KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 5: Seznamy

Učební materiál k přednášce 2. 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 5: Seznamy

Obsah lekce: V této lekci se budeme zabývat seznamy, což jsou speciální hierarchické datové struktury konstruované pomocí tečkových párů. Ukážeme několik typických procedur pro manipulaci se seznamy: procedury pro konstrukci seznamů, spojování seznamů, obracení seznamů, mapovací proceduru a další. Vysvětlíme, jaký je vztah mezi seznamy chápanými jako data a seznamy chápanými jako symbolické výrazy. Dále představíme typový systém jazyka Scheme a poukážeme na odlišnost abstraktního interpretu od skutečných interpretů jazyka Scheme, které musí řešit správu paměti.

Klíčová slova: mapování, reverze seznamu, seznam, spojování seznamů, správa paměti, typový systém.

5.1 Definice seznamu a příklady

V minulé lekci jsme uvedli složenou strukturu, kterou jsme nazývali *tečkový pár*. Tečkové páry jsou elementy jazyka, které v sobě agregují dva další elementy. Jelikož páry jsou samy o sobě elementy jazyka, můžeme konstruovat páry jejichž (některé) prvky jsou opět páry. Tímto způsobem lze vytvářet libovolně složité hierarchické struktury postupným „vnořováním párů“ do sebe. V této lekci se zaměříme na speciální případ takových hierarchických dat konstruovaných z tečkových párů, na takzvané *seznamy*.

Než přikročíme k definici seznamu, potřebujeme zavést nový speciální element jazyka. Tímto elementem je *prázdný seznam* (nazývaný též *nil*). Tento element bude, intuitivně řečeno, reprezentovat „seznam neobsahující žádný prvek“ (zpřesnění toho pojmu uvidíme dále). Externí reprezentace prázdného seznamu je `()`. Dále platí, že externí reprezentace prázdného seznamu je *čitelná readerem*. Element „prázdný seznam“ se dle bodu (D) definice vyhodnocování 2.7 uvedené na straně 47 bude *vyhodnocovat sám na sebe*. Máme tedy:

```
()  ⟼  ()
```

Element *prázdný seznam* je pouze jeden, nemá se tedy smysl bavit o „různých prázdných seznamech“. Připomeňme, že z předchozího výkladu již známe několik dalších elementů určených ke speciálním účelům. Jsou to elementy *pravda*, *nepravda* a element *nedefinovaná hodnota*.

Poznámka 5.1. V některých interpretech není pravdou, že se prázdný seznam vyhodnocuje sám na sebe a vyhodnocení výrazu `()` skončí chybou „CHYBA: Pokus o vyhodnocení prázdného seznamu“. Element *prázdný seznam* můžeme v takových interpretech získat pomocí kvotování tím, že potlačíme vyhodnocení výrazu `()`, viz příklady:

```
(quote ())  ⟼  ()
'()         ⟼  ()
```

Ve všech programech v této a v následujících lekcích budeme uvádět prázdný seznam vždy ve kvotovaném tvaru. Dodejme, že v některých dialektech jazyka LISP je prázdný seznam navázaný na symbol `nil`.

Nyní můžeme zavést pojem samotný seznam.

Definice 5.2. *Seznam* je každý element L jazyka Scheme splňující právě jednu z následujících podmínek:

1. L je prázdný seznam (to jest L je element vzniklý vyhodnocením `'()`), nebo
2. L je pár ve tvaru `(`E `.` L'`)`, kde E je libovolný element a L' je seznam. V tomto případě se element E nazývá *hlava seznamu* L a seznam L' se nazývá *tělo seznamu* L (řidčeji též *ocas seznamu* L).

Předpokládejme, že seznam L je ve tvaru `(`E `.` L'`)`. Pod pojmem *prvek seznamu* L rozumíme element E (*první prvek seznamu* L) a dále prvky seznamu L'. Počet všech prvků seznamu se nazývá *délka seznamu*. Prázdný seznam `()` *nemá žádný prvek*, to jest má *délku* 0. ■

Před tím, než uvedeme příklady seznamů, si popíšeme jejich externí reprezentaci. Jelikož je každý seznam buďto párem, nebo prázdným seznamem, souhlasí externí reprezentace seznamů s reprezentacemi těchto elementů. Reprezentaci tečkových párů, kterou jsme si ukázali v předchozí lekci ještě upravíme. Řekli jsme si, že v případě, kdy je druhým prvkem páru opět pár, používáme zkrácený zápis, viz podrobný popis v sekci 4.2. Zkrácení zápisu spočívalo v odstranění tečky náležející reprezentaci vnějšího páru a závorek náležejících vnitřnímu páru. V případě, že druhým prvkem páru je prázdný seznam, zkracujeme stejným způsobem: odstraníme tečku z reprezentace páru a *obě závorky* z reprezentace prázdného seznamu – reprezentace prázdného seznamu tak nebude zobrazena vůbec. Viz příklad následující příklad.

Příklad 5.3. V následujících ukázkách jsou zobrazeny zkrácené externí reprezentace seznamů (uprostřed) a jejich slovní popis (vpravo):

`()`	`= ()`	prázdný seznam
`(a . ())`	`= (a)`	jednoprvkový seznam obsahující symbol `a`
`(a . (b . ()))`	`= (a b)`	dvouprvkový seznam obsahující symboly `a` a `b`
`(1 . (2 . (3 . ())))`	`= (1 2 3)`	tříprvkový seznam
`(1 . ((20 . 30) . ()))`	`= (1 (20 . 30))`	dvouprvkový seznam jehož druhý prvek je pár
`((1 . ()) . ((2 . ()) . ()))`	`= ((1) (2))`	dvouprvkový seznam obsahující jednoprvkové seznamy

Upozorněme na fakt, že i prázdný seznam je element, tedy prázdný seznam sice nemůže obsahovat žádné prvky, ale může být prvek jiných (neprázdných) seznamů. Viz následující příklady:

`(() . ())`	`= (())`	jednoprvkový seznam obsahující prázdný seznam
`(() . (b . ()))`	`= (() b)`	dvouprvkový seznam s prázdným seznamem jako prvním prvkem
`(a . (() . ()))`	`= (a ())`	dvouprvkový seznam s prázdným seznamem jako druhým prvkem
`(() . (() . ()))`	`= (() ())`	dvouprvkový seznam jehož oba prvky jsou prázdné seznamy

Jak je již z předchozích ukázek patrné, zkrácená externí reprezentace seznamů je ve tvary symbolických výrazů tak, jak jsme je zavedli v první lekci v definici 1.6 na straně 19. Korespondence mezi symbolickými výrazy (seznamy) a elementy jazyka (seznamy) se budeme ještě podrobněji zabývat.

Neprázdné seznamy jsou tedy páry, a tak pro ně platí vše, co bylo řečeno v předchozí lekci o párech. Z pohledu seznamů můžeme pohlížet na konstruktor páru `cons` a selektory `car` a `cdr` následovně:

konstruktor `cons` bere dva argumenty, libovolný element E a seznam L (v tomto pořadí), a vrací seznam který vznikne přidáním elementu E na začátek seznamu L.

selektor `cdr` bere jako argument neprázdný seznam a vrací seznam, který vznikne z původního seznamu odebráním prvního prvku. Jinými slovy, `cdr` pro daný seznam vrací jeho *tělo.*

selektor `car` bere jako argument neprázdný seznam a vrací jeho první prvek. Jinými slovy, `car` pro daný seznam vrací jeho *hlavu.*

Příklad 5.4. Příklady konstrukce seznamů použitím procedury `cons`.

`(cons 'a '())`	$\longmapsto$	`(a . ()) = (a)`
`(cons 'a (cons 'b '()))`	$\longmapsto$	`(a . (b . ())) = (a b)`
`(cons 1 (cons 2 (cons 3 '())))`	$\longmapsto$	`(1 . (2 . (3 . ()))) = (1 2 3)`
`(cons 1 (cons (cons 20 30) '()))`	$\longmapsto$	`(1 . ((20 . 30) . ())) = (1 (20 . 30))`
`(cons (cons 10 20) (cons 3 '()))`	$\longmapsto$	`((10 . 20) . (3 . ())) = ((10 . 20) 3)`
`(cons (cons 1 '()) (cons 2 '()))`	$\longmapsto$	`((1 . ()) . (2 . ())) = ((1) 2)`
`(cons (cons 1 '()) (cons (cons 2 '()) '()))`	$\longmapsto$	`((1 . ()) . ((2 . ()) . ())) = ((1) (2))`

V následujících ukázkách vidíme konstrukci seznamů obsahujících prázdné seznamy jako svoje prvky:

`(cons '() '())`	$\longmapsto$	`(() . ()) = (())`
`(cons '() (cons 'b '()))`	$\longmapsto$	`(() . (b . ())) = (() b)`
`(cons 'a (cons '() '()))`	$\longmapsto$	`(a . (() . ())) = (a ())`
`(cons '() (cons '() '()))`	$\longmapsto$	`(() . (() . ())) = (() ())`

Následující hierarchické struktury vzniklé použitím `cons` *nejsou seznamy*:

```
(cons 10 (cons 20 30))          ⟼  (10 . (20 . 30)) = (10 20 . 30)
(cons 'a (cons 'b (cons 'c 'd)))  ⟼  (a . (b . (c . d))) = (a b c . d)
```

Pomocí procedur `cons`, `car` a `cdr` můžeme vytvářet další odvozené konstruktory a selektory pro práci se seznamy. Například můžeme definovat pomocné procedury pro přístup k prvním několika prvkům seznamu.

```
(define first car)
(define second cadr)
(define third caddr)
```

Nebo jednoduché konstruktory seznamů. Třeba procedury tvořící jedno-, dvou- nebo tříprvkové seznamy:

```
(define jednoprvkovy
  (lambda (x)
    (cons x '())))

(define dvouprvkovy
  (lambda (x y)
    (cons x (cons y '()))))

(define triprvkovy
  (lambda (x y z)
    (cons x (cons y (cons z '())))))
```

Analogicky bychom samozřejmě mohli vyrábět procedury na vytváření víceprvkových seznamů.

