KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 8: Rekurze a indukce

Učební materiál k přednášce 23. listopadu 2006
(pracovní verze textu určená pro studenty)

JAN KONEČNÝ, VILÉM VYCHODIL

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

Olomouc 2006

Lekce 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[13]. 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:

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

[13] 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)[14].

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 $P(i + 1)$.*

[14] 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 $\mathcal{L}_{()} = \mathcal{L} \cup \{()\}$. 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 56) 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 $\langle \text{element}, \text{seznam} \rangle$ 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.

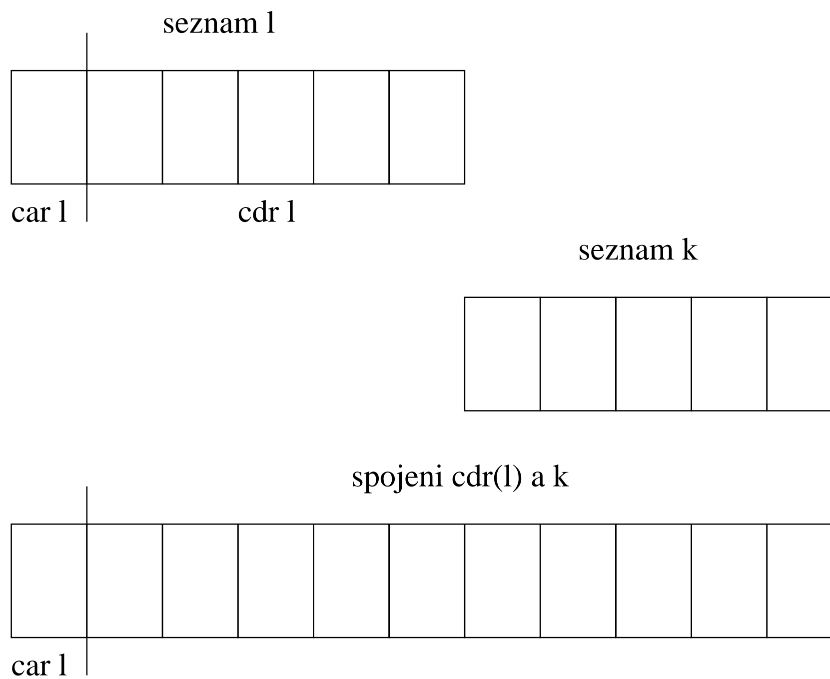
Zobrazení length a append2 můžeme velmi snadno nadefinovat rekurzí přes seznamy. Při tomto typu rekurze jsou funkční hodnoty $f(\ldots, l, \ldots)$, kde l je seznam, vyjádřeny pomocí funkčních hodnot $f(\ldots, k, \ldots)$, kde k je seznam *vyjádřitelný z l pomocí (aspoň jednoho) použití* car *a* cdr.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



„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 length(()) (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 append2.

V případě 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 172 a 173. 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 fázi 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).

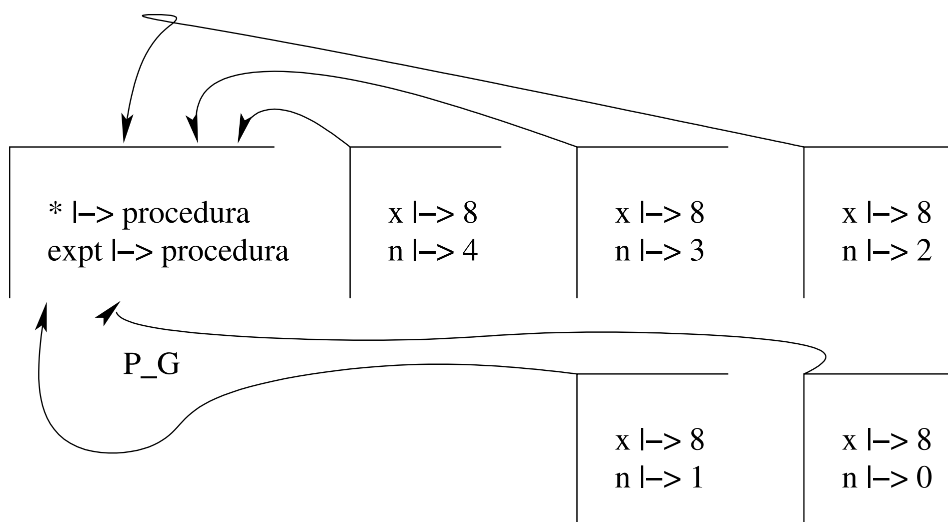
Při aplikaci `expt` s argumenty `8` a `4` došlo de facto k pěti aplikacím této procedury (všechny proběhly ve fází navíjení). Při těchto aplikacích vzniklo tím pádem pět prostředí. Otázkou by možná mohlo být, jestli to není nějaký „nadbytečný luxus“. V žádném případě tomu tak není. Během fáze navíjení 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)` (NACRT OBRAZKU).



