $\langle \textit{parametry}\rangle$ je seznam formálních argumentů, $\langle \textit{tělo}\rangle$ je element reprezentující tělo procedury a $\mathcal{P}$ je prostředí vzniku procedury. V lekci 3 jsme rozšířili procedury tak, že jsme umožnili, aby v jejich těle bylo přítomno víc výrazů. Toto rozšíření jsme učinili z důvodu pohodlného zavedení interních definic. Jelikož se ale jedná o rys, který není čistě funkcionální (při vytváření definic dochází k vedlejšímu efektu jímž je modifikace prostředí), budeme se dále zabývat konceptem procedur tak, jak jsme jej představili v lekci 2 (jeden výraz v těle). Uživatelsky definovaná procedura je tedy element jazyka, který v sobě agreguje tři hodnoty (seznam formálních argumentů, prostředí a tělo). Reprezentaci uživatelsky definovaných procedur jakožto elementů našeho jazyka můžeme tedy provést zcela přímočaře tak, jak je to ukázáno v programu 12.11. Procedura `make-procedure` je konstruktor uživatelsky definovaných procedur akceptující tři argumenty: prostředí, seznam argumentů a tělo. Tyto tři položky jsou v elementu fyzicky uloženy do tříprvkového metaseznamu. Dále máme k dispozici tři selektory `procedure-environment`, `procedure-arguments` a `procedure-body` vracející prostředí, seznam argumentů a tělo pro daný element typu „uživatelsky definovaná procedura“. Nakonec jsme opět zavedli predikát testující daný typ. Jelikož primitivní procedury a uživatelsky definované procedury chápeme souhrnně jako procedury, vytvoříme navíc dodatečný predikát testující, zda-li je element procedura (primitivní nebo uživatelsky definovaná):

```
(define scm-procedure?
  (lambda (elem)
    (or (scm-primitive? elem)
        (scm-user-procedure? elem))))
```

Posledním typem uvažovaných elementů jazyka budou speciální formy. Speciální formy budou implementované opět pomocí metaprocedur. Pro každou speciální formu, kterou bude náš interpret obsahovat, budeme vytvářet speciální uživatelsky definovanou metaproceduru. Zatím tedy vytvoříme pouze základní reprezentaci elementů typu „speciální forma“:

Program 12.10. Vyhledávání vazeb v prostředí.

```
(define scm-assoc
  (lambda (key alist)
    (cond ((scm-null? alist) scm-false)
          ((equal? key (pair-car (pair-car alist))) (pair-car alist))
          (else (scm-assoc key (pair-cdr alist))))))

(define lookup-env
  (lambda (env symbol search-nonlocal? not-found)
    (let ((found (scm-assoc symbol (get-table env))))
      (cond ((not (equal? found scm-false)) found)
            ((global? env) not-found)
            ((not search-nonlocal?) not-found)
            (else (lookup-env (get-pred env) symbol #t not-found))))))
```

Program 12.11. Reprezentace uživatelsky definovaných procedur.

```
(define make-procedure
  (let ((make-physical-procedure (curry-make-elem 'procedure)))
    (lambda (env args body)
      (make-physical-procedure (list env args body)))))

(define procedure-environment (lambda (proc) (car (get-data proc))))
(define procedure-arguments (lambda (proc) (cadr (get-data proc))))
(define procedure-body (lambda (proc) (caddr (get-data proc))))

(define scm-user-procedure? (curry-scm-type 'procedure))
```

```
(define make-specform (curry-make-elem 'specform))
(define scm-specform? (curry-scm-type 'specform))
```

Nyní se dostáváme do bodu, kdy si můžeme dovolit malou epistemickou úvahu. Je zajímavé, že při našem exkurzu programováním v jazyku Scheme jsme postupovali od procedur vyšších řádů směrem k párům. U párů pro uvedli, že je můžeme plně vyjádřit pomocí uživatelsky definovaných procedur vyšších řádů. Nyní postupujeme zdánlivě obráceně. Uživatelsky definované procedury máme úplně reprezentované pomocí (meta) párů.

12.4 Vstup a výstup interpretu

Konstruovaný interpret jazyka Scheme musí mít k dispozici základní vstupní a výstupní části, konkrétně *reader* (proceduru realizující načítání vstupních symbolických výrazů a jejich převod do interní reprezentace) a *printer* (proceduru, která se stará o vytištění externí reprezentace elementů).

Pro načtení symbolického výrazu můžeme použít primitivní proceduru `read`, se kterou jsme se již setkali v lekci 5. Po načtení však musíme ještě provést konverzi hodnoty získané aplikací `read` do naší interní reprezentace. To jest všechny načtené symboly, čísla a seznamy je potřeba převést na příslušné elementy „symbol“, „číslo“ a „páry“ tak, jak jsme je představili v předchozí sekci. Tuto konverzi pro nás bude provádět nově vytvořená procedura `expr->intern`, viz program 12.12. Samotný reader je již možné naprogramovat

Program 12.12. Převod do interní reprezentace a implementace readeru.

```
(define expr->intern
  (lambda (expr)
    (cond ((symbol? expr) (make-symbol expr))
          ((number? expr) (make-number expr))
          ((and (boolean? expr) expr) scm-true)
          ((boolean? expr) scm-false)
          ((null? expr) the-empty-list)
          ((pair? expr) (make-pair (expr->intern (car expr))
                                   (expr->intern (cdr expr))))
          ((eof-object? expr) #f)
          (else (error "READER: Syntactic error.")))))

(define scm-read
  (lambda ()
    (expr->intern (read))))
```

pomocí `read` a procedury `expr->intern` tak, jak je to uvedeno v programu 12.12. Dodejme, že pokud bychom neměli v jazyku Scheme k dispozici proceduru `read`, museli bychom rovněž naprogramovat samotné načítání vstupních výrazů. V případě jazyka Scheme by to nebylo příliš obtížné, ale tématicky tento problém spadá do jiných kurzů. Laskavého čtenáře tímto odkazujeme na kurzy *formální jazyky a automaty* a *překladače*. V následující ukázce je uvedeno použití procedury `expr->intern`.

```
(expr->intern 1)           ⟹  (number . 1)
(expr->intern '+)          ⟹  (symbol . +)
(expr->intern '(1 . 2))    ⟹  (pair (number . 1) number . 2)
(expr->intern '(- 13))     ⟹  (pair (symbol . -) pair (number . 13) empty-list)
  ⋮
```

Printer se používá především k *vypisování výsledků vyhodnocení*. Při implementaci printeru musíme oproti readeru naprogramovat opačnou konverzi. Tedy konverzi elementu v interní reprezentaci na čitelnou reprezentaci, jenž může být vytištěna na obrazovku, zapsána do souboru, a tak dále. Pochopitelně, že naprogramovat printer je obecně vždy jednodušší než naprogramovat reader (konverze řetězce znaků na strukturovaná data je mnohem obtížnější než konverze strukturovaných dat na řetězec znaků[19]). Jedna z možností, jak naprogramovat printer je uvedena v programu 12.13. Printer bychom samozřejmě mohli realizovat i mnohem jednodušeji, například následující procedurou, která provede pouze vytištění interní reprezentace elementu na obrazovku:

```
(define scm-print
  (lambda (elem)
    (display elem)))
```

V tomto případě by ale výpis některých elementů nebyl přehledný (uvidíme dále).

12.5 Implementace vyhodnocovacího procesu

V této sekci rozebereme implementaci vyhodnocovacího procesu včetně aplikace procedur a speciálních forem. Postup bude kopírovat teorii, kterou jsme probrali v prvních dvou lekcích tohoto textu. Nejprve při-

[19]Toto pozorování by pro nás mělo být vlastně malým poučením. Při programování čehokoliv se vždy vyplatí reprezentovat data v co možná nejvíc strukturované podobě. Nikdy tím nemůžeme nic ztratit (snad kromě větší paměťové náročnosti na jejich uložení) a přidaná hodnota může být opravdu velká. Není náhodou, že metodám (automatického) strukturování velkých dat se v informatice věnuje řada disciplín.

Program 12.13. Převod do externí reprezentace Implementace printeru.

```
(define intern->expr
  (lambda (expr)
    (cond ((scm-pair? expr) (cons (intern->expr (pair-car expr))
                                  (intern->expr (pair-cdr expr))))
          ((scm-env? expr) "#<environment>")
          ((scm-primitive? expr) "#<primitive-procedure>")
          ((scm-user-procedure? expr) "#<user-defined-procedure>")
          ((scm-specform? expr) "#<special-form>")
          ((scm-undefined? expr) "#<undefined>")
          (else (get-data expr)))))

(define scm-print
  (lambda (elem)
    (display (intern->expr elem))))
```

pomeňme, že aplikace primitivních procedur a speciálních forem bude řešena pomocí aplikace uživatelsky definovaných metaprocedur.

Jak jsme již předeslali v předchozí sekci, v některých případech je možné vytvořit primitivní procedury s využitím primitivních metaprocedur pomocí jejich „pouhého zabalení“ do pomocné procedury. Pomocná procedura sloužící jako jakási obálka se stará o konverzi elementů na metaelementy (před aplikací metaprocedury) a opačně o konverzi metaelementů na elementy (po aplikaci metaprocedury). Na provedení tohoto zabalení „metaprocedury“ můžeme vytvořit překvapivě jednoduchou proceduru `wrap-primitive`, která je zobrazena v programu 12.14. Použití této procedury uvidíme v dalších sekcích. Princip `wrap-primitive`

Program 12.14. Reprezentace primitivních procedur.

```
(define wrap-primitive
  (lambda (proc)
    (make-primitive
     (lambda arguments
       (expr->intern (apply proc (map get-data arguments)))))))
```

si nejlépe uvědomíme na následujícím příkladu, ve kterém nejprve na symbol `p` navážeme element jímž je primitivní procedura vzniklá z metaprocedury sčítání. Datovou složkou tohoto elementu je tedy metaprocedura vzniklá vyhodnocením vnitřního λ-výrazu uvedeného v těle procedury z programu 12.14. Viz příklad:

`(define p (wrap-primitive +))`
`p` $\Longmapsto$ `(primitive .` „metaprocedura realizující proceduru sčítání čísel“`)`

Primitivní proceduru bychom nyní mohli aplikovat následovně:

`((cdr p) (make-number 10) (make-number 20))` $\Longmapsto$ `(number . 30)`

Během předchozí aplikace byla provedena extrakce metačísel z elementů reprezentujících čísla 10 a 20. Dále byla aplikována primitivní metaprocedura sčítání a jejím výsledkem je metačíslo 30. To bylo nakonec zkonvertováno na element pomocí procedury `expr->intern`, kterou jsme představili v programu 12.12 na straně 302. Výše uvedenou metodou převezmeme v našem novém interpretu další aritmetické procedury.