Příklad 5.5. Upozorněme znovu na fakt, že ne každý seznam je pár. Konkrétně *prázdný seznam není pár.* Vyhodnocení následujícího kódu proto skončí chybou.

```
(car '())  ⟼  „CHYBA: Argument není pár“
(cdr '())  ⟼  „CHYBA: Argument není pár“
```

Poznámka 5.6. V předchozím textu jsme neprázdné seznamy zavedli jako páry jejichž druhým prvkem byl seznam. To byla v podstatě jakési naše „úmluva“ jak reprezentovat lineární datovou strukturu (seznam) pomocí zanořených párů. Samozřejmě, že principiálně nám nic nebrání v tom, abychom udělali jinou úmluvu. Například bychom mohli zaměnit význam prvního a druhého prvku a seznam bychom mohli nadefinovat jako libovolný element L splňující právě jednu z následujících podmínek:

1. L je prázdný seznam, nebo
2. L je pár ve tvaru $(L' \,.\, E)$, kde L' je seznam a E je libovolný element.

Pak bychom například výsledek vyhodnocení výrazu

```
(cons 1 (cons 2 (cons 3 '())))  ⟼  (1 . (2 . (3 . ())))
```

nepovažovali za seznam, ale vyhodnocení následujícího výrazu ano:

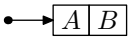
```
(cons (cons (cons '() 1) 2) 3)  ⟼  ((() . 1) . 2) . 3)
```

Neformálně řečeno, jediný rozdíl v obou chápáních seznamu je ve „směru“ v jakém se do sebe tečkové páry zanořují. Ve zbytku textu budeme vždy uvažovat zavedení seznamu tak, jak jsme jej uvedli v definici 5.2 na začátku této lekce.

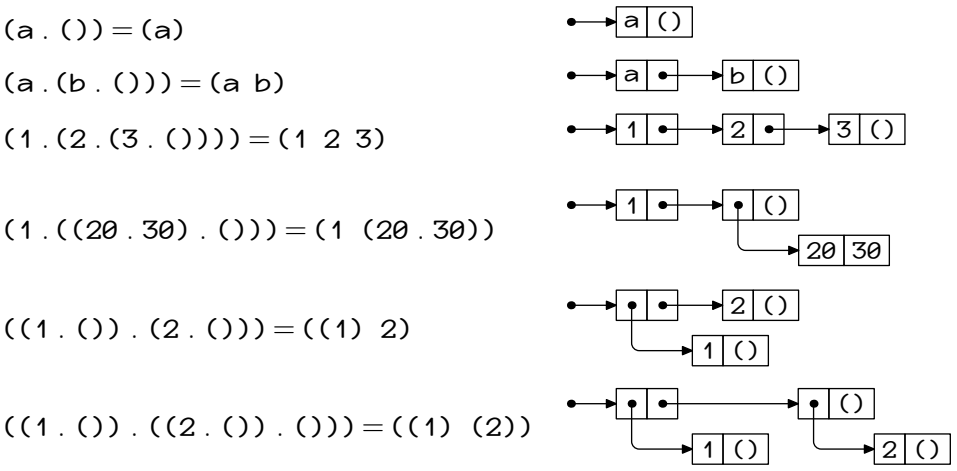
Mimo tečkové notace párů jsme v lekci 4 také představili *boxovou notaci* párů. Tu teď trochu upravíme: Za samotný tečkový pár budeme považovat pouze „tečku“, která obsahuje *ukazatel* na určité místo v paměti, kde je uložen první a druhý prvek páru. Pro ilustraci viz obrázek 5.1.

Naše úprava původní boxové notace má dvě výhody. První výhodou je, že tato notace je přehlednější. Člověk se v nákresu hierarchické struktury v původní notaci snadno ztratí kvůli množství do sebe vnořených boxů.

Obrázek 5.1. Boxová notace tečkového páru používající ukazatel



Obrázek 5.2. Seznamy z příkladu 5.4 v boxové notaci



V nové notaci je každý pár zakreslen zvlášť. Pokud je některý pár obsažen v jiném páru, v diagramu se to promítne tak, že první nebo druhou složkou páru je pouze tečka, ze které vede ukazatel do obsaženého páru. Příklady jsou uvedeny v obrázku 5.2. Další výhodou upravené notace je její větší korespondence s fyzickým uložením párů v pamětích počítačů.

V této lekci jsme vlastně poprvé nahlédli poněkud blíž k fyzické reprezentaci hierarchických dat v počítačích i když, na druhou stranu, pořád se o ní bavíme na dost abstraktní úrovni, která nám zaručuje jisté „pohodlí“. Ve většině interpretů funkcionálních jazyků jsou seznamy reprezentovány dynamickými datovými strukturami, které na sebe vzájemně ukazují pomocí *ukazatelů* – *pointerů* (anglicky *pointers*).

5.2 Program jako data

V definici 1.6 na straně 19 jsme zavedli pojem seznam, jako speciální případ symbolického výrazu. Symbolický výraz je čistě syntaktický pojem, který jsme zavedli proto, abychom mohli formálně popsat programy v jazyku Scheme – programy jsou posloupnosti symbolických výrazů. Jinými slovy, *seznam chápaný jako symbolický výraz* je tedy *část programu.*

V této lekci jsme však doposud pracovali s jiným pojmem „seznam“. Konkrétně jsme pracovali se seznamy, které jsme chápali jako speciální elementy jazyka (speciální hierarchická data). Nabízí se přirozená otázka, jaký je vztah mezi seznamy chápanými jako symbolické výrazy a seznamy chápanými jako elementy jazyka? Z předchozích ukázek je patrné, že se externí reprezentace elementů typu seznam (ve své zkrácené podobě) shoduje s tím, jak jsme zavedli seznamy jako symbolické výrazy. Zde je samozřejmě nutnou podmínkou, aby měl element seznam externí reprezentaci čitelnou readerem, tedy jeho prvky by měly být pouze čísla, symboly a páry (to zahrnuje i další seznamy). Například seznam vzniklý vyhodnocením

```
(cons 'a (cons (lambda () 10) '()))  ⟹  (a „konstantní procedura“)
```

nemá čitelnou externí reprezentaci, protože jeho druhým prvkem je procedura. Externí reprezentace takto vzniklého seznamu tedy není symbolický výraz.

Nyní se zaměříme na opačný problém. V okamžiku, kdy reader přečte vstupní seznam (symbolický výraz), tak jej převede do jeho interní reprezentace. V první lekci, kde jsme se o symbolických výrazech a jejich

externích reprezentacích bavili, jsme záměrně neřekli, jak interní reprezentace seznamů vypadají. Udělali jsme to proto, že až teď je možné tvar interní reprezentace zcela pochopit. *Interní reprezentací symbolického výrazu seznam* je právě hierarchická struktura –seznam– zavedená v definici 5.2 v této lekci. Reader při čtení vstupního symbolického výrazu podle něj konstruuje odpovídající hierarchickou strukturu, která je uložena v paměti interpretu. Dál se již pracuje pouze s touto hierarchickou strukturou, která je interní reprezentací načítaného symbolického výrazu. Zároveň taky platí, že externí reprezentace této hierarchické struktury se shoduje s výchozím symbolickým výrazem. Uvědomte si, že z tohoto nového pohledu je vlastně *program totéž co data.*

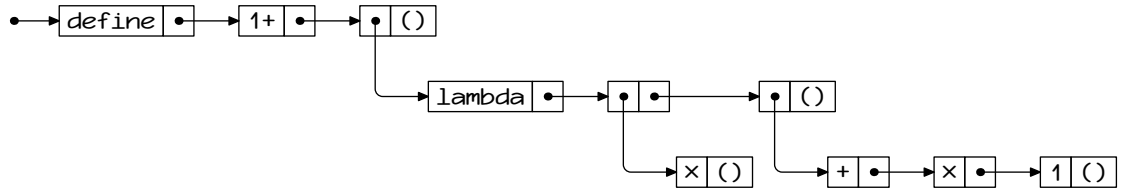
V dialektech jazyka LISP, k nimž patří i jazyk Scheme, se programy chápou jako totéž co data, protože programy se skládají se seznamů. V této lekci navíc dál uvidíme, že zkonstruované seznamy během výpočtu budeme schopni vyhodnocovat. Chápání programů jako dat je doménou prakticky pouze jen programovacích jazyků z rodiny dialektů LISPu. V ostatních programovacích jazycích je datová reprezentace programů buďto skoro nemožná kvůli neprakticky složité syntaxi jazyka nebo je těžkopádná.

Objasněme si tuto problematiku na příkladu. Uvažujme následující program, který je de facto složen z jediného symbolického výrazu, kterým je seznam:

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

Reader předchozí seznam načte a vytvoří podle něj hierarchickou strukturu (element seznam), která je zobrazena v obrázku 5.3.

Obrázek 5.3. Program `(define 1+ (lambda (x) (+ x 1)))` jako data.



Nyní se můžeme vrátit k vyhodnocování párů. V předchozí lekci jsme uvedli, že se zatím vyhodnocováním párů nebudeme zabývat. Nyní, jelikož víme, že seznamy jsou konstruované z párů, se nabízí rozšířit vyhodnocovací proces i pro páry. Při vyhodnocování tečkového páru rozlišíme dvě situace. Konkrétně bude důležité, zda je tento pár seznam, nebo ne. V případě, že ano, vyhodnocujeme tento pár = seznam, jako dříve. V opačném případě skončí vyhodnocování chybou. Vyhodnocení elementu E

Definice 5.7 (vyhodnocení elementu E v prostředí $\mathcal{P}$).
Výsledek vyhodnocení elementu E v prostředí $\mathcal{P}$, značeno $\mathrm{Eval}[E, \mathcal{P}]$, je definován:

(A) Pokud je E *číslo*, pak $\mathrm{Eval}[E, \mathcal{P}] := E$.

(B) Pokud je E *symbol*, mohou nastat tři situace:

(B.1) Pokud $E \mapsto_{\mathcal{P}} F$, pak $\mathrm{Eval}[E, \mathcal{P}] := F$.

(B.2) Pokud E *nemá vazbu v* $\mathcal{P}$ a pokud $\mathcal{P}' \prec \mathcal{P}$, pak $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Eval}[E, \mathcal{P}']$.

(B.e) Pokud E *nemá vazbu v* $\mathcal{P}$ a pokud $\mathcal{P}$ *je globální prostředí*, pak ukončíme vyhodnocování hlášením „CHYBA: Symbol E nemá vazbu.“.

(C) Pokud je E *tečkový pár* tvaru $(E_1 \ . \ E_a)$, pak mohou nastat dvě situace:

(C.α) E_a je seznam ve tvaru $(E_2 \ \ E_3 \ \cdots \ E_n)$. Pak $F_1 := \mathrm{Eval}[E_1, \mathcal{P}]$. Dále rozlišujeme tři situace:

(C.1) Pokud F_1 je *procedura*, pak se v nespecifikovaném pořadí vyhodnotí $E_2, \ldots, E_n$:

$$\begin{aligned} F_2 &:= \mathrm{Eval}[E_2, \mathcal{P}], \\ F_3 &:= \mathrm{Eval}[E_3, \mathcal{P}], \\ &\vdots \quad\ \vdots \\ F_n &:= \mathrm{Eval}[E_n, \mathcal{P}]. \end{aligned}$$

Potom položíme $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Apply}[F_1, F_2, \ldots, F_n]$.

(C.2) Pokud F_1 je *speciální forma*, pak $\mathrm{Eval}[E] := \mathrm{Apply}[F_1, E_2, \ldots, E_n]$.