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[15]. 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.

[15]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 185 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 201), 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` ⟨*test*⟩ ⟨*důsledek*⟩ ⟨*náhradník*⟩`)` v koncové pozici výrazu Λ,
pak ⟨*důsledek*⟩ i ⟨*náhradník*⟩ (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 ⟨$důsledek_1$⟩, ..., ⟨$důsledek_n$⟩ jsou v koncové pozici výrazy Λ a pokud je přítomna i `else`-větev, pak je i ⟨*náhradník*⟩ v koncové pozici Λ;

(iv) je-li `(and` ⋯ ⟨*výraz*⟩`)` v koncové pozici výrazu Λ, pak je ⟨*výraz*⟩ v koncové pozici výrazu Λ;
totéž platí pro speciální formu `or`;

(v) je-li `(let (⋯)` ⋯ ⟨*výraz*⟩`)` v koncové pozici výrazu Λ, pak je ⟨*výraz*⟩ 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^{5}RS 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` ⋯, 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 204, 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 49, 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[16]. 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

[16] 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 201, 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 fází 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 fází výpočtu jsme narazili na limitní podmínku, nastává fáze odvíjení. V dalším kroku se tedy

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

```
(fib 5)          tohle nakreslit do stromu
  (fib 4)
    (fib 3)
      (fib 2)
        (fib 1)
        (fib 0)
      (fib 1)
    (fib 2)
      (fib 1)
      (fib 0)
  (fib 3)
    (fib 2)
      (fib 1)
      (fib 0)
    (fib 1)
5
```

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

```
(fib 5)                     tohle okomentovat v souladu se stromem
(+ (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 šipky (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 210 a 216, 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 144. Druhou verzi jsme implementovali pomocí `foldr`, viz program 7.1 na straně 172. 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 123, 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 173. 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 rekurzivní 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 146, 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).

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)))))
```

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 125, využívala těžkopádně `build-list` a `list-ref` a byla hodně neefektivní. Další implementace byla ukázána v programu 7.4 na straně 174 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 (reverse2 (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:

```
PRVNI SEZNAM: (1 7 8)       (7 8)        (7 8)      (7 8)      (8)         ()
DRUHY SEZNAM: (2 3 9 10)    (2 3 9 10)   (3 9 10)   (9 10)     (9 10)      (9 10)
VYSLEDEK:     ()            (1)          (1 2)      (1 2 3)    (1 2 3 7)   (1 2 3 7 8)

PRVNI SEZNAM: ()              ()
DRUHY SEZNAM: (10)            ()
VYSLEDEK:     (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 219, 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`

   U každého řešení pak určete složitost.
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. Upravte procedury `fib` a `fak` tak aby:
   - vracely místo výsledku počet aplikací
   - vracely místo výsledku tečkový pár `(výsledek . počet aplikací)`
6. Naprogramuje proceduru `reverse` pomocí pojmenovaného `let`.
7. 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.
2. Zamyslete se, proč první argument speciální formy `if` není v koncové pozici. Co by nefungovalo?
3. Určete složitosti procedur uvedených v sekci 8.6.
4.

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

5. TODO

6.
```
(define reverse
  (lambda (l)
    (let iter ((l l)
               (accum '()))
      (if (null? l)
          accum
          (iter (cdr l) (cons (car l) accum))))))
```

7. (a)
```
(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)))))
```