Jelikož bude během vyhodnocování občas potřeba konvertovat metaseznamy (seznamy složené z metapárů) na seznamy (seznamy složené z párů) a obráceně, vytvoříme si pomocný konstruktor `convert-list`, viz jeho kód v programu 12.15. Procedura `convert-list` je de facto obecný konstruktor seznamů a me-

Program 12.15. Obecný konvertor seznamu na seznam ve vnitřní reprezentaci a obráceně.

```
(define convert-list
  (lambda (a-null? a-car a-cdr b-cons b-nil f l)
    (if (a-null? l)
        b-nil
        (b-cons (f (a-car l))
                (convert-list a-null? a-car a-cdr b-cons b-nil f
                              (a-cdr l))))))
```

taseznamů. Pomocí něj můžeme vytvořit řadu konvertorů těchto datových struktur a metastruktur. Ty nejdůležitější z nich jsou uvedeny v programu 12.16. Procedura `scm-list->list` konvertuje seznamy na

Program 12.16. Konverze seznamů na metaseznamy o obráceně.

```
(define scm-list->list
  (lambda (scm-l)
    (convert-list scm-null? pair-car pair-cdr cons '() (lambda (x) x) scm-l)))

(define list->scm-list
  (lambda (l)
    (convert-list null? car cdr make-pair the-empty-list (lambda (x) x) l)))

(define map-scm-list->list
  (lambda (f scm-l)
    (convert-list scm-null? pair-car pair-cdr cons '() f scm-l)))
```

metaseznamy. Naopak procedura `list->scm-list` konvertuje seznamy na metaseznamy. Pomocí procedury `map-scm-list->list` je rovněž možné převést seznam na metaseznam, ale během zpracování prvků výchozího seznamu se ještě používá procedura jednoho argumentu k modifikaci prvků (analogicky jako u standardní procedury `map`).

Nyní se již můžeme podívat na implementaci vyhodnocovacího procesu. Nejprve se budeme zabývat implementací procedury provádějící vyhodnocení elementů. V ní použijeme několik procedur, které objasníme dále. K vyhodnocování elementů bude sloužit procedura `scm-eval`, která je uvedena včetně jednoduchých komentářů v programu 12.17 na straně 305. Všimněte si, že `scm-eval` má dva argumenty, první z nich je element, který vyhodnocujeme, a druhým je aktuální prostředí, ve kterém tento element vyhodnocujeme. To koresponduje s tím jak jsme zavedli Eval$[E, \mathcal{P}]$ v definici 2.7 na straně 48. V těle procedury `scm-eval` je jeden `cond`-výraz, ve kterém se rozhoduje o způsobu vyhodnocení elementu na základě jeho typu. Zde uplatníme manifestaci typů a predikáty testující typ elementu, které jsme doposud zavedli.

V prvním případě je vyřešena situace, kdy je daný element symbol. V tomto případě je hledá jeho vazba (počínaje předaným aktuálním prostředím) pomocí procedury `lookup-env` z programu 12.10 na straně 301.

Druhá větev `cond`-výrazu ošetřuje případ, kdy je daný element seznam nebo lépe řečeno, kdy je daný element *pár*[20]. V tomto případě, je nejprve v daném prostředí vyhodnocen první prvek páru. Všimněte

[20] Uvědomte si, že test toho, zda-li daný element seznam nelze provést v konstantním čase. Z důvodu efektivity by tedy bylo nešťastné definovat vyhodnocování *seznamů*, ale mnohem jednodušší je pracovat přímo s *páry*, které daný seznam tvoří. Tak je tomu i v tomto případě. Navíc nám tento postup umožní zapisovat program pomocí párů v tečkové notaci (i když to zřejmě není příliš užitečné a ani přehledné).

Program 12.17. Implementace vlastního vyhodnocovacího procesu.

```
;; vyhodnot vyraz v danem prostredi
(define scm-eval
  (lambda (elem env)

    ;; vyhodnocovani elementu podle jejich typu
    (cond

     ;; symboly se vyhodnocuji na svou aktualni vazbu
     ((scm-symbol? elem)
      (let* ((binding (lookup-env env elem #t #f)))
        (if binding
            (pair-cdr binding)
            (error "EVAL: Symbol not bound"))))

     ;; vyhodnoceni seznamu
     ((scm-pair? elem)

      ;; nejprve vyhodnotime prvni prvek seznamu
      (let* ((first (pair-car elem))
             (args (pair-cdr elem))
             (f (scm-eval first env)))

        ;; podle prvniho prvku rozhodni o co se jedna
        (cond

         ;; pokud se jedna o proceduru, vyhodnot argumenty a aplikuj
         ((scm-procedure? f)
          (scm-apply f (map-scm-list->list
                        (lambda (elem)
                          (scm-eval elem env))
                        args)))

         ;; pokud se jedna o formu, aplikuj s nevyhodnocenymi argumenty
         ((scm-specform? f)
          (scm-specform-apply env f (scm-list->list args)))

         ;; na prvnim miste stoji nepripustny prvek
         (error "EVAL: First element did not eval. to procedure"))))

     ;; vse ostatni (cisla, boolean, ... se vyhodnocuje na sebe sama)
     (else elem))))
```

si, že zde dochází k *rekurzivní aplikaci* scm-eval. Výsledek vyhodnocení je navázán v lokálním metaprostředí na metasymbol f. Dále vyhodnocování postupuje dvěma směry podle toho jakého typu je element navázaný na f. Pokud je to procedura (primitivní nebo uživatelsky definovaná), provedeme aplikaci metaprocedury scm-apply, kterou si záhy popíšeme. Tato metaprocedura má na starosti provedení aplikace procedury f s danými argumenty. Všimněte si, že argumenty jsou předány ve formě metaseznamu, jehož prvku jsou vyhodnocené elementy ze seznamu argumentů (zde opět provádíme rekurzivní aplikaci scm-eval). V případě, kdy je na f navázána speciální forma je provedena její aplikace pomocí metaprocedury scm-specform-apply. Zde si všimněme toho, že argumenty jsou scm-specform-apply předány bez vyhodnocení a navíc jako jeden z argumentů pro scm-specform-apply předáváme aktuální prostředí. To je nezbytné k tomu, aby mohla každá speciální forma provádět další vyhodnocování a aby věděla, ve kterém prostředí má vyhodnocování provádět.

Konečně v poslední větvi cond-výrazu (else-větev) obsaženého v těle scm-eval je vyřešeno vyhodnocování všech ostatních elementů (tedy čísel, pravdivostních hodnot, prázdného seznamu, nedefinované hodnoty, procedur, speciálních forem a prostředí): tyto elementy se *vyhodnocují na sebe sama.*

Všimněte si, že procedura scm-eval de facto implementuje postup, který jsme uvedli v definici 2.7 na straně 48. Procedura scm-eval se od tohoto postupu liší jen v technických drobnostech. Například implementace separátního bodu (A) z definice 2.7 není přítomna, protože je řešena již v rámci bodu (D), viz else-větev v proceduře scm-eval. Nyní se tedy můžeme konečně „prakticky přesvědčit“, že vyhodnocovací proces skutečně funguje tak, jak jsme jej původně uvedli.

Abychom dokončili vyhodnocovací proces, musíme vytvořit metaprocedury provádějící aplikaci procedur a speciálních forem. Aplikace speciálních forem je elementární. Jelikož každá forma řídí vyhodnocování svých argumentů jiným způsobem, necháváme při aplikaci speciální formy veškerý průběh vyhodnocování na metaproceduře, která je datovou složkou speciální formy. Proceduru scm-specform-apply použitou ve scm-eval bychom tedy mohli naprogramovat takto:

```
(define scm-specform-apply
  (lambda (env form args)
    (cond ((scm-specform? form) (apply (get-data form) env args))
          (else (error "APPLY: Expected special form")))))
```

Na předchozím kódu si opět všimněte, že metaproceduře realizující speciální formu je předáno prostředí jako první z argumentů. Implementaci jednotlivých speciálních forem ukážeme v další sekci.

V případě aplikace procedur je situace složitější. Musíme jednak rozlišit mezi primitivními procedurami a uživatelsky definovanými procedurami. Situace v případě primitivních procedur bude podobně jednoduchá jako v případě speciálních forem, jak záhy uvidíme. V případě aplikace uživatelsky definovaných procedur však musíme provést kroky popsané v definici 2.12 na straně 50. To jest, musíme vytvořit nové lokální prostředí s vazbami a v něm vyhodnotit tělo procedury.

K vytvoření tabulky vazeb mezi formálními argumenty a předanými argumenty, která je nezbytnou součástí nově vytvářeného lokálního prostředí, bude sloužit procedura make-bindings z programu 12.18. Procedura make-bindings akceptuje dva argumenty, prvním s nich je seznam formálních argumentů a druhým je metaseznam elementů (hodnot), které se mají na dané formální argumenty „navázat“. Výsledkem aplikace make-bindings je tabulka vazeb, která je posléze použita jako součást nového prostředí.

Samotná aplikace procedur je prováděna pomocí scm-apply, viz program 12.19. Ještě před tím, že probereme scm-apply, se budeme zabývat obecnější pomocnou metaprocedurou scm-env-apply, která je rovněž uvedena v programu 12.19. Metaprocedura scm-env-apply bere jako argumenty proceduru, prostředí a argumenty s nimiž má být procedura aplikována. Pokud jde o primitivní proceduru, její aplikace je provedena prostou aplikací metaprocedury. V tomto případě nehraje prostředí předané scm-env-apply žádnou roli. V případě, že je vyžadována aplikace uživatelsky definované procedury se vytvoří nová tabulka vazeb mezi formálními argumenty této procedury a hodnotami, se kterými proceduru aplikujeme. Dále je pomocí této tabulky a předaného prostředí (navázaného na env) vytvořeno nové prostředí a v něm je vyhodnoceno tělo

Program 12.18. Vytvoření tabulky vazeb mezi formálními a skutečnými argumenty.

```
(define make-bindings
  (lambda (formal-args args)
    (cond ((scm-null? formal-args) the-empty-list)
          ((scm-symbol? formal-args)
           (make-pair (make-pair formal-args (list->scm-list args))
                      the-empty-list))
          (else (make-pair
                 (make-pair (pair-car formal-args) (car args))
                 (make-bindings (pair-cdr formal-args) (cdr args)))))))
```

Program 12.19. Implementace aplikace procedur.

```
(define scm-env-apply
  (lambda (proc env args)
    (cond ((scm-primitive? proc) (apply (get-data proc) args))
          ((scm-user-procedure? proc)
           (scm-eval (procedure-body proc)
                     (make-env env
                               (make-bindings (procedure-arguments proc)
                                              args))))
          (else (error "APPLY: Expected procedure")))))

(define scm-apply
  (lambda (proc args)
    (cond ((scm-primitive? proc) (scm-env-apply proc #f args))
          ((scm-user-procedure? proc)
           (scm-env-apply proc (procedure-environment proc) args))
          (else (error "APPLY: Expected procedure")))))
```

procedury. Jinými slovy, `scm-env-apply` v případě uživatelsky definovaných procedur provádí vyhodnocení jejich těla v prostředí, jehož předek je prostředí navázané na `env`. Tím je `scm-env-apply` obecnější než tradiční `apply`, kde je předek nově vzniklého prostředí dán prostředím vzniku procedury.