(C.e) Pokud F_1 není *procedura ani speciální forma*, skončíme vyhodnocování hlášením „CHYBA: Nelze provést aplikaci: E se nevyhodnotil na proceduru ani na speciální formu.“.

(C.β) E_a není seznam. Pak skončíme vyhodnocování chybovým hlášením „CHYBA: Nelze provést aplikaci: E_a není seznam argumentů.“.

(D) Ve všech ostatních případech klademe $\mathrm{Eval}[E, \mathcal{P}] := E$. ■

Poznámka 5.8. Protože seznamy jako S-výrazy jsou interně reprezentovány jako hierarchické struktury konstruované z párů, mohli bychom programy psát přímo jako tečkové páry. Například jednoduchý program `(+ (* 7 3) 5)` bychom též mohli psát třeba těmito způsoby:

```
(+ (* 7 3) . (5))                    ⟹  26
(+ (* 7 3) . (5 . ()))               ⟹  26
(+ (* 7 . (3)) 5)                    ⟹  26
(+ . ((* . (7 . (3 . ()))) . (5 . ()))) ⟹  26
```

Při praktickém programování se však výše uvedené vyjadřování kódu přímo pomocí párů nepoužívá.

V sekci 4.5 jsme si ukázali speciální formu `quote` a kvotování. Stejně tak, jako jsme kvotovali čísla, symboly a páry, můžeme samozřejmě také kvotovat i seznamy:

```
(quote ())                           ⟹  ()
(quote (+ 1 2))                      ⟹  (+ 1 2)
'(+ 1 2)                             ⟹  (+ 1 2)
'(1 2 3 4)                           ⟹  (1 2 3 4)
'(abbe blangis curval durcet)        ⟹  (abbe blangis curval durcet)
'(1 (2 (3)) 4)                       ⟹  (1 (2 (3)) 4)
```

Poznámka 5.9. Kdybychom se pokusili kvotovat dvojnásobně – třeba takto:

```
''blah  ⟹  (quote blah)
```

obdržíme jako výsledek skutečně seznam `(quote blah)`. To by nás ale nemělo překvapit. Když přepíšeme výraz `''blah` bez použití apostrofů (pokud odstraníme syntaktický cukr), dostáváme výraz `(quote (quote blah))`. Tento výraz se vyhodnocuje následujícím způsobem. Na symbol `quote` je navázána speciální forma, které je při aplikaci předán jako argument seznam `(quote blah)` v nevyhodnocené podobě. Speciální forma `quote` tento seznam bez dalšího vyhodnocování vrací. Máme tedy:

```
(quote (quote blah))  ⟹  (quote blah)
```

což je stejný výsledek, který dostaneme při použití apostrofů místo `quote`.

U problematiky procedur jako dat ještě zůstaňme. V první lekci jsme hovořili o readeru jako o části interpretu, která slouží k převodu symbolických výrazů do jejich interních forem. Reader je nám v jazyku k dispozici i jako procedura. Procedura `read` je procedura jednoho argumentu (podle standardu R^{5}RS je to procedura s volitelným druhým parametrem, ale my jím zabývat nebudeme), která čeká na uživatelův vstup a po jeho dokončení vrací interní reprezentaci uživatelem zadaného výrazu. Uvažujme například následující kód:

```
(+ 1 (read))
```

Po jeho vyhodnocení bude uživatel interpretem vyzván k zadání vstupu. Pokud uživatel na vstup zadá například `100`, pak reader převede tento řetězec snaků na element „číslo 100“, které bude přičteno k jedničce a výsledkem je tedy `101`. Kdybychom například vzali výraz

```
(let ((x (read)))
  (cons '+ (cons 1 x))),
```

a kdybychom uvažovali, že uživatel po vyzvání zadá na vstup řetězec znaků „`(a (b 10) c)`“, pak jej reader načte jako seznam a převede jej do interní formy. Tento seznam je dále navázán v lokálním prostředí na `x` a v tomto prostředí bude vyhodnoceno tělo `let`-bloku. Výsledkem vyhodnocení by tedy byl pětiprvkový seznam `(+ 1 a (b 10) c)`. Pomocí `read` tedy můžeme načítat data pomocí uživatelského vstupu. Obzvlášť zajímavé to je v případě, kdy data chápeme jako program. O tom se budeme bavit v následující lekci a k použití `read` se ještě vrátíme.

5.3 Procedury pro manipulaci se seznamy

V této sekci si představíme několik užitečných procedur pro vytváření seznamů a práci se seznamy. Každou proceduru podrobně popíšeme, uvedeme příklady použití a v případě některých procedur rovněž ukážeme, jak bychom je mohli implementovat, kdyby nebyly v interpretu k dispozici.

První z těchto procedur je procedura (konstruktor) `list` vytvářející seznam podle výčtu jeho prvků v daném pořadí. Proceduru lze použít s libovolným (i nulovým) počtem argumentů ve tvaru:

`(list` $\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$`)`

Výsledkem její aplikace je pak seznam těchto argumentů `(`$\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$`)`.

```
(list)                    ⟼  ()
(list 1 2 3)              ⟼  (1 2 3)
(list + 1 2)              ⟼  („procedura sčítání“ 2 3)
(list '+ 1 2)             ⟼  (+ 1 2)
(list (+ 1 2))            ⟼  3
(list (list 1) (list 2))  ⟼  ((1) (2))
```

Poznámka 5.10. Je důležité uvědomit si rozdíl mezi konstrukcí seznamu vznikající vyhodnocením

`(list` $\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$`)`

a zdánlivě ekvivalentním použitím speciální formy `quote`:

`(quote (`$\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$`))`

Speciální forma `quote` zabrání vyhodnocování svého argumentu. Zatímco `list` je procedura a během vyhodnocování prvního z uvedených výrazů jsou před samotnou aplikací `list` vyhodnoceny všechny předané argumenty. Ukažme si rozdíly na příkladech:

(a) V prvním příkladě získáme stejné seznamy, uvažované prvky se totiž vyhodnocují samy na sebe:

```
(list 1 2 #t ())      ⟼  (1 2 #t ())
(quote (1 2 #t ()))   ⟼  (1 2 #t ())
```

(b) V druhém příkladě budeme uvažovat dva prvky – seznam a symbol s navázanou hodnotou:

```
(define x 10)
(list (+ 1 2) x)      ⟼  (3 10)
(quote ((+ 1 2) x))   ⟼  ((+ 1 2) x)
```

V případě procedury `list` došlo k vyhodnocení argumentů, kdežto při použití `quote` nikoliv.

Jiným konstruktorem seznamů je procedura `build-list`. Ta se používá se dvěma argumenty. Prvním z nich je číslo n určující délku konstruovaného seznamu a druhým argumentem je tak zvaná *tvořící procedura* F, což je procedura jednoho argumentu. Tvořící procedura má jediný účel a to pro daný index vracet prvek, který chceme na dané pozici do seznamu vložit. Výsledkem aplikace procedury `build-list` je tedy seznam o zadané délce n, jehož prvky jsou výsledky aplikací tvořící procedury na čísla $0, 1, \ldots, n-1$. Formálně zapsáno, výsledkem aplikace `build-list` je seznam

$$(\mathrm{Apply}[F, 0] \;\; \mathrm{Apply}[F, 1] \cdots \mathrm{Apply}[F, n-1]).$$

Viz následující vysvětlující příklady:

```
(build-list 5 (lambda (x) x))   ⟹  (0 1 2 3 4)
(build-list 5 (lambda (x) 2))   ⟹  (2 2 2 2 2)
(build-list 0 (lambda (x) x))   ⟹  ()
(build-list 5 -)                ⟹  (0 -1 -2 -3 -4)
(build-list 5 list)             ⟹  ((0) (1) (2) (3) (4))
```

Podotkněme, že procedura `build-list` je jednou z procedur, které ukazují praktičnost použití procedur vyšších řádů. Procedura `build-list` je procedura vyššího řádu, protože bere jako jeden ze svých dvou argumentů proceduru. Tato předaná procedura slouží programátorům k vyjádření toho, jaké prvky mají být v seznamu na daných pozicích. Procedura `build-list` je vlastně nejobecnějším konstruktorem seznamů dané délky s důležitou vlastností: hodnoty prvků obsažených v těchto seznamech závisí pouze na jejich pozici.

Poznámka 5.11. Všimněte si, že při popisu `build-list` jsme neuvedli, v jakém pořadí bude tvořící procedura na jednotlivé indexy aplikována – uvedli jsme jen pozici výsledků aplikací ve výsledném seznamu. Z pohledu čistého funkcionálního programování je přísně vzato jedno, v jakém pořadí aplikace proběhly – – libovolné pořadí aplikací vždy povede na stejný seznam. Z pohledu abstrakčních bariér se na `build-list` můžeme opět dívat jako na primitivní proceduru jejíž implementace je nám (zatím skryta). Známe pouze její vstupní argumenty a víme, jak bude vypadat výsledek aplikace.

Další užitečnou procedurou, kterou si představíme, je procedura `length` sloužící ke zjišťování délky seznamu. Procedura akceptuje jeden argument jímž je seznam:

`(length` ⟨*seznam*⟩`)`

a vrací jeho délku, neboli počet jeho prvků.

Příklad 5.12. (a) Délka tříprvkového seznamu je přirozeně tři. `(length '(1 2 3))` ⟹ `3`

(b) Pro prázdný seznam je procedurou `length` vrácena nula. `(length '())` ⟹ `0`

(c) Pokud by seznam l jako svůj prvek obsahoval další seznam l', pak se celý seznam l' počítá jako jeden prvek v seznamu nehledě na počet prvků v l'. Tento rozdíl si nejlépe uvědomíme na následujících příkladech:

```
(length '(a b c d))     ⟹  4
(length '(a (b c) d))   ⟹  3
(length '(a (b c d)))   ⟹  2
(length '((a b c d)))   ⟹  1
```

V sekci 5.1 jsme definovali procedury `first`, `second` a `third` pro přístup k prvnímu, druhému a třetímu prvku seznamu. Často ale potřebujeme přistupovat k nějakému prvku seznamu, jehož pozici v době psaní kódu neznáme, a vypočítáme ji až během výpočtu (to je ostatně spíš pravidlem než výjimkou). V takovém případě můžeme použít proceduru `list-ref`. Používá se se dvěma argumenty ⟨*seznam*⟩ a ⟨*pozice*⟩. Její aplikací je pak vrácen prvek seznamu ⟨*seznam*⟩ na pozici ⟨*pozice*⟩. Pozice jsou přitom počítány od nuly – – první prvek je tedy na pozici nula, druhý na pozici jedna, a tak dále (číslování pozic prvků v seznamu se shoduje s tím, co používá i procedura `build-list`).

```
(list-ref '(a (b c) d) 0)   ⟹  a
(list-ref '(a (b c) d) 1)   ⟹  (b c)
(list-ref '(a (b c) d) 2)   ⟹  d
```

```
(list-ref '(a (b c) d) 3)   ⟼   „CHYBA: prvek na pozici 3 neexistuje“
(list-ref '() 0)            ⟼   „CHYBA: prvek na pozici 0 neexistuje“
```

Další procedurou, kterou budeme někdy potřebovat, je procedura `reverse`, která akceptuje jeden argument, kterým je seznam, a vrací nový seznam obsahující tytéž prvky jako výchozí seznam, ale v opačném pořadí:

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

Poznámka 5.13. Všimněte si, že procedura `reverse` nemodifikuje původní seznam, ale vytváří (konstruuje) podle něj nový. Původní seznam zůstává zachován. V následujícím kódu převracíme seznam navázaný na symbol `sez`.

```
(define sez '(1 2 3))
(reverse sez)          ⟼   (3 2 1)
sez                    ⟼   (1 2 3)
```

Jak je z příkladu patrné, seznam navázaný na `sez` se nezměnil. To je zcela v souladu s funkcionálním duchem našeho jazyka. Nové seznamy jsou konstruovány na základě jiných seznamů.

Manipulaci se seznamy je principiálně možné provádět dvojím způsobem: *konstruktivně* a *destruktivně*. Konstruktivní přístup je vlastní funkcionálnímu programování a spočívá pouze ve vytváření nových seznamů na základě jiných, přitom nedochází k žádné modifikaci již existujících seznamů. Byl-li seznam jednou vytvořen, již jej nelze fyzicky změnit. Destruktivní přístup se používá v procedurálních jazycích a spočívá právě v modifikaci (již existujících) seznamů. Destruktivní modifikace jakýchkoliv struktur je vždy nebezpečná, protože těsně souvisí s *vedlejším efektem*. V dalším díle tohoto textu ukážeme, že i v jazyku Scheme máme k dispozici aparát pro destruktivní práci se seznamy, nyní však budeme se seznamy pracovat pouze konstruktivně.

S tím, co už známe (konkrétně s procedurami `length`, `build-list` a `list-ref`), můžeme naprogramovat vlastní obracení seznamu. To znamená, že proceduru `reverse` bychom v interpretu jazyka Scheme nemuseli mít jako primitivní, ale mohli bychom ji zavést jako uživatelsky definovanou proceduru. Její implementaci nalezneme v programu 5.1. Základní myšlenka programu je následující. Nejprve pomocí procedury `length`

Program 5.1. Obracení seznamu pomocí `build-list`.

```
(define reverse
  (lambda (l)
    (let ((len (length l)))
      (build-list len
                  (lambda (i)
                    (list-ref l (- len i 1)))))))
```

zjistíme délku původního seznamu. Výsledný (převrácený) seznam, který chceme zkonstruovat, bude mít stejný počet prvků. Zjištěná délka n vstupního seznamu bude tedy vstupním argumentem pro konstruktor `build-list`. Tvořící procedura předaná `build-list` pak bude pro argument i (pozice v konstruovaném seznamu) vracet prvek, který je v původním seznamu na pozici $n-i-1$. Tedy na stejné pozici, ale počítáno od konce. Přesvědčte se, že procedura funguje jak má a to včetně mezního případu, kdy je jejím vstupem prázdný seznam[6].

Další představenou procedurou bude `append`. Procedura `append` slouží ke spojování seznamů. Jako argumenty bere dva seznamy

[6]Podotkněme, že implementace `reverse` uvedená v programu 5.1 není příliš efektivní. Stejně tomu bude i u implementací dalších procedur v této sekci. Daleko efektivnější implementace těchto procedur si ukážeme v dalších lekcích. Nyní se spíše než na efektivitu soustředíme na funkcionální styl práce se seznamy.

`(append` $\langle seznam_1 \rangle$ $\langle seznam_2 \rangle$`)`

a vrací tyto dva seznamy spojené v jeden seznam tak, že za posledním prvkem prvního seznamu následuje první prvek druhého seznamu. Viz příklady použití procedury `append`:

```
(append '(a b c) '(1 2))       ⟹  (a b c 1 2)
(append '(a (b c)) '(1 2))     ⟹  (a (b c) 1 2)
(append '(a (b) c) '((1 2)))   ⟹  (a (b) c (1 2))
(append '() '(1 2))            ⟹  (1 2)
(append '(a b c) '())          ⟹  (a b c)
(append '() '())               ⟹  ()
```

Všimněte si, že například na druhém řádku předchozí ukázky jsme spojovali seznamy `(a (b c))` a `(1 2)`, to jest první z těchto seznamů měl jako druhý prvek seznam `(b c)`. Ve výsledném spojení je opět druhým prvkem seznam `(b c)`, nedochází tedy ke vkládání jednotlivých prvků ze seznamu `(b c)` do výsledného seznamu. Z předchozí ukázky je rovněž patrné, že operace „skládání seznamů“, která je prováděna při aplikaci `append` se chová neutrálně vůči prázdnému seznamu. Spojení seznamu l s prázdným seznamem (nebo obráceně) dá opět seznam l. Jako důsledek získáváme, že spojením prázdného seznamu se sebou samým je prázdný seznam, viz poslední řádek ukázky.

Podotkněme, že provedení spojení dvou seznamů si lze představit jako sérii postupných aplikací konstruktoru `cons`. Výsledek spojení dvou seznamů je právě to, co bychom získali, kdybychom postupně všechny prvky $\langle prvek_1 \rangle, \langle prvek_2 \rangle, \ldots, \langle prvek_n \rangle$ prvního seznamu $\langle seznam_1 \rangle$ připojili na začátek seznamu druhého seznamu takto:

`(cons` $\langle prvek_1 \rangle$ `(cons` $\langle prvek_2 \rangle \cdots$ `(cons` $\langle prvek_n \rangle$ $\langle seznam_2 \rangle$`)` $\cdots$ `)`.

Například v případě seznamů `(a b c)` a `(1 2)` máme:

```
(append '(a b c) '(1 2))                ⟹  (a b c 1 2)
(cons 'a (cons 'b (cons 'c '(1 2))))    ⟹  (a b c 1 2)
```

Také proceduru pro spojování dvou seznamů už budeme schopni naprogramovat. Její naprogramování lze provést mnoha způsoby. Nejefektivnější z nich je založen právě předchozí násobné aplikaci `cons`. My si však zatím ukážeme méně efektivní verzi spojení dvou seznamů (nazvanou `append2`) uvedenou v programu 5.2. Pomocí procedury `build-list` vytvoříme seznam, jehož délka bude součtem délek původních seznamů,

Program 5.2. Spojení dvou seznamů pomocí `build-list`.

```
(define append2
  (lambda (l1 l2)
    (let ((len1 (length l1))
          (len2 (length l2)))
      (build-list (+ len1 len2)
                  (lambda (i)
                    (if (< i len1)
                        (list-ref l1 i)
                        (list-ref l2 (- i len1)))))))))
```

protože nám jde o to zkonstruovat jejich spojení. Tvořící procedura pak pomocí procedury `list-ref` vybírá prvky z původních seznamů. V těle tvořící procedury je potřeba rozlišit dva případy (všimněte si podmínky formulované ve speciální formě `if`). Konkrétně musíme rozlišit případ, kdy ještě vybíráme prvky z prvního seznamu a kdy je již vybíráme z druhého (v případě druhého seznamu je navíc potřeba přepočítat index, zdůvodněte si podrobně proč).

Spojování seznamů je asociativní operace. To znamená, že třeba vyhodnocení výrazu

```
(append (append '(1 2) '(3 4)) '(5 6))   ⟹  (1 2 3 4 5 6)
```

má stejný výsledek jako vyhodnocení výrazu

```
(append '(1 2) (append '(3 4) '(5 6)))  ⟹  (1 2 3 4 5 6).
```

Jelikož jsme v předchozí části viděli, že prázdný seznam se chová vůči spojování seznamů neutrálně, můžeme říct, že *operace spojování seznamů* je *monoidální operace* (viz sekci 2.5) na množině všech seznamů. Máme tedy další důležitý příklad monoidální operace. Podotkněme, že spojování pochopitelně není komutativní, protože spojením dvou seznamů v opačném pořadí nemusíme získat stejný výsledek.

Procedura `append` je ve skutečnosti obecnější, než jak jsme na v úvodu předeslali. Procedura `append` bere libovolný počet argumentů jimiž jsou seznamy:

`(append` ⟨*seznam*$_1$⟩ ⟨*seznam*$_2$⟩ $\cdots$ ⟨*seznam*$_n$⟩`)`,

kde $n \geq 0$. Následující ukázky ukazují použití `append` s různým počtem argumentů.

```
(append '(a b c) '(1 2) '(#t #f))  ⟹  (a b c 1 2 #t #f)
(append '(a b c) '() '(#t #f))     ⟹  (a b c #t #f)
(append '() '() '())               ⟹  ()
(append '(a b c))                  ⟹  (a b c)
(append)                           ⟹  ()
(let ((s '(a b c)))
  (append s s s s))                ⟹  (a b c a b c a b c a b c)
```

Při aplikaci bez argumentu vrací `append` prázdný seznam. Pokud vás to překvapuje, připomeňme, se například i procedury `+` a `*` se chovaly tak, že pro prázdný seznam argumentů vracely neutrální prvek.

Nyní obrátíme naši pozornost na další proceduru vyššího řádu pracující se seznamy. Často je potřeba z daného seznamu vytvořit nový seznam stejné délky, jehož prvky jsou nějakým způsobem „změněny“. Takové činnosti se říká *mapování procedury přes seznam*. K provedení této operace a ke konstrukci nového seznamu tímto způsobem nám ve Scheme slouží procedura `map`. Procedura `map` bere dva argumenty. Prvním z nich je procedura ⟨*proc*⟩ jednoho argumentu. Druhým argumentem je seznam. Proceduru ⟨*proc*⟩ budeme ve zbytku kapitoly nazývat *mapovaná procedura*. Aplikace `map` tedy probíhá ve tvaru

`(map` ⟨*proc*⟩ ⟨*seznam*⟩`)`

Výsledkem této aplikace je seznam výsledků aplikace procedury ⟨*proc*⟩ na každý prvek seznamu ⟨*seznam*⟩. Označme prvky seznamu ⟨*seznam*⟩ postupně ⟨*prvek*$_1$⟩, ⟨*prvek*$_2$⟩, ..., ⟨*prvek*$_n$⟩. Aplikací procedury `map` tedy dostáváme seznam

`(`Apply(⟨*proc*⟩, ⟨*prvek*$_1$⟩) Apply(⟨*proc*⟩, ⟨*prvek*$_2$⟩) $\cdots$ Apply(⟨*proc*⟩, ⟨*prvek*$_n$⟩)`)`.