Naprogramovat `scm-apply` pomocí `scm-env-apply` je již velmi jednoduché. V programu 12.19 vidíme, že v případě aplikace primitivních procedur je voláno `scm-env-apply` s prostředním argumentem nastaveným na „nepravda“ (hodnota tohoto argumentu může být jakákoliv, protože jak jsme již zjistili, nebude k ničemu použita). V případě aplikace uživatelsky definovaných procedur je opět voláno `scm-env-apply`, tentokrát je ale prostředí předka nastaveno na prostředí vzniku procedury (což je prostředí uložené v datové složce elementu reprezentujícího uživatelsky definovanou proceduru).

12.6 Počáteční prostředí a cyklus REPL

Nyní nadefinujeme počáteční prostředí našeho interpretu. Jelikož se snažíme vytvářet čistě funkcionální interpret Scheme, tedy interpret, ve kterém žádná procedura ani speciální forma *nemá vedlejší efekt*, náš interpret nebude obsahovat speciální formu `define`, jejímž vedlejším efektem je modifikace aktuálního prostředí. V důsledku budeme muset v našem interpretu vytvářet rekurzivní procedury pomocí y-kombinátorů.

Dalším zajímavým rysem vašeho interpretu bude, že na počátku vyhodnocování nebudeme uvažovat pouze jediné prostředí, ale hned několik prostředí, které mezi sebou budou mít určité vazby. Konkrétně budeme rozlišovat dvě prostředí:

- *prostředí primitivních definic* (anglický název *toplevel environment*) – prostředí, které nemá předka (přísně vzato je to tedy *globální prostředí*), v němž jsou symboly navázány na primitivní procedury, speciální formy a případně další elementy vyjma uživatelsky definovaných procedur,
- *prostředí odvozených definic* (anglický název *midlevel environment*) – prostředí, jehož předkem je prostředí primitivních definic, v němž jsou symboly navázány na uživatelsky definované procedury.

Proč vůbec potřebujeme vytvořit dvě prostředí? Přísně vzato bychom nemuseli, ale tím pásem bychom v počátečním prostředí nemohli mít na žádný symbol navázanou netriviální uživatelsky definovanou proceduru. Každá uživatelsky definovaná procedura má totiž v sobě obsaženu informaci o prostředí svého vzniku. Jelikož nemáme k dispozici prostředky pro „změnu (mutaci) prostředí“, nemůžeme vytvořit proceduru, která by byla navázána v prostředí svého vlastního vzniku. Kromě prostředí primitivních definic tedy musíme mít k dispozici i *nové prostředí*, v němž mohou být definice uživatelsky definovaných procedur jejichž prostředím vzniku je právě prostředí primitivních definic. Takovým novým prostředím je právě prostředí odvozených definic[21]. Podotkněme, že uživatelsky definované procedury navázané v prostředí odvozených definic se „vzájemně nevidí“, protože prostředím vzniku všech procedur je prostředí primitivních definic. Kdybychom chtěli, aby některé z uživatelsky definovaných procedur mohly používat jiné, museli bychom vytvořit nové „prostředí odvozených definic II.“ jehož předkem by bylo existující prostředí odvozených definic. V tomto případě by uživatelsky definované procedury navázané na symboly v prostředí odvozených definic II. mohly používat procedury navázané ve výchozím prostředí odvozených definic. Dále bychom mohli dle potřeby vytvářet další „prostředí odvozených definic III., IV., . . . “.

Nyní si tedy rozebereme obsah prostředí primitivních definic a prostředí odvozených definic. Začneme *prostředím primitivních definic*. V našem interpretu bude prostředí zavedeno pomocí konstruktoru globálního prostředí `make-global-env` z programu 12.9 na straně 300.

```
(define scheme-toplevel-env
  (make-global-env
   `(
     ⋮
     )))
```

Výpustka uvedená v předchozím kódu značí místo, kam budou uváděny definice vazeb symbolů, kterými se budeme zabývat ve zbytku této sekce. Nejprve ukážeme definice několika základních speciálních forem. V programu 12.20 je ukázána definice speciální formy `if`. Tato definice (a každá další uvedená) je ve tvaru

Program 12.20. Definice speciální formy `if` v globálním prostředí.

```
(if . ,(make-specform
        (lambda (env condition expr . alt-expr)
          (let ((result (scm-eval condition env)))
            (if (equal? result scm-false)
                (if (null? alt-expr)
                    the-undefined-value
                    (scm-eval (car alt-expr) env))
                (scm-eval expr env))))))
```

páru `(`⟨*symbol*⟩ `.` ⟨*element*⟩`)`, kde `symbol` je metasymbol určující „jméno symbolu“ a ⟨*element*⟩ je konkrétní element navázaný na symbol. V našem případě je metasymbolem `if` a element navázaný na příslušný symbol v prostředí primitivních definic vznikne vyhodnocením výrazu `(make-specform` · · · . Zde by nás nemělo překvapit uvedení „ `,` “ před výrazem, protože si musíme uvědomit, že celý pár `(`⟨*symbol*⟩ `.` ⟨*element*⟩`)`

[21] Některé uživatelsky definované procedury bychom navázané mít mohli, třeba procedury vracející konstantní číselnou hodnotu nebo projekce. Tyto procedury totiž ve svém těle kromě formálních argumentů nepoužívají žádné další symboly. Tím pádem může být prostředí jejich vzniku nastaveno na „nepravda“.

je vložen do předchozího výrazu na místě výpustky, která je v kvazikvotovaném seznamu (viz seznam navázaný na scheme-toplevel-env).

Prohlédneme-li si výraz v programu 12.20 definující speciální formu if vidíme, že odpovídá formálnímu popisu if z definice 1.31 na straně 36. Všimněte si, že speciální forma je realizována metaprocedurou, jejíž první argument je *prostředí* a další argumenty odpovídají argumentům speciální formy if. Prostředí je předáváno kvůli tomu, abychom mohli v těle metaprocedury realizující speciální formu provádět vyhodnocování, viz aplikaci speciálních forem v programu 12.17. V těle metaprocedury je nejprve vyhodnocen argument condition a výsledek je navázán na symbol result. Podle výsledné hodnoty je rozhodnuto, zda-li se vyhodnotí expr nebo nepovinný poslední argument alt-expr. Ošetřen je i případ vrácení nedefinované hodnoty v případě, že condition se vyhodnotilo na „nepravda“ a v if-výrazu chybí náhradník, viz definici 1.31 na straně 36. Všimněte si, že pro definici speciální formy if jsme, mimo jiné, použili speciální metaformu if.

Analogicky jako speciální formu if bychom mohli zavést speciální formu and jejíž definici nalezneme v programu 12.21. Při naprogramování speciální formy jsme opět postupovali tak, že se chová jako and

Program 12.21. Definice speciální formy and v globálním prostředí.

```
(and . ,(make-specform
         (lambda (env . exprs)
           (let and-eval ((exprs exprs))
             (cond ((null? exprs) scm-true)
                   ((null? (cdr exprs)) (scm-eval (car exprs) env))
                   (else (let ((result (scm-eval (car exprs) env)))
                           (if (equal? result scm-false)
                               scm-false
                               (and-eval (cdr exprs))))))))))
```

představená v definici 2.22 na straně 66. Metaprocedura ve svém těle používá iterativní proceduru, která postupně prochází a vyhodnocuje jednotlivé elementy předané speciální formě. Pokud se některý z elementů vyhodnotí na „nepravda“, je iterace ukončena a vrácena je pravdivostní hodnota „nepravda“. Pokud již zbývá jen poslední element, je vrácena hodnota vzniklá jeho vyhodnocením. Pokud byly zpracovány všechny prvky, je výsledek aplikace speciální formy pravdivostní hodnota „pravda“.

Kromě and můžeme vytvořit i primitivní proceduru not následovně:

```
(not . ,(make-primitive
         (lambda (elem)
           (if (equal? elem scm-false)
               scm-true
               scm-false))))
```

Jelikož je not procedura a nikoliv speciální forma, není ji předáváno prostředí a předaný argument je již ve své vyhodnocené podobě. Pomocí not a and můžeme vyjadřovat i disjunktivní podmínky (viz komentář v sekci 2.7). Samozřejmě, že z programátorského hlediska by bylo dobré naprogramovat rovněž i speciální formu or. Realizace speciální formy or je jedním z řešených příkladů na konci této sekce, proto ji nyní uvádět nebudeme.

V programu 12.22 jsou uvedeny definice speciálních forem lambda, the-environment a quote. Jak je vidět, realizace těchto forem je velmi jednoduchá. Speciální forma lambda vytvoří nový element typu „uživatelsky definovaná procedura“ na základě předaného prostředí, seznamu argumentů a těla. Speciální forma the-environment je realizována metaprocedurou, která pouze vrací aktuální prostředí (tato metaprocedura je vlastně *identita*). Analogicky speciální forma quote vrací předaný element (bez jeho vyhodnocení), metaprocedura realizující quote je tedy *projekce*.

Pro uživatelsky definované procedury můžeme vytvořit sadu selektorů, viz program 12.23. Se třemi

Program 12.22. Definice speciálních forem `lambda`, `the-environment` a `quote` v globálním prostředí.

```
(lambda . ,(make-specform
            (lambda (env args body)
             (make-procedure env args body))))

(the-environment . ,(make-specform (lambda (env) env)))

(quote . ,(make-specform (lambda (env elem) elem)))
```

Program 12.23. Definice selektorů uživatelsky definovaných procedur v globálním prostředí.

```
(procedure-environment . ,(make-primitive procedure-environment))
(procedure-arguments . ,(make-primitive procedure-arguments))
(procedure-body . ,(make-primitive procedure-body))

(environment-parent . ,(make-primitive get-pred))

(environment->list . ,(make-primitive
                        (lambda (elem)
                          (if (equal? elem scm-false)
                              scm-false
                              (get-table elem)))))
```

procedurami `procedure-environment`, `environment-parent` a `environment->list` jsme se už setkali v lekci 6. Tyto procedury vrací prostředí vzniku procedury, nadřazené prostředí daného prostředí a poslední procedura převádí tabulku na čitelný asociační seznam. V programu 12.23 máme navíc selektor `procedure-arguments`, který pro danou proceduru vrací seznam jejich argumentů. Selektor `procedure-body` pro danou proceduru vrací výraz, který je tělem procedury.