Viz následující příklady použití procedury `map`:

```
(map (lambda (x) (+ x 1)) '(1 2 3 4))   ⟹  (2 3 4 5)
(map - '(1 2 3 4))                      ⟹  (-1 -2 -3 -4)
(map list '(1 2 3 4))                   ⟹  ((1) (2) (3) (4))
(map (lambda (x) x) '(a (b c) d))       ⟹  (a (b c) d)
(map (lambda (x) #f) '(1 2 3))          ⟹  (#f #f #f)
(map (lambda (x) #f) '())               ⟹  ()
(map (lambda (x) (<= x 3)) '(1 2 3 4))  ⟹  (#t #t #t #f)
(map even? '(1 2 3 4))                  ⟹  (#f #t #f #t)
```

Mapování procedury přes jeden seznam jsme schopni implementovat. Kód procedury je uveden v programu 5.3. V tomto programu jsme pomocí procedury `build-list` vytvořili nový seznam o stejné délce jako je seznam původní. Tvořící procedura nám pro daný index vybere prvek z původního seznamu na odpovídající pozici (pomocí `list-ref`) a pak na něj aplikuje mapovanou proceduru.

Proceduru `map` lze použít rovněž s více než se dvěma argumenty. Seznam nemusí být jen jeden, ale může jich být jakýkoli kladný počet k:

`(map` ⟨*proc*⟩ ⟨*seznam*$_1$⟩ ⟨*seznam*$_2$⟩ $\cdots$ ⟨*seznam*$_k$⟩`)`

Všechny seznamy ⟨*seznam*$_i$⟩ $(1 \leq i \leq k)$ musí mít stejnou délku. Mapovací procedura ⟨*proc*⟩ musí být procedurou k argumentů. Jinými slovy: musí být procedurou tolika argumentů, kolik je seznamů. Výsledkem je pak seznam výsledků aplikací procedury ⟨*proc*⟩ na ty prvky seznamů, které se v předaných seznamech nacházejí na stejných pozicích. Pro vysvětlení viz příklady aplikací:

Program 5.3. Mapovací procedura pracující s jedním seznamem pomocí `build-list`.

```
(define map1
  (lambda (f l)
    (build-list (length l)
                (lambda (i)
                  (f (list-ref l i))))))
```

```
(map (lambda (x y) (+ x y)) '(1 2 3) '(10 20 30))  ⟼  (11 22 33)
(map + '(1 2 3) '(10 20 30))                       ⟼  (11 22 33)
(map cons '(a b) '(x y))                           ⟼  ((a . x) (b . y))
(map cons '(a b #t) '(x y #f))                     ⟼  ((a . x) (b . y) (#t . #f))
(map <= '(1 2 3 4 5) '(5 4 3 2 1))                 ⟼  (#t #t #t #f #f)
(map (lambda (x y z)
       (list x (+ y z))) '(a b) '(1 2) '(10 20))   ⟼  ((a 11) (b 22))
```

Proceduru `map` pro více než jeden seznam zatím vytvořit neumíme. Ale vrátíme se k této problematice v příštích lekcích. Opět (obdobně jako v případě procedury `build-list`) jsme zatajili v jakém pořadí se bude mapovaná procedura na jednotlivé prvky seznamu aplikovat. Z čistě funkcionálního pohledu na programování, o který nám v této části textu jde, to asi není důležité.

5.4 Datové typy v jazyku Scheme

V této sekci si nejprve shrneme všechny typy elementů, které jsme si do této chvíle představili. Typ elementy budeme stručně nazývat *datový typ*, protože na elementy se můžeme dívat jako na *data* (*hodnoty*), nad kterými provádíme různé výpočty. Ukážeme predikáty, kterými můžeme identifikovat typ daného elementu jazyka. Dále si ukážeme predikát `equal?`, kterým je možné „porovnávat“ libovolné dva elementy.

Z předchozích lekcí již známe tyto základní typy elementů jazyka: *čísla*, *pravdivostní hodnoty*, *symboly*, *tečkové páry*, *prázdný seznam*, *procedury* (*primitivní procedury* a *uživatelsky definované procedury*). K těmto základním typů elementů bychom ještě mohli přidat element *nedefinovaná hodnota* i když standard R^{5}RS jazyka Scheme nedefinovanou hodnotu chápe jinak, viz [R5RS]. V této sekci jsme rovněž představili odvozený typ elementu *seznam*[7].

Při vytváření programů v jazyku Scheme je leckdy potřeba řídit výpočet podle typu argumentu nebo více argumentů, které jsou předané procedurám. Při psaní procedur je tedy vhodné mít k dispozici predikáty, kterými můžeme pro daný element rozhodnout o jeho typu. V jazyku Scheme máme k dispozici následující predikáty, kterými můžeme kvalifikovat typy elementů:

`boolean?`	*pravdivostní hodnota*,
`list?`	*seznam*,
`null?`	*prázdný seznam*,
`number?`	*číslo*,
`pair?`	*tečkový pár*,
`procedure?`	*procedura* (*primitivní* nebo *uživatelská*),
`symbol?`	*symbol*.

V levé části výpisu máme uvedena jména symbolů, na které jsou predikáty navázány a v pravém sloupci je uveden popis jejich významu.

Poznámka 5.14. Pokud nepočítáme predikát `list?`, jsou uvedené predikáty vzájemně disjunktní, to jest: při jejich aplikaci na jakýkoli element nám *nejvýše jeden z nich vrátí pravdu*. Pokud predikát `list?` vrací pro

[7] Seznam je skutečně „odvozený typ elementu“, protože přísně vzato je každý seznam buďto prázdný seznam (speciální element) nebo je to tečkový pár ve speciálním tvaru.

nějaký element pravdu, pak vrací pravdu i právě jeden z predikátů `pair?` nebo `null?` (to plyne ze zavedení seznamu, viz definici 5.2 na straně 114). Viz následující příklady:

```
(list? '(10 20 30))    ⟹  #t
(list? '(10 20 . 30))  ⟹  #f
(pair? '(10 20 30))    ⟹  #t
(pair? '(10 20 . 30))  ⟹  #t
```

Nyní se můžeme opět vrátit k rozdílu mezi symbolem a hodnotou, která je na něm navázána, o kterém jsme již několikrát mluvili. Opět je nutné dobře si uvědomovat tento rozdíl. Uvažujme následující příklad:

```
(define x 10)
(symbol? x)          ⟹  #f
(symbol? 'x)  ⟹  #t
```

V této ukázce máme v globálním prostředí navázáno číslo `10` na symbol `x`. Při prvním použití predikátu `symbol?` je vrácena pravdivostní hodnota `#f`, protože symbol `x` se vyhodnotí na svou vazbu (číslo `10`), což není symbol. V druhém případě je vráceno `#t`, protože `'x` se vyhodnotí na symbol `x`.

Predikáty uvedené v této sekci můžeme použít ke spouště účelů. Můžeme pomocí nich například vytvořit „bezpečnou verzi“ selektorů `car` a `cdr`, které se odlišují od původních selektorů tím, že při aplikaci na prázdný seznam nezpůsobí chybu. Namísto toho vrací prázdný seznam `()`[8]:

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

(define safe-cdr
  (lambda (l)
    (if (null? l)
        '()
        (cdr l))))
```

V obou případech jsme nejdříve použitím predikátu `null?` zjistili, zda není na formální argument `l` navázán prázdný seznam. Pokud ano, je výsledkem vyhodnocení prázdný seznam `()`, jinak na hodnotu navázanou na `l` aplikujeme původní selektor.

Další ukázkou použití predikátů rozhodujících typ je jednoduchý predikát `singleton-list?`. Ten zjišťuje, jestli je jeho argument jednoprvkový seznam:

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

Program je vlastně přímým přepisem tvrzení, že jednoprvkový seznam je pár a jeho druhý prvek je prázdný seznam `()`.

Ve Scheme existuje predikát `equal?`, kterým je možné porovnávat libovolné elementy:

Definice 5.15 (predikát `equal?`). Procedura `equal?` se používá se dvěma argumenty

`(equal?` ⟨*element*$_1$⟩ ⟨*element*$_2$⟩`)`

a vrací `#t`, právě když pro ⟨*element*$_1$⟩ a ⟨*element*$_2$⟩ platí:

- oba ⟨*element*$_1$⟩ a ⟨*element*$_2$⟩ jsou buď `#t` nebo `#f`;
- oba ⟨*element*$_1$⟩ a ⟨*element*$_2$⟩ jsou stejné symboly;

[8] Takto fungují selektory `car` a `cdr` v jazyce Common LISP

- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou čísla, jsou obě buď v přesné nebo v nepřesné reprezentaci a obě čísla jsou si numericky rovny, to jest aplikací `=` na $\langle element_1 \rangle$ a $\langle element_2 \rangle$ bychom získali `#t`;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou prázdné seznamy;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou nedefinované hodnoty;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou stejné primitivní procedury nebo stejné uživatelsky definované procedury, nebo stejné speciální formy;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou páry ve tvarech $\langle element_1 \rangle = (E_1 \ . \ F_1)$ a $\langle element_2 \rangle = (E_2 \ . \ F_2)$ a platí:
  - výsledek aplikace `equal?` na E_1 a E_2 je `#t`
  - výsledek aplikace `equal?` na F_1 a F_2 je `#t`;

a vrací `#f` ve všech ostatních případech. ■

Při použití predikátu `equal?` na čísla je potřeba, aby byla obě buď v přesné nebo v nepřesné reprezentaci. To je rozdíl vzhledem k predikátu `=`, u kterého rozhoduje jen numerická rovnost. Viz následující příklad:

```
(equal? 2 2.0)   ⟹ #f
(= 2 2.0)        ⟹ #t
(equal? 2 2)     ⟹ #t
(equal? 2.0 2.0) ⟹ #t
```

Ačkoliv „rovnost procedur“ budeme testovat jen zřídka, zdali vůbec, objasněme co znamená, že dvě uživatelsky definované procedury jsou „stejné“. Každý uživatelsky definovaná procedura je z našeho pohledu „stejná“ jen sama se sebou. I kdybychom vzali dvě procedury vzniklé vyhodnocením téhož λ-výrazu, z pohledu predikátu `equal?` se bude jednat o různé elementy, viz následující ukázku. Máme:

```
(equal? (lambda (x) (* x x)) (lambda (x) (* x x)))  ⟹  #f
```

Ale na druhou stranu:

```
(define na2 (lambda (x) (* x x)))
(equal? na2 na2)                    ⟹  #t
```

Rovněž upozorněme na fakt, že `equal?` se hodí k porovnávání jakýchkoliv elementů, tedy i hierarchických struktur. Z popisu `equal?` vidíme, že tímto predikátem můžeme snadno testovat rovnost dvou tečkových párů, což nemůžeme provést pomocí predikátu `=` (rovnost čísel):