Nyní můžeme popsat, jak do prostředí primitivních definic zabudujeme primitivní procedury `apply` a `eval`. Jelikož chceme, aby `apply` pracoval s libovolným počtem argumentů, to jest ve tvaru

`(apply` $\langle procedura \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$ $\langle seznam \rangle$`)`,

zavedeme nejprve pomocnou metaproceduru `apply-collect-arguments`, která ze všech předaných argumentů ve tvaru $\langle arg_1 \rangle \cdots \langle arg_n \rangle$ $\langle seznam \rangle$ sestaví (jediný) *seznam všech argumentů*. Kód této metaprocedury je v programu 12.24. V programu 12.25 jsou pak uvedeny definice procedur `eval` (procedura dvou argu-

Program 12.24. Sestavení seznamu argumentů pro obecný typ volání procedury `apply`.

```
(define apply-collect-arguments
  (lambda (args)
    (cond ((null? args) (error "APPLY: argument missing"))
          ((and (not (null? args)) (null? (cdr args)))
           (scm-list->list (car args)))
          (else (cons (car args) (apply-collect-arguments (cdr args)))))))
```

mentů z nichž druhý – reprezentující prostředí – je vždy povinný), `apply` a `env-apply` (zobecněná verze `apply`, které je jako druhý argument předáno prostředí, viz program 12.19 na straně 307).

Program 12.25. Definice eval, apply and env-apply v globálním prostředí.

```
(eval . ,(make-primitive
          (lambda (elem env)
            (scm-eval elem env))))

(apply . ,(make-primitive
           (lambda (proc . rest)
            (scm-apply proc (apply-collect-arguments rest)))))

(env-apply . ,(make-primitive
               (lambda (proc env . rest)
                 (scm-env-apply proc
                                env
                                (apply-collect-arguments rest)))))
```

Ostatní procedury v prostředí primitivních vazeb mohou být vytvořeny rutinně, nebudeme je tedy všechny vypisovat. Tímto čtenáře odkazujeme na zdrojové kódy interpretu jazyka Scheme, který je dodáván spolu s tímto učebním textem. Naznačme ale zhruba, jak definice vypadají. Budeme předpokládat, že v prostředí primitivních vazeb budeme mít na symbol pi navázánu číselnou hodnotu čísla π. Tuto vazbu bychom provedli třeba takto:

```
(pi . ,(make-number (* 4 (atan 1))))
```

Aritmetické procedury (sčítání, odčítání, zaokrouhlování a podobně) můžeme vytvořit pomocí metaprocedury wrap-primitive, kterou jsme již popsali v programu 12.14 na straně 303. Při definici aritmetických procedur tedy budeme postupovat následovně:

```
(* . ,(wrap-primitive *))
(+ . ,(wrap-primitive +))
(- . ,(wrap-primitive -))
(/ . ,(wrap-primitive /))
(< . ,(wrap-primitive <))
(<= . ,(wrap-primitive <=))
(= . ,(wrap-primitive =))
  ⋮
(tan . ,(wrap-primitive tan))
(truncate . ,(wrap-primitive truncate))
(zero? . ,(wrap-primitive zero?))
```

Rovněž definice konstruktorů a selektorů párů je jednoduchá. Pouze převedeme metaprocedury z programu 12.7 na straně 297 na primitivní procedury a uvedeme vazby na příslušné symboly.

```
(cons . ,(make-primitive make-pair))
(car . ,(make-primitive pair-car))
(cdr . ,(make-primitive pair-cdr))
```

Predikát testující prázdnost seznamu vytvoříme pomocí porovnání daného elementu s elementem reprezentujícím prázdný seznam:

```
(null? . ,(make-primitive
           (lambda (l)
             (expr->intern (equal? l the-empty-list)))))
```

Nyní se můžeme začít věnovat prostředí odvozených definic. Toto prostředí vytvoříme analogicky jako prostředí primitivních definic. Jediným rozdílem bude, že prostředí odvozených definic již bude mít jako svého předka nastaveno jiné prostředí, konkrétně právě prostředí primitivních definice. Následující fragment kódu ukazuje tvar, v jakém můžeme prostředí odvozených definic zavést.

```
(define scheme-midlevel-env
  (make-env
   scheme-toplevel-env
   (assoc->env
    `(
      ⋮
      (sgn . ,(make-procedure
               scheme-toplevel-env
               (expr->intern '(x))
               (expr->intern '(if (= x 0)
                                  0
                                  (if (> x 0)
                                      1
                                      -1)))))
      ⋮
       ))))
```

V předchozím kódu jsou opět uvedeny výpustky a je zde uveden příklad definice uživatelsky definované procedury `sgn`. Element navázaný na symbol `sgn` v tomto prostředí je skutečně uživatelsky definovaná procedura, protože se jedná o element vytvořený aplikací `make-procedure`, viz program 12.11 na straně 301. Při vytvoření této uživatelsky definované procedury jsme jako prostředí vzniku předali prostředí primitivních vazeb (navázané na `scheme-toplevel-env`). Seznam argumentů procedury signum jsme vytvořili převedením metaseznamu `(x)` do jeho interní formy. Stejným způsobem jsme zapsali tělo procedury.

Ze vztahu obou prostředí je patrné, že ani v prostředí odvozených definic nemůžeme vytvářet rekurzivní procedury bez použití y-kombinátoru, protože procedura není navázána na symbol v prostředí svého vzniku. Při definici rekurzivních procedur si tedy musíme pomoci y-kombinátorem, viz sekci 9.2. Příklady rekurzivních procedur definovaných v prostředí odvozených definic najdeme v programech 12.26 (výpočet délky seznamu) a 12.27 (mapování přes jeden seznam).

Program 12.26. Procedura `length` v prostředí odvozených definic.

```
(length . ,(make-procedure
            scheme-toplevel-env
            (expr->intern '(l))
            (expr->intern
             '((lambda (y)
                 (y y l))
               (lambda (length l)
                 (if (null? l)
                     0
                     (+ 1 (length length (cdr l)))))))))
```

Poslední věcí, kterou musíme vyřešit, je implementace cyklu REPL, ve kterém poběží samotné vyhodnocování. REPL bude realizován jednoduchou iterativní metaprocedurou, která simuluje činnost vyhodnocovacího cyklu tak, jak jsme jej popsali v sekci 1.5. Viz kód uvedený v programu 12.28. Nejprve je vytvořeno nové prostředí, jehož předkem je prostředí odvozených vazeb. Toto prostředí je navázáno na symbol `init-env`. Dále se opakuje cyklus, ve kterém je vždy načten symbolický výraz, poté je převeden do interní reprezentace, vyhodnocen v prostředí navázaném na `init-env`, výsledek je vytištěn a celý cyklus se opakuje dokud není vyčerpán vstup (nebo nedojde k chybě). Na posledním řádku metaprogramu (interpretu) tedy

Program 12.27. Procedura `map` v prostředí odvozených definic.

```
(map . ,(make-procedure
         scheme-toplevel-env
         (expr->intern '(f l))
         (expr->intern
          '((lambda (y)
              (y y l))
            (lambda (map l)
              (if (null? l)
                  ()
                  (cons (f (car l)) (map map (cdr l)))))))))
```

Program 12.28. Implementace cyklu REPL.

```
(define scm-repl
  (lambda ()
    (let ((init-env (make-env scheme-midlevel-env the-empty-list)))
      (let loop ()
        (display "]=> ")
        (let ((elem (scm-read)))
          (if (not elem)
              'bye-bye
              (let ((result (scm-eval elem init-env)))
                (newline)
                (scm-print result)
                (newline)
                (newline)
                (loop))))))))
```

spustíme metaproceduru `(scm-repl)`, která dále řídí průběh vyhodnocování. Tím jsme završili vývoj první verze našeho interpretu (další vylepšení ukážeme v dalším díle tohoto učebního textu).

Zdrojový kód našeho interpretu (včetně komentářů) nepřesahuje 600 řádků, z pohledu velikosti se tedy jedná o *velmi malý program*. Velké programy běžně přesahují stovky tisíc i miliony řádků. Například jádra operačních systémů mívají kolem pěti milionů řádků, stejně tak kancelářské balíky. Mezi „největší programy“ patří bezpečnostní software pro řízení leteckého provozu a raketové systémy. I přes to, že náš program je pozoruhodně malý, jedná se o implementaci interpretu Turingovsky úplného programovacího jazyka, tedy jazyka, který je z hlediska své vyjadřovací síly stejně silný jako běžně používané programovací jazyky (například C, C++, LISP, Pascal, . . .).

12.7 Příklady použití interpretu

V této sekci ukážeme příklady použití nově vytvořeného interpretu. Příklady budeme komentovat pouze stručně, protože všechny konstrukce jsou již čtenářům důvěrně známé. Výsledné hodnoty zobrazujeme stejně jako je výstup našeho interpretu.

Nejprve ukážeme použití a chování speciální formy `quote`:

```
quote           ⟾  Specform: #<special-form>
(quote blah)    ⟾  Symbol: blah
'blah           ⟾  Symbol: blah
''blah          ⟾  Pair: (quote blah)
```

Další příklad ukazuje kvotování seznamu:

```
(+ 1 2 3)        ⟾  Number: 6
'(+ 1 2 3)       ⟾  Pair: (+ 1 2 3)
(+ 1 (* 2 3))    ⟾  Number: 7
'(+ 1 (* 2 3))   ⟾  (+ 1 (* 2 3))
```

Speciální elementy se vyhodnocují na sebe sama, jako obvykle:

```
()    ⟾  Empty-list: ()
'()   ⟾  Empty-list: ()
#t    ⟾  Boolean: #t
'#t   ⟾  Boolean: #t
#f    ⟾  Boolean: #f
'#f   ⟾  Boolean: #f
```

Speciální formu `if` používáme obvyklým způsobem. Z posledního příkladu je vidět, že `if` se skutečně chová jako speciální forma a nikoliv jako procedura:

```
if                    ⟾  Specform: #<special-form>
(if 1 2 3)            ⟾  Number: 2
(if #f 1 2)           ⟾  Number: 2
(if #f 1)             ⟾  Undefined: #<undefined>
(if #t 1 blah-blah)   ⟾  Number: 1
```

Následující příklad ukazuje použití speciální formy `and`:

```
(and)              ⟾  Boolean: #t
(and 10)           ⟾  Number: 10
(and #f)           ⟾  Boolean: #f
(and 0 2 4)        ⟾  Number: 4
(and 3 (= 1 1))    ⟾  Boolean: #t
```

Pomocí speciální formy `lambda` vytváříme procedury:

```
lambda                       ⟾  Specform: #<special-form>
(lambda (x) (+ x 1))         ⟾  Procedure: #<user-defined-procedure>
((lambda (x) (+ x 1)) 10)    ⟾  Number: 11
```

Náš interpret umožňuje práci se speciálními formami jako s elementy prvního řádu. V následujícím příkladu je speciální forma předána jako argument proceduře. Podobnou konstrukci by nám drtivá většina interpretů jazyka Scheme vůbec neumožnila (speciální formy nejsou v existujících interpretech jazyka Scheme chápány jako elementy prvního řádu).

```
(((lambda (procedura)
    (procedura (x) (+ x 1)))
  lambda)
 10)  ⟾  Number: 11
```

Rekurzivní procedury můžeme definovat pomocí y-kombinátoru, jako třeba:

```
((lambda (y)
   (y y 6))
 (lambda (fak n)
   (if (= n 0)
       1
       (* n (fak fak (- n 1))))))  ⟾  Number: 720
```

Na symbol `map` je navázána uživatelsky definovaná procedura:

```
map                              ⟹  Procedure: #<user-defined-procedure>
(procedure-environment map)      ⟹  Environment: #<environment>
(procedure-arguments map)        ⟹  Pair: (f l)
(procedure-body map)             ⟹  Pair: ((lambda (y) (y y l)) (lambda (map ···
```

Tato uživatelsky definovaná procedura funguje standardně:

```
(map - '(1 2 3 4))                             ⟹  Pair: (-1 -2 -3 -4)
(map (lambda (x) (cons x ())) '(a b c d))      ⟹  Pair: ((a) (b) (c) (d))
(map even? '(0 1 2 3 4 5 6))                   ⟹  Pair: (#t #f #t #f #t #f #t)
(map (lambda (x) (<= x 3)) '(0 1 2 3 4 5 6))   ⟹  Pair: (#t #t #t #t #f #f #f)
```

Následující příklad ukazuje použití `map` a rekurzivní procedury počítající faktoriál:

```
(map
 (lambda (n)
   ((lambda (y)
      (y y n))
    (lambda (fak n)
      (if (= n 0)
          1
          (* n (fak fak (- n 1)))))))
 '(0 1 2 3 4 5 6 7 8 9))  ⟹  Pair: (1 1 2 6 24 120 720 5040 40320 362880)
```

Následující dvě ukázky demonstrují použití libovolných a nepovinných argumentů.

```
((lambda list (map - list)) 1 2 3 4 5 6)          ⟹  Pair: (-1 -2 -3 -4 -5 -6)
((lambda (x y . list)
   (cons x (cons y (map - list)))) 1 2 3 4 5 6)   ⟹  Pair: (1 2 -3 -4 -5 -6)
```

Pomocí `procedure-environment` můžeme získat prostředí vzniku procedury:

```
(procedure-environment
 ((lambda (x)
    (lambda (y) (+ x y)))
  10))  ⟹  Environment: #<environment>
```

Vazby v prostředí můžeme vypsat pomocí `environment->list`:

```
(environment->list
 (procedure-environment
  ((lambda (x)
     (lambda (y) (+ x y)))
   10)))  ⟹  Pair: ((x . 10))
```

V následujícím příkladu je získáno a vypsáno počáteční prostředí (jeho tabulka vazeb je prázdná). V druhém případě je pak vypsána tabulka vazeb v předchůdci aktuálního prostředí, což je prostředí odvozených definic.

```
(the-environment)                      ⟹  Environment: #<environment>
(environment->list (the-environment))  ⟹  Empty-list: ()
(environment->list
 (environment-parent
  (the-environment)))  ⟹  Pair: ((sgn . #<user-defined-procedure>)
                                  (length . #<user-defined-procedure>)
                                  (map . #<user-defined-procedure>))
```

Procedura `eval` je možné používat pouze se dvěma argumenty:

```
(eval '(+ 1 2) (the-environment))  ⟹  Number: 3
```

V následující ukázce je vyhodnocen seznam `(length (quote (a b c)))` ve třech různých prostředích. V posledním případě dojde při vyhodnocení k chybě, protože v prostředí primitivních definic není procedura `length` definovaná.

```
(eval '(length '(a b c)) (the-environment))         ⟹  Number: 3
(eval '(length '(a b c)) (environment-parent
                          (the-environment)))        ⟹  Number: 3
(eval '(length '(a b c)) (environment-parent
                          (environment-parent
                           (the-environment))))     ⟹  Error (symbol not bound)
```

Další příklad ukazuje vyhodnocení seznamu `'(* x x)` v prostředí vzniku procedury:

```
(eval '(* x x)
      (procedure-environment
       ((lambda (x)
          (lambda (y) (+ x y)))
        10)))   ⟹  Number: 100
```

Proceduru `apply` je možné použít i s více jak dvěma argumenty:

```
(apply + '(1 2 3 4))             ⟹  Number: 10
(apply + 1 2 '(3 4))             ⟹  Number: 10
(apply + 1 2 3 4 '())            ⟹  Number: 10
(apply map - '((1 2 3 4)))       ⟹  Pair: (-1 -2 -3 -4)
(apply map - '(1 2 3 4) '())     ⟹  Pair: (-1 -2 -3 -4)
```

V dalším příkladu využijeme faktu, že na symbol `pi` máme navázanou hodnotu čísla π:

```
pi  ⟹  Number: 3.141592653589793
```

Při vyhodnocení následujícího výrazu nehraje globální vazba `pi` roli, protože interpret používá lexikální rozsah platnosti:

```
(((lambda (pi)
    (lambda (x)
      (+ x pi)))
  10)
 20)  ⟹  Number: 30
```

Totéž platí pro explicitní aplikaci:

```
(apply ((lambda (pi)
          (lambda (x)
            (+ x pi)))
        10)
       20
       '())  ⟹  Number: 30
```

V následující ukázce jsme provedli aplikaci uživatelsky definované procedury, přitom jsme „dočasně nastavili“ předka této procedury na lokální prostředí v němž je na `y` navázaná hodnota `100`:

```
(env-apply (lambda (x) (+ x y))
           ((lambda (y) (the-environment)) 100)
           20 '())  ⟹  Number: 120
```

Shrnutí

Zabývali jsme se automatickým přetypováním a generickými procedurami. Pro generické procedury jsme zavedli jednoduchou metodu jejich aplikace prostřednictvím vyhledávání metod pomocí vzorů uvedených v tabulkách. Na konkrétním příkladu generických procedur jsme ukázali jejich praktické použití. Dále jsme se seznámili s konceptem manifestovaných typů. Pomocí manifestovaných typů jsme implementovali reprezentaci elementů jazyka Scheme. S jejich využitím jsme dále naprogramovali vyhodnocovací proces. Nakonec jsme zkompletovali interpret čistě funkcionální podmnožiny jazyka Scheme.

Pojmy k zapamatování

- přetypování, implicitní/explicitní přetypování,
- automatické přetypování, koerce,
- generická procedura, tabulka generických procedur,
- manifestované typy, visačky, tagy,
- metajazyk, metainterpret, metaelement, metaprocedura.

Nově představené prvky jazyka Scheme

- procedury `number->string` a `string-append`

Kontrolní otázky

1. *Co jsou to generické procedury?*
2. *Jaké jsou výhody a nevýhody automatického přetypování?*
3. *Co jsou to manifestované typy?*
4. *Jak jsme v naší implementaci jazyka Scheme reprezentovali jednotlivé elementy jazyka?*
5. *Kolik jsme uvažovali počátečních prostředí a proč?*
6. *Jaký je rozdíl mezi jazykem/interpretem a metajazykem/metainterpretem?*

Cvičení

1. Bez použití interpretu jazyka Scheme zjistěte, jak se vyhodnotí následující výrazy používající generické `+` tak, jak jsme jej představili v sekci 12.1:
```
(+ 2/3 "." 0.5)                        ⟹
(+ (- 1) "x" (- 2))                    ⟹
(+ 1 2 "+" 3 4)                        ⟹
(+ 2 (+ 2 "3") 10)                     ⟹
(+ 2 (log (exp 2)))                    ⟹
(apply + '("a" "b" (+ 1 2) 2))         ⟹
(apply + (map number->string '(1 2 3))) ⟹
```
2. Obohaťte vytvořený interpret jazyka Scheme o primitivní proceduru `foldr`. To jest `foldr` by měla být ve vytvořeném interpretu k dispozici přímo v počátečním prostředí a mělo by se jednat o primitivní proceduru, tedy nikoliv o uživatelsky definovanou proceduru. Naprogramujte tuto primitivní proceduru tak, aby mohla pracovat s libovolným množstvím seznamů (vždy však alespoň s jedním).
3. Analogicky proveďte obohacení interpretu o primitivní proceduru `foldl`.
4. Obohaťte vytvořený interpret jazyka Scheme o speciální formu `or`. Tuto speciální formu naprogramujte tak, aby se chovala přesně jako speciální forma `or` v jazyku Scheme, viz [R5RS].
5. Obohaťte vytvořený interpret jazyka Scheme o speciální formu `let` (nepojmenovanou verzi).
6. Obohaťte vytvořený interpret jazyka Scheme o pojmenovaný `let`. Jelikož v jazyku nemáme k dispozici `define`, musíme si při programování pojmenovaného `let` pomocí y-kombinátoru. Speciální formu ale vytvořte tak, abychom při rekurzivní aplikaci nemuseli předávat proceduru prostřednictvím jejího prvního argumentu. Vytvořený pojmenovaný `let` by se tedy z uživatelského pohledu měl chovat jako pojmenovaný `let` popsaný ve [R5RS].
7. Obohaťte interpret jazyka Scheme o speciální formu `dyn-apply`, která bude provádět aplikaci procedur podobně jako `apply`, ale při aplikaci procedur pomocí `dyn-apply` se budou při vyhodnocování těla procedury hledat vazby symbolů ve smyslu dynamického rozsahu platnosti, tedy v prostředí aplikace procedury. Na následujícím příkladu je vidět rozdíl použití `apply` a `dyn-apply`:

```
((lambda (x)
   (apply ((lambda (x)
             (lambda (y)
               (+ x y))) 100)
          '(20)))
 1000)  ⟾  120

((lambda (x)
   (dyn-apply ((lambda (x)
                 (lambda (y)
                   (+ x y))) 100)
              '(20)))
   1000)  ⟾  1020
```

8. Obohaťte interpret o novou speciální formu `delta`, která je po syntaktické stránce shodná s formou `lambda`. Vyhodnocením δ-výrazu vznikne speciální typ uživatelsky definované procedury. Při aplikaci procedury vzniklé vyhodnocením δ-výrazů je uplatněn dynamický rozsah platnosti. Zavedení δ-výrazů do jazyka nám tedy umožní vytvářet uživatelsky definované procedury, jejichž vyhodnocování probíhá v souladu s dynamickým rozsahem platnosti. Viz příklad použití a rozdíl v použití speciálních forem `lambda` a `delta`.

```
((lambda (x)
   (((lambda (x)
       (lambda (y)
         (+ x y))) 100) 20)) 1000)  ⟾  120

((lambda (x)
   (((lambda (x)
       (delta (y)
         (+ x y))) 100) 20)) 1000)  ⟾  1020
```

Při implementaci využijte svých znalostí ze sekce 2.6.

9. Napište, jak interpret Scheme vyhodnotí následující symbolické výrazy:

```
(environment->list
 ((lambda (x) (the-environment)) 10))  ⟾

((lambda (x)
   (eval '(- x) (the-environment)))
 100)  ⟾

(eval '(+ x 1)
      ((lambda (x) (the-environment)) 10))  ⟾

((lambda (x)
   ((lambda (x)
      (eval '(cons x (quote x)) (procedure-environment x)))
    (lambda (x) 'blah)))
 1000)  ⟾

(environment->list
 (procedure-environment
  ((lambda (y)
     (lambda (x)
       (+ x 1)))
   100)))  ⟾
```

Úkoly k textu

1. Vytvořte systém generických procedur `+`, `-`, `*`, `modulo`, `quotient` a `=` sloužících pro práci s čísly a polynomy. Polynomy (jedné proměnné) reprezentujte seznamy jejich koeficientů tak, jak jsme to dělali v sekci 8.6. Základní aritmetické procedury `+`, `-`, `*` slouží k provádění aritmetických operací s čísly a polynomy. Například tedy pro `*` by to mělo pokrývat součin čísel, součin polynomů a součin čísla s polynomem. Procedury `modulo` a `quotient` by měly vracet podíl čísel nebo polynomů a zbytek po dělení číslem nebo polynomem. Predikát `=` by měl sloužit k porovnávání čísel a polynomů. Při implementaci si pomozte tím, že čísla jsou de facto konstantní polynomy. Kde lze, provádějte automatické přetypování.

2. Po dokončení předchozího úkolu naprogramujte procedury pro sčítání a násobení matic. Pokud jsme provedli správnou implementaci v předchozím bodě, měla by vaše implementace operací s maticemi umožňovat pracovat s maticemi jejichž prvky jsou čísla nebo polynomy. Důkladně vaši implementaci otestujte.

3. Obohaťte interpret jazyka Scheme o speciální formy `cond` a `let*`. Obě speciální formy naprogramujte tak, aby se chovaly stejně, jak jsme je představili v lekcích 2 a 3. Implementace obou speciálních forem důkladně otestujte a přesvědčte se o jejich správnosti.

4. Obohaťte interpret jazyka Scheme o speciální formu `named-lambda`, která je po syntaktické stránce totožná se speciální formou `lambda`. Speciální forma `named-lambda` slouží k vytváření procedur stejně jako speciální forma `lambda`, navíc však v těle procedury vytvořené pomocí `named-lambda` je vždy na symbol `self` navázaná samotná procedura. Pomocí `self` je tedy možné provádět rekurzivní volání, viz následující příklad použití.

```
(map (named-lambda (n)
       (if (= n 0)
           1
           (* n (self (- n 1)))))
     '(0 1 2 3 4 5 6 7 8))  ⟹  (1 1 2 6 24 120 720 5040 40320)
```

5. V předchozích příkladech na procvičení jsme viděli dva přístupy k aplikaci procedur řídící se dynamickým rozsahem platnosti. První metodou bylo použití explicitní aplikace pomocí `dyn-apply`. Druhým způsobem bylo zavedení δ-výrazů, pomocí nichž vznikaly procedury aplikované vždy ve smyslu dynamického rozsahu platnosti. Ani jedno z těchto řešení není úplně ideální, protože v některých případech by se mohlo hodit, abychom v těle jedné procedury uvažovali většinu symbolů ve smyslu lexikálního rozsahu platnosti (zajímat se budeme o lexikální vazby symbolů) a některé symboly ve smyslu dynamického rozsahu platnosti (zajímat se budeme o dynamické vazby symbolů).

   Obohaťte interpret o speciální formu `dynamic`, jejímž jediným argumentem bude symbol. Výsledkem vyhodnocení `(dynamic x)` (v daném prostředí) je dynamická vazba symbolu `x`, tedy vazba, která je hledána v dynamicky nadřazených prostředích (pokud není nalezena v lokálním prostředí), viz sekci 2.6. Následující ukázka demonstruje použití `dynamic`.

```
((lambda (x)
   (((lambda (x)
       (lambda (y)
         (+ x y))) 100) 20)) 1000)  ⟹  120

((lambda (x)
   (((lambda (x)
       (lambda (y)
         (+ (dynamic x) y))) 100) 20)) 1000)  ⟹  1020
```

Řešení ke cvičením

1. "2/3.0.5","-1x-2","12+7","22310",4.0,Error: No method for these types,"123"
2. Vytvoříme pomocnou proceduru `scm-foldr` následovně:
```
(define scm-foldr
  (lambda (f basis . lists)
    (if (scm-null? (car lists))
        basis
        (scm-apply
         f
         (append (map pair-car lists)
                 (list (apply
                        scm-foldr
                        f basis (map pair-cdr lists))))))))
```
   Dále je nutné do tabulky vazeb prostředí „toplevel“ přidat záznam:
```
(foldr . ,(make-primitive scm-foldr))
```
3. Vytvoříme pomocnou proceduru `scm-foldl` následovně:
```
(define scm-foldl
  (lambda (f basis . lists)
    (let iter ((lists lists)
               (accum basis))
      (if (scm-null? (car lists))
          accum
          (iter (map pair-cdr lists)
                (scm-apply f (append (map pair-car lists)
                                     (list accum))))))))
```
   Dále je nutné do tabulky vazeb prostředí „toplevel“ přidat záznam:
```
(foldl . ,(make-primitive scm-foldl))
```
4. Vytvoříme pomocnou proceduru `scm-or` následovně:
```
(define scm-or
  (lambda (env . exprs)
    (let or-eval ((exprs exprs))
      (cond ((null? exprs) scm-false)
            ((null? (cdr exprs)) (scm-eval (car exprs) env))
            (else (let ((result (scm-eval (car exprs) env)))
                    (if (equal? result scm-false)
                        (or-eval (cdr exprs))
                        result)))))))
```
   Dále je nutné do tabulky vazeb prostředí „toplevel“ přidat záznam:
```
(or . ,(make-specform scm-or))
```
5. Vytvoříme pomocnou proceduru `map-scm-list` pro mapování přes seznam ve vnitřní reprezentaci:
```
(define map-scm-list
  (lambda (f l)
    (convert-list scm-null? pair-car pair-cdr
                  make-pair the-empty-list f l)))
```
   Dále vytvoříme pomocnou proceduru `scm-let`:
```
(define scm-let
  (lambda (env bindings body)
    (let ((proc (make-procedure
                 env
                 (map-scm-list pair-car bindings)
                 body)))
      (scm-apply proc
                 (map-scm-list->list
```

```
                    (lambda (elem)
                      (scm-eval (pair-car (pair-cdr elem)) env))
                    bindings)))))
```

Dále je nutné do tabulky vazeb prostředí „toplevel" přidat záznam:

```
(let . ,(make-specform scm-let))
```

6. Vytvoříme nejprve pomocný konstruktor seznamu v interní reprezentaci:

```
(define make-list
  (lambda args
    (if (null? args)
        the-empty-list
        (make-pair (car args)
                   (apply make-list
                          (cdr args))))))
```

Pro programování pomocné procedury, která bude v konstruovaném interpretu realizovat speciální formu „pojmenovaný `let`" si musíme uvědomit, že kód ve tvaru

```
(let ⟨jméno⟩ ((⟨symbol₁⟩ ⟨výraz₁⟩)
             (⟨symbol₂⟩ ⟨výraz₂⟩)
               ⋮
             (⟨symbolₙ⟩ ⟨výrazₙ⟩))
  ⟨tělo⟩)
```

stačí nahradit kódem ve tvaru

```
((lambda (y)
   (y y ⟨výraz₁⟩ ⟨výraz₂⟩ ··· ⟨výrazₙ⟩))
 (lambda (⟨jméno⟩ ⟨symbol₁⟩ ⟨symbol₂⟩ ··· ⟨symbolₙ⟩)
   ((lambda (⟨jméno⟩) ⟨tělo⟩)
    (lambda (⟨symbol₁⟩ ⟨symbol₂⟩ ··· ⟨symbolₙ⟩)
      (⟨jméno⟩ ⟨jméno⟩ ⟨symbol₁⟩ ⟨symbol₂⟩ ··· ⟨symbolₙ⟩)))))
```

Podle tohoto postupu naprogramujeme pomocnou proceduru `scm-named-let`:

```
(define scm-named-let
  (lambda (env . args)
    (if (not (scm-symbol? (car args)))
        (apply scm-let env args)
        (let* ((name (car args))
               (bindings (cadr args))
               (body (caddr args))
               (y (make-symbol 'y))
               (y-comb (make-procedure
                        env
                        (make-list y)
                        (make-pair y
                          (make-pair y
                            (map-scm-list
                             (lambda (elem)
                               (scm-eval (pair-car (pair-cdr elem)) env))
                             bindings)))))
               (proc (make-procedure
                      env
                      (make-pair
                       name
                       (map-scm-list pair-car bindings))
                      (make-list
                       (make-list (make-symbol 'lambda)
                                  (make-list name)
                                  body)
```

```
                         (make-list (make-symbol 'lambda)
                                    (map-scm-list pair-car bindings)
                                    (make-pair name
                                      (make-pair name
                                          (map-scm-list pair-car bindings)))))))
          (scm-apply y-comb (list proc))))))
```

Dále je nutné do tabulky vazeb prostředí „toplevel" přidat záznam:

```
(let . ,(make-specform scm-named-let))
```

7. Vytvoříme pomocnou proceduru `scm-dyn-apply` následovně:

```
(define scm-dyn-apply
  (lambda (env proc . rest)
    (scm-eval
     (make-pair (make-symbol 'env-apply)
                (make-pair
                 proc
                 (make-pair
                  env
                  (let iter ((ar rest))
                    (if (null? (cdr ar))
                        (make-pair (car ar) the-empty-list)
                        (make-pair (car ar)
                                   (iter (cdr ar))))))))
     env)))
```

Dále je nutné do tabulky vazeb prostředí „toplevel" přidat záznam:

```
(dyn-apply . ,(make-specform scm-dyn-apply))
```

8. Nejprve vytvoříme datovou reprezentaci nového typu elementu:

```
(define make-dyn-procedure
  (let ((make-physical-procedure (curry-make-elem 'dyn-procedure)))
    (lambda (args body)
      (make-physical-procedure (list args body)))))

(define dyn-procedure-arguments (lambda (proc) (car (get-data proc))))
(define dyn-procedure-body (lambda (proc) (cadr (get-data proc))))

(define scm-user-dyn-procedure? (curry-scm-type 'dyn-procedure))
```

Upravíme `scm-procedure?` tak, aby brala ohled i na nový typ procedury:

```
(define scm-procedure?
  (lambda (elem)
    (or (scm-primitive? elem)
        (scm-user-procedure? elem)
        (scm-user-dyn-procedure? elem))))
```

Dále přidáme do tabulky vazeb prostředí „toplevel" záznam:

```
(delta . ,(make-specform
           (lambda (env args body)
             (make-dyn-procedure args body))))
```

Upravíme `scm-eval` tak, aby se při volání `scm-apply` předávalo aktuální prostředí jako nový poslední argument. Kritický fragment kódu by měl vypadat takto:

```
  ⋮
((scm-procedure? f)
 (scm-apply f
            (map-scm-list->list
             (lambda (elem)
               (scm-eval elem env))
             args)
```

```
               env))
                 ⋮
```

Dále upravíme `scm-env-apply` a `scm-apply` následovně tak, že do nich přidáme větve ošetřující průběh aplikaci v případě procedur vzniklých vyhodnocením δ-výrazů.

```
(define scm-env-apply
  (lambda (proc env args)
    (cond ((scm-primitive? proc) (apply (get-data proc) args))
          ((scm-user-procedure? proc)
           (scm-eval (procedure-body proc)
                     (make-env env
                               (make-bindings (procedure-arguments proc)
                                              args))))
          ((scm-user-dyn-procedure? proc)
           (scm-eval (dyn-procedure-body proc)
                     (make-env env
                               (make-bindings (dyn-procedure-arguments proc)
                                              args))))
          (else (error "APPLY: Expected procedure")))))

(define scm-apply
  (lambda (proc args . env)
    (cond ((scm-primitive? proc) (scm-env-apply proc #f args))
          ((scm-user-procedure? proc)
           (scm-env-apply proc (procedure-environment proc) args))
          ((scm-user-dyn-procedure? proc)
           (scm-env-apply proc (car env) args))
          (else (error "APPLY: Expected procedure")))))
```

9. `((x . 10))`, `-100`, `11`, `(1000 . x)`, `((y . 100))`

Reference

[SICP] Abelson H., Sussman G. J.: *Structure and Interpretation of Computer Programs.*
http://mitpress.mit.edu/sicp/full-text/book/book.html
The MIT Press, Cambridge, Massachusetts, 2nd edition, 1986. ISBN 0–262–01153–0.

[BW88] Bird R., Wadler P.: *Introduction to Functional Programming.*
Prentice Hall, Englewood Cliffs, New Jersey, 1988. ISBN 0–13–484197–2.

[Ch36] Church A.: An unsolvable problem of elementary number theory.
American Journal of Mathematics **58**(1936), 345–363.

[Ch41] Church A.: *The Calculi of Lambda Conversion.*
Princeton University Press, Princeton (NJ, USA), 1941.

[Di68] Dijkstra E. W.: Go To Statement Considered Harmful.
Communications of the ACM **11**(3)(1968), 147–148.

[Dy96] Dybvig R. K.: *The Scheme Programming Language.*
http://www.scheme.com/tspl3/
The MIT Press, Cambridge, Massachusetts, 3rd edition, 2003. ISBN 0–262–54148–3.

[F3K01] Felleisen M., Findler R. B., Flatt M., Krishnamurthi S.:
How to Design Programs: An Introduction to Computing and Programming.
http://www.htdp.org/
The MIT Press, Cambridge, Massachusetts, 2001.

[FF96a] Friedman D. P., Felleisen M.: *The Little Schemer.*
MIT Press, 4th edition, 1996.

[FF96b] Friedman D. P., Felleisen M.: *The Seasoned Schemer.*
MIT Press, 1996.

[Gi97] Giloi W. K.: Konrad Zuse's Plankalkül: The First High-Level "non von Neumann" Programming Language. *IEEE Annals of the History of Computing* **19**(2)1997, 17–24.

[Ho01] Hopcroft J. E. a kol.: *Introduction to Automata Theory, Languages, and Computation,*
Addison-Wesley, 2001.

[R5RS] Kelsey R., Clinger W., Rees J. (editoři):
Revised^5 Report on the Algorithmic Language Scheme,
http://www.schemers.org/Documents/Standards/R5RS/
Higher-Order and Symbolic Computation **11**(1)1998, 7–105;
ACM SIGPLAN Notices **33**(9)1998, 26–76.

[KV08] Klir G. J., Vysocký P.: *Počítače z Loretánského náměstí: Život a dílo Antonína Svobody.*
Nakladatelství ČVUT, Praha, 2008. ISBN 978–80–01–03953–3.

[Ko97] Kozen D. C.: *Automata and Computability,*
Springer, 1997

[Kr06] Krishnamurthi S.: *Programming Languages: Application and Interpretation.*
http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/

[Ll87] Lloyd, J. W.: *Foundations of Logic Programming*.
Springer-Verlag, New York, 2nd edition, 1987.

[ML95] Manis V. S., Little J. J.: *The Schematics of Computation.*
Prentice Hall, Englewood Cliffs, New Jersey, 1995. ISBN 0–13–834284–9.

[MC60] McCarthy J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine.
Communications of the ACM **3**(4)1960, 184–195.

[MC67] McCarthy J.: A Basis for a Mathematical Theory of Computation.
Obsaženo ve: Braffort P., Hirschberg D. (editoři), *Computer Programming and Formal Systems.*
North-Holland, 1967.

[vN46] von Neumann J.: The principles of large-scale computing machines.
Ann. Hist. Comp **3**(3)1946.

[NS97] Nerode A., Shore A. R.: *Logic for Applications.*
Springer-Verlag, New York, 2nd edition, 1997.

[NM00] Nilson U., Maluszynski J.: *Logic, programming and Prolog.*
http://www.ida.liu.se/~ulfni/lpp/

[PeSa58] Perlis A. J., Samelson K.: Preliminary Report – International Algorithmic Language.
Communications of the ACM **1**(12)1958, 8–22.

[Pe81] Perlis A. J.: The American side of the development of ALGOL.
Obsaženo ve: Wexelblat R. L. (editor): *History of Programming Languages.* Academic Press, 1981.

[Qu96] Queinnec C.: *Lisp in Small Pieces.*
Cambridge University Press, 1996.

[Ro63] Robinson J. A.: Theorem-Proving on the Computer.
Journal of the ACM **10**(2)(1963), 163–174.

[Ro65] Robinson J. A.: A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the ACM **12**(1)(1965), 24–41.

[Ro00] Rojas R. a kol.: *Plankalkül: The First High-Level Programming Language and its Implementation*.
http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm
Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000.

[SS86] Shapiro E., Sterling S.: *The Art of Prolog.*
The MIT Press, Cambridge, Massachusetts, 1986.

[Si04] Sitaram D.: *Teach Yourself Scheme in Fixnum Days.*
http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html
text je autorem postupně aktualizován, 1998–2004.

[R6RS] Sperber M., Dybvig R., Flatt M., Van Straaten A., Findler R., Matthews J.:
Revised^6 Report on the Algorithmic Language Scheme,
http://www.r6rs.org/
Journal of Functional Programming **19**(S1)2009, 1–301.

[SF94] Springer G., Friedman D. P.: *Scheme and the Art of Programming.*
The MIT Press, Cambridge, Massachusetts, 1994. ISBN 0–262–19288–8.

[St76] Steele G. L.: *LAMBDA: The Ultimate Declarative*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-379.pdf
AI Memo **379**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1976.

[SS76] Steele G. L., Sussman G. J.: *LAMBDA: The Ultimate Imperative*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-353.pdf
AI Memo **353**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1976.

[SS78] Steele G. L., Sussman G. J.: *The Revised Report on SCHEME: A Dialect of LISP.*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-452.pdf
AI Memo **452**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1978.

[SS75] Sussman G. J., Steele G. L.: *SCHEME: An Interpreter for Extended Lambda Calculus.*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-349.pdf
AI Memo **349**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1975.

[Wec92] Wechler W.: *Universal Algebra for Computer Scientists.*
Springer-Verlag, Berlin Heidelberg, 1992.

[Zu43] Zuse K.: *Ansätze einer Theorie des allgemeinen Rechnens unter besonderer Berücksichtigung des Aussagenkalküls und dessen Anwendung auf Relaisschaltungen*.
Nepublikovaný rukopis, Zuse Papers 045/018, 1943.

[Zu72] Zuse K.: Der Plankalkül.
http://www.zib.de/zuse/English_Version/Inhalt/Texte/Chrono/40er/Pdf/0233.pdf
Berichte der Gesellschaft für Mathematik und Datenverarbeitung. Nr. 63, Sankt Augustin, 1972.

A Seznam vybraných programů

2.1 Výpočet délky přepony v pravoúhlém trojúhelníku . . . 51
2.2 Infixová aplikace procedury dvou argumentů (procedura `infix`) . . . 53
2.3 Rozložení procedury na dvě procedury jednoho argumentu (procedura `curry+`) . . . 53
2.4 Vytáření nových pocedur posunem a násobením . . . 59
2.5 Vytváření procedur reprezentujících polynomické funkce . . . 60
2.6 Kompozice dvou procedur (procedura `compose2`) . . . 61
2.7 Přibližná směrnice tečny a přibližná derivace . . . 62
2.8 Procedura negace (procedura `not`) . . . 65
2.9 Predikáty sudých a lichých čísel (procedury `even?` a `odd?`) . . . 65
2.10 Minumum a maximum ze dvou prvků (procedury `min` a `max`) . . . 69
2.11 Hledání extrémních hodnot (procedura `extrem`) . . . 69
3.1 Procedura `dostrel` s lokálními vazbami vytvářenými s využitím `and`. . . . 87
3.2 Procedura `dostrel` s lokálními vazbami vytvářenými pomocí speciální formy `define`. . . . 88
3.3 Procedura `derivace` s použitím interní definice. . . . 91
3.4 Procedura `derivace` s použitím speciální formy `let`. . . . 91
4.1 Příklad abstrakční bariéry: výpočet kořenů kvadratické rovnice. . . . 104
4.2 Implementace procedury `koreny` pomocí procedur vyšších řádů . . . 105
4.3 Procedury `cons`, `car` a `cdr` (implementace tečkových párů pomocí procedur vyšších řádů) . . . 107
5.1 Obracení seznamu pomocí `build-list` (procedura `reverse`) . . . 123
5.2 Spojení dvou seznamů pomocí `build-list` (procedura `append2`) . . . 124
5.3 Mapování přes jeden seznam pomocí `build-list` (procedura `map1`) . . . 126
6.1 Délka seznamu (procedura `length`) . . . 145
6.2 Filtrace prvků seznam (procedura `filter`) . . . 147
6.3 Procedury `remove` a `member?` . . . 147
6.4 Vrácení prvku na dané pozici (procedura `list-ref`) . . . 147
6.5 Vrácení pozic výskytu prvku (procedura `list-indices`) . . . 148
6.6 Test přítomnosti prvku v seznamu s navrácením příznaku (procedura `find`) . . . 150
6.7 Součet druhých mocnin . . . 151
6.8 Konstruktor seznamu (procedura `list`) . . . 151
7.1 Délka seznamu pomocí `foldr` (procedura `length`) . . . 174
7.2 Spojení dvou seznamů pomocí `foldr` (procedura `append2`) . . . 175
7.3 Spojení seznamů pomocí `foldr` (procedura `append`) . . . 176
7.4 Mapování přes jeden seznam pomocí `foldr` (procedura `map1`) . . . 176
7.5 Mapování přes libovolné seznamy pomocí `foldr` (procedura `map`) . . . 177
7.6 Filtrace prvků seznam pomocí `foldr` (procedura `filter`) . . . 178
7.7 Test přítomnosti prvku v seznamu pomocí `foldr` (procedura `member?`) . . . 178
7.8 Nahrazování prvků v seznamu pomocí `foldr` (procedura `replace`) . . . 178
7.9 Procedury `genuine-foldl` a `foldl` pomocí `foldr` a reverze seznamu . . . 183
7.10 Složení libovolného množství procedur pomocí `foldl` (procedura `compose`) . . . 184
7.11 Procedury `genuine-foldl` a `foldl` pro libovolný počet seznamů . . . 185
7.12 Výpočet faktoriálu pomocí procedur vyšších řádů (procedura `fac`) . . . 187
7.13 Výpočet prvků Fibinacciho posloupnosti pomocí procedur vyšších řádů (procedura `fib`) . . . 187
8.1 Rekurzivní procedura počítající x^n (procedura `expt`) . . . 200

8.2 Rychlá rekurzivní procedura počítající x^n (procedura expt) 203
8.3 Rekurzivní výpočet faktoriálu (procedura fac) 206
8.4 Rekurzivní výpočet prvků Fibonacciho posloupnosti (procedura fib) 206
8.5 Iterativní faktoriál (procedura fac) 208
8.6 Iterativní faktoriál s interní definicí (procedura fac) 212
8.7 Iterativní verze expt 212
8.8 Iterativní mocnění s pomocí zásobníku (procedura expt) 213
8.9 Iterativní Fibonacciho čísla (procedura fib) 217
8.10 Iterativní Fibonacciho čísla s interní definicí (procedura fib) 218
8.11 Iterativní faktoriál používající pojmenovaný let (procedura fac) 220
8.12 Iterativní Fibonacciho čísla používající pojmenovaný let (procedura fib) 220
8.13 Délka seznamu pomocí rekurze (procedura length) 221
8.14 Délka seznamu pomocí iterace (procedura length) 221
8.15 Spojení dvou seznamů pomocí rekurze (procedura append2) 222
8.16 Vrácení prvku na dané pozici pomocí rekurze (procedura list-ref) 222
8.17 Vrácení seznamu bez prvních prvků (procedura list-tail) 223
8.18 Mapování přes jeden seznam pomocí rekurze (procedura map1) 223
8.19 Rekurzivní verze vytváření seznamů (procedura build-list) 223
8.20 Neefektivní verze vytváření seznamů (procedura build-list) 224
8.21 Iterativní verze vytváření seznamů (procedura build-list) 224
8.22 Neefektivní reverze seznamu (procedura reverse) 225
8.23 Iterativní reverze seznamu (procedura reverse) 225
9.1 Rekurzivní výpočet faktoriálu pomocí y-kombinátoru 245
9.2 Spojovaní seznamů append naprogramované rekurzivně 249
9.3 Spojovaní seznamů append naprogramované rekurzivně bez použití pomocné procedury . . 249
9.4 Skládání funkcí compose bez použití procedury foldl 250
9.5 Implementace procedury hloubkového nahrazovaní atomů depth-replace 253
10.1 Prohledávání stromu do hloubky 261
10.2 Prohledávání stromu do šířky 261
10.3 Výpočet potenční množiny 265
10.4 Efektivnější výpočet potenční množiny 265
10.5 Výpočet všech permutací prvků množiny 266
10.6 Výpočet permutace prvků množiny pomocí faktoradických čísel 268
10.7 Výpočet všech kombinací prvků množiny 268
10.8 Výpočet všech kombinací s opakováním 269
11.1 Procedura pro zjednodušování aritmetických výrazů 273
11.2 Interní definice v proceduře pro zjednodušování výrazů 274
11.3 Tabulka procedur pro zjednodušování výrazů 274
11.4 Procedura assoc pro vyhledávání v asociačním seznamu 275
11.5 Vylepšená procedura pro zjednodušování aritmetických výrazů 276
11.6 Procedura pro symbolickou derivaci 278
11.7 Procedura prefix->postfix pro převod výrazů do postfixové notace 281
11.8 Procedura prefix->polish pro převod výrazů do postfixové bezzávorkové notace 281
11.9 Procedura prefix->infix pro převod výrazů do infixové notace 282

11.10 Procedura postfix-eval vyhodnocující postfixové výrazy . . . 283
11.11 Jednoduchá tabulka s prostředím vazeb pro postfixový evaluátor . . . 284
11.12 Procedura polish-eval vyhodnocující výrazy v polské notaci . . . 286
12.1 Procedura match-type? (porovnávání datového typu se vzorem) . . . 291
12.2 Konkrétní tabulka metod generické procedury . . . 292
12.3 Procedura table-lookup (vyhledání operace v tabulce metod generické procedury) . . . 292
12.4 Procedura apply-generic (aplikace generické procedury) . . . 292
12.5 Generická procedura pro sčítání . . . 293
12.6 Procedury pro práci s manifestovanými typy . . . 294
12.7 Reprezentace tečkových párů . . . 297
12.8 Reprezentace prostředí . . . 299
12.9 Konstruktor pro globální prostředí . . . 300
12.10 Vyhledávání vazeb v prostředí . . . 301
12.11 Reprezentace uživatelsky definovaných procedur . . . 301
12.12 Převod do interní reprezentace a implementace readeru . . . 302
12.13 Převod do externí reprezentace Implementace printeru . . . 303
12.14 Reprezentace primitivních procedur . . . 303
12.15 Obecný konvertor seznamu na seznam ve vnitřní reprezentaci a zpět . . . 304
12.16 Konverze seznamů na metaseznamy o obráceně . . . 304
12.17 Implementace vlastního vyhodnocovacího procesu . . . 305
12.18 Tabulka vazeb mezi formálními/skutečnými argumenty (procedura make-bindings) . . . 307
12.19 Implementace aplikace procedur. . . . 307
12.20 Definice speciální formy if v globálním prostředí . . . 308
12.21 Definice speciální formy and v globálním prostředí . . . 309
12.22 Definice forem lambda, the-environment a quote v globálním prostředí . . . 310
12.23 Definice selektorů uživatelsky definovaných procedur v globálním prostředí . . . 310
12.24 Procedura apply-collect-arguments (sestavení seznamu argumentů pro apply) . . . 310
12.25 Definice eval, apply and env-apply v globálním prostředí . . . 311
12.26 Procedura length v prostředí odvozených definic . . . 312
12.27 Procedura map v prostředí odvozených definic . . . 313
12.28 Procedura scm-repl (implementace cyklu REPL) . . . 313

B Seznam obrázků

1.1 Výpočetní proces jako abstraktní entita . . . 8
1.2 Schéma cyklu REPL . . . 25
1.3 Prostředí jako tabulka vazeb mezi symboly a elementy . . . 26
2.1 Prostředí a jejich hierarchie . . . 48
2.2 Vznik prostředí během aplikace procedur z programu 2.1 . . . 52
2.3 Vznik prostředí během aplikace procedur z programu 2.3 . . . 54
2.4 Vznik prostředí během aplikace procedur z programu 2.3 . . . 55
2.5 Vyjádření funkcí pomocí posunu a násobení funkčních hodnot. . . . 59
2.6 Různé polynomické funkce, skládání funkcí a derivace funkce. . . . 60
3.1 Šikmý vrh ve vakuu . . . 79
3.2 Vznik prostředí během vyhodnocení programu . . . 82
3.3 Vznik prostředí během vyhodnocení programu z příkladu 3.5 . . . 83
3.4 Hierarchie prostředí . . . 85
4.1 Boxová notace tečkového páru. . . . 101
4.2 Tečkové páry z příkladu 4.8 v boxové notaci . . . 101
4.3 Schéma abstrakčních bariér . . . 104
4.4 Vznik prostředí při aplikaci procedury z příkladu 4.2 . . . 105
4.5 Prostředí vznikající při použití vlastní implementace párů . . . 107
4.6 Vrstvy v implementaci racionální aritmetiky . . . 110
4.7 Boxová notace tečkových párů – zadání ke cvičení . . . 112
5.1 Boxová notace tečkového páru používající ukazatel . . . 118
5.2 Seznamy z příkladu 5.4 v boxové notaci . . . 118
5.3 Program `(define 1+ (lambda (x) (+ x 1)))` jako data. . . . 119
5.4 Procedury a prostředí u párů uchovávajících délku seznamu . . . 130
8.1 Schématické zachycení úvahy o spojení dvou seznamů . . . 196
8.2 Schématické zachycení aplikace procedury `expt`. . . . 201
8.3 Prostředí vzniklá během vyhodnocení `(expt 8 4)` . . . 202
8.4 Schématické zachycení aplikace rychlé procedury `expt`. . . . 204
8.5 Schématické zachycení aplikace rekurzivní verze `fac`. . . . 207
8.6 Schématické zachycení aplikace iterativní verze `fac`. . . . 208
8.7 Schématické zachycení iterativní verze procedury `expt`. . . . 213
8.8 Schématické zachycení aplikace `expt` vytvořené s využitím zásobníku. . . . 214
8.9 Schématické zachycení aplikace rekurzivní verze `fib`. . . . 215
8.10 Postupné provádění aplikací při použití rekurzivní verze `fib`. . . . 215
8.11 Schématické zachycení aplikace iterativní verze `fib`. . . . 217
8.12 Schématické zachycení aplikace iterativní verze `length`. . . . 222
10.1 Příklad n-árního stromu . . . 259
10.2 Ukázka průchodu do šířky a do hloubky . . . 260
10.3 Výsledek aplikace stare a vylepšené verze `power-set` . . . 266
10.4 Faktoradická čísla a permutace . . . 267
11.1 Struktura výrazu `(+ (* 2 x) (- (/ (+ x 2) z)) 5)` . . . 280
11.2 Fyzická struktura seznamu `(+ (* 2 x) (- (/ (+ x 2) z)) 5)` . . . 280
12.1 Fyzická reprezentace páru `(10 . ahoj)` pomocí metaelementů . . . 298
12.2 Fyzická reprezentace seznamu `(lambda (x) (+ x 1))` pomocí metaelementů . . . 298