```
(equal? '(a b) '(a b))   ⟹ #t
(= '(a b) '(a b))        ⟹ „CHYBA: Argumenty nejsou čísla“
```

Poznámka 5.16. Pozorní čtenáři si jistě všimli, že jsme neuvedli žádné predikáty, kterými by bylo možné identifikovat speciální formy. Neuvedli jsme je proto, že ve specifikaci jazyka R^{5}RS, viz dokument [R5RS], takové predikáty nejsou. Standard jazyka Scheme R^{5}RS totiž nechápe speciální formy jako elementy jazyka. Stejně tak jsme mohli vytvořit dva predikáty, které testují zda-li je daná procedura primitivní nebo uživatelsky definovaná. Představený systém predikátů tedy mohl být mnohem bohatší. Pro naše účely si ale plně vystačíme s předchozími procedurami, některé nové predikáty určující typy elementů představíme v poslední lekci tohoto dílu textu.

Na závěr této sekce o datových typech v jazyku Scheme podotkněme, že jiné programovací jazyky mají k datovým typům odlišný přístup. Teorie datových typů je jedna z řady disciplín, které jsou v informatice dodnes hluboce studovány a výzkum v této oblasti postupuje pořád dál. Každý jazyk lze z hlediska použitého modelu typů rozdělit do několika kategorií. První rozdělení je založeno na tom, v jakém okamžiku je možné provést kontrolu typu elementů:

Staticky typované jazyky. Jedná se o programovací jazyky, pro které je z principu možné udělat kontrolu typů již před interpretací nebo během překladu programu, pouze na základě znalosti jeho syntaktické struktury.

Dynamicky typované jazyky. U dynamicky typovaných jazyků platí, že pouhá struktura kódu (vždy) nestačí ke kontrole typů a typy elementů musí být kontrolovány až za běhu programu. U dynamicky typových jazyků je rovněž možné, že jedno jméno (symbol nebo proměnná) může během života programu nést hodnoty různých typů.

Z toho co víme o jazyku Scheme je jasné, že se jedná o *dynamicky typovaný jazyk*, protože když uvážíme třeba výraz

```
(let ((x (read)))
  (number? x)),
```

tak je zcela zřejmé, že o hodnotě jeho vyhodnocení, která závisí na kontrole typu elementu navázaného na `x` budeme moci rozhodnout až za běhu programu, protože na `x` bude navázán vstupní výraz zadaný uživatelem. Naproti tomu třeba jazyk C je staticky typový, typ všech proměnných je navíc potřeba explicitně *deklarovat*, to jest „dát překladači na vědomí“ před jejich prvním použitím v programu.

Další úhel pohledu na vlastnosti typového systému je jeho „síla“. Některé programovací jazyky umožňují používat procedury (operace) jen s argumenty přesně daného typu a pokud je vstupní argument jiného typu, provedení operace selže. Tak se chová i jazyk Scheme. Naopak jiné programovací jazyky definují sadu pravidel pro „převod“ mezi datovými typy (příkladem takového jazyka je třeba jazyk PERL). Proto rozlišujeme dva typy „síly“ typového systému:

Silně typované jazyky. Silně typovaný jazyk má pro každou operaci přesně vymezený datový typ argumentů. Pokud je operace použita s jiným typem argumentů než je přípustné, dochází k chybě.

Slabě typované jazyky. Slabě typované jazyky mají definovánu sadu konverzních pravidel, která umožňují, pokud je to nutné, převádět mezi sebou data různých typů. Pokud je provedena operace s argumenty, které jí typově neodpovídají, je provedena konverze typů tak, aby byla operace proveditelná.

Poslední vlastností je typová bezpečnost:

Bezpečně typované jazyky. Tyto jazyky se chovají korektně z hlediska provádění operací mezi různými typy elementů. To znamená, že pokud je operace pro daný typ elementů proveditelná, její provedení nemůže způsobit havárii programu.

Nebezpečně typované jazyky. Nebezpečně typované jazyky jsou jazyky, které nejsou bezpečně typované. Výsledek operace mezi různými typy tedy může vést k chybě při běhu programu. Takovým jazykem je třeba C, kde chyba tohoto typu může být způsobena například přetečením paměti nebo dereferencí ukazatele z ukazujícího na místo paměti nepatřící programu.

Jazyk Scheme je z tohoto pohledu bezpečně typovaný jazyk.

5.5 Implementace párů uchovávajících délku seznamu

V této sekci si ukážeme implementaci „párů“, které si pamatují délku seznamu. To povede na zvýšení efektivity při práci se seznamy – délku seznamu bude možné zjistit k konstantním čase, protože bude přímo kódovaná v seznamu. Přestože uvádíme, že ukážeme novou implementaci pár, bude se vlastně jednat o trojice, jejichž prvky budou $\langle$*hlava*$\rangle$, $\langle$*tělo*$\rangle$ a $\langle$*délka*$\rangle$. A právě v posledním jmenovaném prvku budeme uchovávat délku seznamu.

Seznam délky n budeme pomocí těchto párů definovat následovně:

- pro $n = 0$ je to prázdný seznam `()`;
- pro $n \in N$ je to pár jehož prvek *hlava* je libovolný element, prvek *tělo* je seznam délky $n - 1$ a prvek *délka* je číslo n.

Kdykoli budeme v této sekci mluvit o seznamech, budeme tím myslet seznam ve smyslu výše uvedené definice. V dalších sekcích ale vrátíme k původní reprezentaci párů, která je součástí interpretu jazyka Scheme.

Náš nový pár bude opět reprezentován procedurou, stejně jako v programu 4.3 na straně 4.3. Svazujeme jí ale tři hodnoty: dvě z nich jsou vstupní argumenty `a` a `b`. Třetí hodnota je pak délka seznamu `b` zvýšená o 1. Konstruktor nového páru bude vypadat následovně:

```
(define cons
  (lambda (a b)
    (lambda (dispatch-f)
      (dispatch-f a b (+ 1 (fast-length b))))))
```

Všimněte si, že v konstruktoru jsme použili proceduru `fast-length` což je nový selektor vracející pro daný seznam jeho délku. Při připojení nového prvku k již existujícímu seznamu zjistíme pomocí tohoto selektoru jeho délku a do aktuálního páru zakódujeme délku zvětšenou o 1 (přidali jsme jeden prvek). Implementaci selektoru `fast-length` uvedeme posléze. Selektory `car` a `cdr` můžeme naprogramovat klasicky pomocí projekcí:

```
(define 1-ze-3 (lambda (x y z) x))
(define 2-ze-3 (lambda (x y z) y))

(define car
  (lambda (par)
     (par 1-ze-3)))

(define cdr
  (lambda (par)
     (par 2-ze-3)))
```

Zjišťování délky seznamu se bude trochu lišit. Nejdříve ověříme, zda se nejedná o prázdný seznam `()`. V tom případě vracíme nulu – tedy délku prázdného seznamu. V opačném případě vracíme prvek *délka*, což je informace o délce, která je součástí každého páru. Viz následující kód:

```
(define 3-ze-3 (lambda (x y z) z))

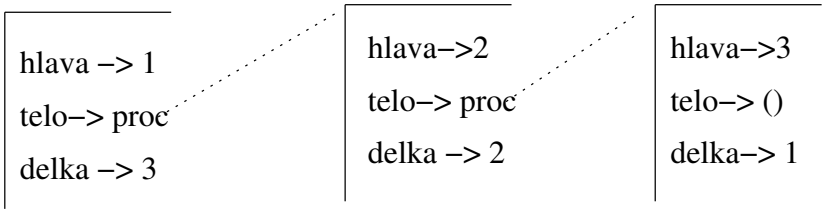
(define fast-length
  (lambda (par)
     (if (null? par)
         0
         (par 3-ze-3))))
```

Příklad 5.17. Podívejme se nyní na použití nové implementace párů:

```
(define seznam (cons 1 (cons 2 (cons 3 '()))))
```

Vytvořili jsme tedy seznam nikoli z tečkových párů, ale z procedur vyšších řádů, které v sobě zapouzdřují tři hodnoty – hlavu, tělo a délku seznamu. Tělo seznamu je přitom zase seznamem. Obrázek 5.4 ukazuje prostředí, která reprezentují seznam. Přerušovaná čára ukazuje prostředí vzniku procedur reprezentujících seznam.

Obrázek 5.4. Procedury a prostředí u párů uchovávajících délku seznamu



Nyní uvedeme použití selektorů: Výraz `(car seznam)` se vyhodnotí na číslo `1`. Na seznam je navázána procedura o jednom argumentu. Tuto proceduru aplikuje selektor `car` ve svém těle na projekci prvního prvku navázanou na symbol `1-ze-3`. Tuto projekci aplikuje procedura `seznam` na elementy navázané na symboly `hlava`, `telo` a `delka` v prostředí jejího vzniku $\mathcal{P}_1$. Výsledkem je tedy číslo `1`.

Výraz `(car (cdr seznam))` se vyhodnotí takto: vyhodnocením podvýrazu `(cdr seznam)` (podobným způsobem jako v předchozím příkladě) dostáváme hodnotu navázanou na symbol `telo` v prostředí $\mathcal{P}_1$, což je opět (neprázdný) seznam – procedura, která bere jako argument projekci, a vrací výsledek její aplikace na hodnoty navázané na symboly `hlava`, `telo` a `delka` v prostředí $\mathcal{P}_2$. A tedy dostáváme číslo 2. Podobně vyhodnocením výrazu `(fast-length (cdr seznam))` bude číslo 2 – hodnota navázaná na symbol `delka` v prostředí $\mathcal{P}_2$.

Poznámka 5.18. Všimněte si, že naše nová procedura `cons` nemůže být použita ke konstrukci obecných párů, to jest párů, jejichž druhý prvek není ani další pár ani prázdný seznam. Jako argument `telo` musí předán seznam; v těle konstruktoru `cons` totiž zjišťujeme délku seznamu navázaného na `telo`.

5.6 Správa paměti během činnosti interpretu

Při našem exkursu funkcionálním paradigmatem jsem se až doposud výhradně bavili o činnosti abstraktního interpretu jazyka Scheme. Skutečné interprety, tedy ty se kterými reálně pracujeme, jsou obvykle výrazně složitější. Vyhodnocovací proces je v nich implementován zhruba tak, jako v našem abstraktním interpretu, jejich složitost však spočívá v tom, že interprety musí řešit řadu otázek souvisejících se správou a organizací paměti.

V této sekci malinko nahlédneme pod pokličku skutečného interpretu jazyka Scheme a zaměříme se na jeden specifický problém správy paměti související s tečkovými páry. Každý interpret jazyka Scheme musí nějak reprezentovat elementy jazyka: čísla, seznamy, procedury, páry a tak dále. Pro reprezentaci párů se obvykle používá struktura obsahující ukazatel na dvojici dalších elementů – elementů „obsažených v páru“. To je v souladu s upravenou boxovou notací uvedenou v této lekci. Během činnosti programu vznikají nové páry konstrukcí a časem může dojít k situaci, že již některý pár, která je uložen v paměti, není potřeba. To znamená, že neexistuje metoda, jak v programu tento pár získat vyhodnocením nějakého výrazu. Demonstrujme si tuto situaci příkladem:

```
(define s '(a b c))
s                    ⟼  (a b c)
(define s #f)
s                    ⟼  #f
```

Nejprve jsme na `s` navázali seznam jehož interní reprezentaci pomocí párů musí udržovat interpret Scheme v paměti. Na třetím řádku jsme na symbol `s` navázali hodnotu `#`. Tím pádem jsem ztratili jakoukoliv vazbu na zkonstruovaný seznam `(a b c)`. Logika věci říká, že interpret jazyka Scheme by to měl nějak poznat a odstranit interní reprezentaci seznamu z paměti[9]. Někoho by nyní mohlo napadnout, že testovat „viditelnost“ seznamů je jednoduché: nabízí se pouze „smazat každý seznam“, který byl dříve navázaný na symbol, který jsme právě redefinovali (to odpovídá předchozí situaci). Není to, bohu žel, tak jednoduché, viz následující příklad:

```
(define s '(a b c))
s                    ⟼  (a b c)
(define r s)
(define s #f)
s                    ⟼  #f
r                    ⟼  (a b c)
```

Zde jsme provedli skoro totéž, co v předchozím případě, provedli jsme ale navíc jednu definici. Před redefinicí vazby `s` jsme na `r` definovali aktuální vazbu `s`. To bylo v okamžiku, kdy na `s` byl ještě navázán seznam `(a b c)`. I když po provedení redefinice `s` již na `s` tento seznam navázaný není, je stále navázaný na `r`. Předchozí navržená metoda „vymazávání seznamů po redefinici“ by tedy nebyla ani účinná a ani korektní (vedla by k havárii interpretu a tím i interpretovaného programu).

[9] Interpret si nemůže dovolit pouze plnit paměť novými páry, uvědomte si, že třeba při každém použití `cons` je vytvořen jeden nový pár, při konstrukci n prvkového seznamu je to n nových párů. Kdyby interpret nevěnoval správě paměti pozornost, činnost většiny programů by byla záhy násilně ukončena operačním systémem z důvodu vypotřebování paměti.

Užitečné je taky uvědomit si, že seznamy obecně nelze jen tak likvidovat celé. Někdy může být ještě část seznamu „dostupná“. Opět si uveďme demonstrační příklad:

```
(define r '(x y))
(define s (cons 100 r))
r                       ⟼  (x y)
s                       ⟼  (100 x y)
(define s #f)
r                       ⟼  (x y)
s                       ⟼  #f
```

Zde jsme zkonstruovali seznam a navázali jej na `r`. Poté jsme na `s` navázali seznam vzniklý z předchozího přidáním prvku na první pozici. Pokud redefinujeme vazbu `s` tak, jak je v příkladu ukázáno, dostaneme se do situace, kdy již není dostupný seznam `(100 x y)`, ale je pořád ještě dostupný dvouprvkový seznam `(x y)` tvořící jeho tělo. To znamená, že není možné celou reprezentaci seznamu `(100 x y)` odstranit z paměti, protože tělo tohoto seznamu je stále dostupné pomocí symbolu `r`. Zároveň je ale jasné, že pár jehož prvním prvkem je číslo 100 a druhým prvkem je (ukazatel na) dvouprvkový seznam `(x y)`, již smazat můžeme.

Jak tedy interprety provádějí správu paměti? Obecně je to tak, že každý interpret má v sobě zabudovaný podprogram, tak zvaný *garbage collector,* který se během vyhodnocování jednou za čas spustí (například, když rychle klesne volná paměť), projde všechny elementy reprezentované v paměti a smaže z nich ty, které již nejsou nijak dostupné. V praxi se k úklidu paměti často používá algoritmus označovaný jako *mark & sweep* (česky by se dalo nazvat „označ a zamet“), jehož činnost si nyní zevrubně popíšeme.

Nejprve podotkněme, že paměť v níž jsou uloženy reprezentace elementů, se nazývá *halda*. Pojem *halda* se v informatice používá ještě v jiném významu (speciální kořenový strom) a čtenářům bude asi důvěrně známý z kursu algoritmické matematiky. V našem kontextu je však „halda“ pouze název pro úsek paměti. Název algoritmu napovídá, že algoritmus má dvě části. První část, zvaná *mark*, spočívá v tom, že se projde přes dosažitelné elementy a to tak, že se začne elementy jež mají vazbu v globálním prostředí (ty jsou zcela jistě dosažitelné). Pokud právě označený element ukazuje na další elementy, i ty budou dále zpracovány (protože jsou tím pádem taky dosažitelné). To se týká například *tečkových párů* (každý pár se odkazuje na první a druhou složku) a *uživatelsky definovaných procedur* (každá uživatelsky definovaná procedura ukazuje na seznam argumentů, tělo a prostředí). Takto se postupuje, dokud je co označovat. V druhém kroku nastupuje fáze *sweep*. Garbage collector projde celou haldu a pro každý element provede následující:

- pokud má element značku, značka je smazána a pokračuje se dalším elementem,
- pokud element značku nemá, pak je element odstraněn z paměti.

Po dokončení cesty haldou jsme v situaci, kdy byly nedostupné elementy smazány a všechny značky byly odstraněny. Mazání značek bylo provedeno jako inicializace pro další spuštění garbage collectoru.

5.7 Odvozené procedury pro práci se seznamy

V této sekci uvedeme řadu procedur pro vytváření a manipulaci se seznamy. Na implementaci těchto procedur pak ukážeme použití seznamů a základních procedur pro práci s nimi, které jsme představili v předchozích lekcích.

Prvními dvěma procedurami, které uvedeme, budou procedury pro zdvojení prvního prvku seznamu (procedura `dupf`) a zdvojení posledního prvku seznamu (procedura `dupl`). Bude se jednat o procedury o jednom argumentu, kterým bude seznam. Vracet budou seznam, obsahující duplikovaný první prvek seznamu v případě `dupf`, respektive duplikovaný poslední prvek seznamu v případě `dupf`. Takže například:

```
(dupf '(a))         ⟼  (a a)
(dupf '(1 2 3))     ⟼  (1 1 2 3)
(dupf '((1 2) 3))   ⟼  ((1 2) (1 2) 3)
```

```
(dupl '(a))         ⟼  (a a)
(dupl '(1 2 3))     ⟼  (1 2 3 3)
(dupl '(1 (2 3)))   ⟼  (1 (2 3) (2 3))
```

Duplikace prvního prvku seznamu vlastně odpovídá konstrukci hlavy seznamu na samotný seznam. Proceduru `dupf` tedy jednoduše naprogramujeme takto:

```
(define dupf
  (lambda (l)
    (cons (car l) l)))
```

Všimněte si, že v těle předchozího λ-výrazu používáme jen procedury `cons` a `car`, které známe z předchozí lekce. Jde nám o připojení prvního prvku na začátek celého seznamu. První prvek získáme aplikací procedury `car` a připojíme jej na začátek seznamu použitím procedury `cons`.

Nyní zdvojení posledního prvku. Postupujeme podle následující úvahy: Chceme-li se dostat k poslednímu prvku, obrátíme seznam pomocí procedury `reverse` a vezmeme první prvek z tohoto obráceného seznamu:

```
(reverse '(a b c))        ⟼  (c b a)
(car (reverse '(a b c)))  ⟼   a
```

Poslední prvek zdvojíme tak, že zdvojíme první prvek obráceného seznamu, a výsledek pak ještě jednou obrátíme:

```
(dupf (reverse '(a b c)))            ⟼  (c c b a)
(reverse (dupf (reverse '(a b c))))  ⟼  (a b c c)
```

Celá procedura tedy bude vypadat takto:

```
(define dupl
  (lambda (l)
    (reverse (dupf (reverse l)))))
```

Kdybychom chtěli vytvořit proceduru pro generování seznamu čísel od a do b, můžeme to udělat následujícím způsobem pomocí procedury `build-list`. Seznam čísel od a do b bude mít délku $b - a + 1$. Tvořící procedura pak bude pro index i vracet číslo $i + a$, které bude vloženo na i-tou pozici v seznamu (počítáno od 0). Přepsáno do Scheme:

```
(define range
  (lambda (a b)
    (build-list (+ 1 (- b a))
                (lambda (i) (+ i a)))))
```

Další dvě procedury budou také využitím procedury `build-list`. Konkrétně se bude jednat o vypuštění n-tého prvku seznamu a vsunutí prvku na n-tou pozici.

```
(define remove
  (lambda (l n)
    (build-list (- (length l) 1)
                (lambda (i)
                  (if (< i n)
                      (list-ref l i)
                      (list-ref l (+ i 1)))))))
```

Pro ilustraci viz následující příklady použití procedury:

```
(define s '(a b c d e))
(remove s 0)  ⟼  (b c d e)
(remove s 2)  ⟼  (a b d e)
(remove s 4)  ⟼  (a b c d)
```

Při definici procedury `remove` jsme použili procedury `build-list`, abychom s ní vytvořili seznam, jehož délka je o jedna kratší. Tvořící procedura pak aplikací procedury `list-ref` vybírá prvky z původního

seznamu. Tyto prvky bereme z odpovídajících indexů, v případě, že jde o indexy menší, než je pozice odstraňovaného prvku `n`, nebo z indexů o jedna vyšších.

Analogickou procedurou je procedur `insert`, vkládající prvek na danou pozici:

```
(define insert
  (lambda (l n elem)
    (build-list (+ (length l) 1)
                (lambda (i)
                  (cond ((< i n) (list-ref l i))
                        ((= i n) elem)
                        (else (list-ref l (- i 1))))))))
```

Proceduru `insert` používáme takto:

```
(define s '(a b c d e))
(insert s 0 666)  ⟼  (666 a b c d e)
(insert s 1 666)  ⟼  (a 666 b c d e)
(insert s 3 666)  ⟼  (a b c 666 d e)
(insert s 5 666)  ⟼  (a b c d e 666)
```

Další dvě procedury, které napíšeme pomocí procedur představených v sekci 5.3 budou procedury nahrazování prvků v seznamu. Jejich společným rysem je použití procedury `map`. Mapovaná procedura bude vypadat v obou případech podobně – vznikne vyhodnocením *lambda*-výrazu:

`(lambda (x) (if` ⟨*podmínka*⟩ `new x))`.

Procedura tedy podle podmínky ⟨*podmínka*⟩ buďto nahrazuje původní prvek prvkem `new` nebo vrací původní prvek `x`. Nahrazování na základě splnění predikátu `prop?` by mohlo být implementováno takto:

```
(define replace-prop
  (lambda (sez new prop?)
    (map (lambda (x) (if (prop? x) new x))
         sez)))
```

Druhou procedurou nahrazení je nahrazování podle vzoru reprezentovaného seznamem pravdivostních hodnot určujících zda-li se má prvek na dané pozici zachovat nebo nahradit novým elementem. Ukažme si nejprve použití této procedury:

```
(replace-pattern '(1 2 3 4) '(#f #f #f #f) 'x)  ⟼  (1 2 3 4)
(replace-pattern '(1 2 3 4) '(#t #f #t #f) 'x)  ⟼  (x 2 x 4)
(replace-pattern '(1 2 3 4) '(#f #t #t #t) 'x)  ⟼  (1 x x x)
(replace-pattern '(1 2 3 4) '(#t #t #t #t) 'x)  ⟼  (x x x x)
```

Proceduru `replace-pattern` můžeme naprogramovat takto:

```
(define replace-pattern
  (lambda (l pattern new)
    (map (lambda (x y)
           (if y new x))
         l pattern)))
```

Další odvozenou procedurou je procedura `list-pref` vracející n-prvkový prefix seznamu. Připomeňme, že n-prvkovým prefixem seznamu rozumíme seznam obsahující prvních n prvků seznamu:

```
(define list-pref
  (lambda (l n)
    (build-list n
                (lambda (i)
                  (list-ref l i)))))
```

V předchozím řešení jsme vytvořili procedurou `build-list` seznam o délce n, prvky do něj pak vybíráme z původního seznamu pomocí procedury `list-ref` z odpovídajících indexů. Podobným způsobem můžeme napsat proceduru vracející n-prvkový suffix (posledních n prvků seznamu) seznamu:

```
(define list-suff
  (lambda (l n)
    (let ((len (length l)))
      (build-list n
                  (lambda (i)
                    (list-ref l (- len (- n i))))))))
```

Další procedury budou procedury *rotací seznamu*. Rotace seznamu doleva je procedurou o jednom argumentu. Tímto argumentem je seznam a procedura vrací seznam, který má oproti původnímu seznamu prvních n prvků na konci a ostatní posunuté vlevo. Tedy například:

```
(rotate-left (1 2 3 4) 1)  ⟼  (2 3 4 1)
(rotate-left (1 2 3 4) 2)  ⟼  (3 4 1 2)
(rotate-left (1 2 3 4) 3)  ⟼  (4 1 2 3)
(rotate-left (1 2 3 4) 4)  ⟼  (1 2 3 4)
```

Analogicky pak bude fungovat procedura rotace seznamu doprava:

```
(rotate-right (1 2 3 4) 1)  ⟼  (4 1 2 3)
(rotate-right (1 2 3 4) 2)  ⟼  (3 4 1 2)
(rotate-right (1 2 3 4) 3)  ⟼  (2 3 4 1)
(rotate-right (1 2 3 4) 4)  ⟼  (1 2 3 4)
```

Rotaci seznamu doleva `rotate-left` můžeme implementovat následujícím způsobem. Pomocí procedury `build-list` vytvoříme nový seznam stejné délky, jako je ten původní, na jednotlivé pozice vybíráme prvky z původního seznamu použitím procedury `list-ref`, a to z pozice vždy o `n` vyšší. Přetečení prvního prvku na konec seznamu je zajištěno použitím procedury `modulo` (vrácení celočíselného zbytku po dělení).

```
(define rotate-left
  (lambda (l n)
    (let ((len (length l)))
      (build-list len
                  (lambda (i)
                    (list-ref l (modulo (+ i n) len)))))))
```

Procedura rotace seznamu doprava bude velice podobná:

```
(define rotate-right
  (lambda (l n)
    (let ((len (length l)))
      (build-list len
                  (lambda (i)
                    (list-ref l (modulo (- i n) len)))))))
```

Předchozí dvě procedury – `rotate-left` a `rotate-right` – můžeme sjednotit pomocí proceduru vyššího řádu *selekce*. Ta bere dva argumenty, prvním z nich je seznam $(E_0\ E_1 \cdots E_{n-1})$, tak jako u procedur rotace, a druhým je procedura reprezentující zobrazení $f : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$. Procedura selekce vrací seznam ve tvaru

$$(E_{f(0,n)}\ E_{f(1,n)} \cdots E_{f(n-1,n)}),$$

to jest seznam vzniklý z výchozího výběrem elementů určeným procedurou reprezentující f. Obecný kód procedury `select` by tedy mohl vypadat následovně:

```
(define select
  (lambda (l sel-f)
    (let ((len (length l)))
      (build-list len
                  (lambda (i)
                    (list-ref l (modulo (sel-f i len) len)))))))
```

Pomocí `select` můžeme vytvořit procedury rotace seznamu, které jsme uvedli výše:

```
(define rotate-left
  (lambda (l n)
    (select l (lambda (i len) (+ i n)))))

(define rotate-right
  (lambda (l n)
    (select l (lambda (i len) (- i n)))))
```

K implementaci `rotate-left` a `rotate-right` stačilo pouze provést specifikaci procedury reprezentující zobrazení $f : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$. V obou případech byla f definována tak, že $f(i, j) = i \pm k$, kde k počet prvků o kolik se má rotace provést (druhý argument j se v tomto případě neuplatnil).

Pomocí procedury `select` můžeme vytvořit ale například i proceduru `reverse`, kterou jsme představili v sekci 5.3:

```
(define reverse
  (lambda (l)
    (select l (lambda (i len) (- len i 1)))))
```

Nebo třeba proceduru na vytvoření seznamu náhodně vybraných prvků z původního seznamu:

```
(define random-select
  (lambda (l n)
    (select l (lambda (i len) (random len)))))
```

V posledním případě již přísně vzato druhý argument předaný `select` při aplikaci nebyla procedura reprezentující zobrazení. To ale v zásadě ničemu nevadí, viz diskusi v sekci 2.5 na straně 56.

5.8 Zpracování seznamů obsahujících další seznamy

V této sekci si ukážeme použití seznamů obsažených v seznamech. A to na dvou konkrétních problémových doménách – na práci s datovými tabulkami a na počty s maticemi.

V první části této sekce použijeme seznamy obsahující další seznamy k reprezentaci datových tabulek. Seznam seznamů o stejných délkách můžeme považovat za tabulku s informacemi. Předvedeme si implementaci několika základních operací s datovými tabulkami. Tabulkou tedy budeme rozumět seznam řádků. A řádky budou seznamy o stejné délce. Jako příklad uvádíme tabulku `mesta`:

```
(define mesta
  '((Olomouc 120 3)
    (Prostejov 50 2)
    (Praha 1200 8)))
```

Proceduru vracející n-tý řádek tabulky napíšeme velice jednoduše. Jelikož tabulka je seznam řádků, jde nám o n-tý prvek tohoto seznamu. Ten získáme pomocí procedury `list-ref`:

```
(define n-th-row
  (lambda (table n)
    (list-ref table n)))
```

Což bychom mohli napsat také stručněji následujícím způsobem:

```
(define n-th-row list-ref)
```

O něco složitější by bylo naprogramovat proceduru vracející „n-tý sloupec“ reprezentovaný jako seznam hodnot v tomto sloupci. Z každého řádku umíme vybrat n-tý prvek pomocí procedury `list-ref`. A jelikož máme všechny řádky v jednom seznamu – tabulce – můžeme přes tento seznam mapovat proceduru, která vznikne vyhodnocením λ-výrazu

```
(lambda (row) (list-ref row n)).
```

Tím dostaneme n-tou hodnotu z každého řádku, tedy n-tý sloupec.

```
(define n-th-column
  (lambda (table n)
    (map (lambda (row)
           (list-ref row n))
         table)))
```

Pro datové tabulky existuje celá řada operací, které mají význam v relační algebře. Jednou z nich je *projekce tabulky*, což je vytvoření nové tabulky z tabulky výchozí výběrem některých jejích sloupců. To jest, výsledkem aplikace projekce bude nová tabulka sestavená z některých sloupců původní tabulky. Proceduru provádějící projekci bychom mohli vytvořit například následujícím způsobem:

```
(define projection
  (lambda (table id-list)
    (map (lambda (row)
           (map (lambda (n)
                  (list-ref row n))
                id-list))
         table)))
```

Viz příklad použití projekce na tabulku navázanou na `mesta`:

```
(projection mesta '(0 2))  ⟹  ((Olomouc 3)
                               (Prostejov 2)
                               (Praha 8))
```

Při použití projekce jsme tedy kromě samotné tabulky ještě předávají seznam sloupců, které budeme chtít v nové tabulce zachovat. Podotkněme, že při implementaci procedury realizující projekci bychom též mohli použít proceduru `n-th-column`, kterou již máme naprogramovanou.

Druhá problémová doména, na které ukážeme využití seznamů a procedur na manipulaci s nimi, budou problematika týkající se číselných matice a operací nad nimi. Matici budeme reprezentovat podobně jako datovou tabulku. Tedy matice bude seznam řádků a řádek přitom bude seznam. Na rozdíl od datové tabulky však tyto řádky budou obsahovat pouze čísla.

První procedurou bude obdoba konstruktoru `build-list`. Chceme vytvořit konstruktor `build-matrix`, který bere jako parametry dvě čísla, představující rozměry matice, a jednu „tvořící“ proceduru, tentokrát však o dvou argumentech. To proto, že vstupem pro tvořící proceduru nebude jen jeden index, tedy pozice v seznamu, ale číslo řádku a číslo sloupce. Procedura `build-matrix` bude vypadat následovně:

```
(define build-matrix
  (lambda (rows cols f)
    (build-list rows
                (lambda (i)
                  (build-list cols
                              (lambda (j)
                                (f i j)))))))
```

V těle předchozí procedury jsme použili konstruktor `build-list` na vytvoření seznamu řádků. Každý z těchto řádků jsme vytvořili opět konstruktorem `build-list`. Viz příklad použití procedury:

```
(build-matrix 2 3 (lambda (x y) (- x y)))  ⟹  ((0 -1 -2) (1 0 -1))
```

Dále vytvoříme procedury pro operace sčítání a odčítání matic. Budou to procedury dvou argumentů, kterými budou matice. Vracet budou novou matici, jejíž prvky budou součty prvků sčítaných matic na stejných pozicích. Implementovány jsou pomocí procedury `map`.

```
(define matrix-+
  (lambda (m1 m2)
    (map (lambda (row1 row2)
           (map + row1 row2))
         m1 m2)))
```

```
(define matrix--
  (lambda (m1 m2)
    (map (lambda (row1 row2)
           (map - row1 row2))
         m1 m2)))
```

Předchozí dvě procedury se odlišovaly jen v použité proceduře při výpočtu součtu nebo rozdílu hodnot. Nabízí se tedy sjednotit obě procedury do jediné procedury vyššího řádu `matrix-map`. Ta bude, oproti procedurám na sčítání a odčítání matic, brát jeden argument navíc. Tímto argumentem bude procedura, která se bude aplikovat na prvky na stejné pozici.

```
(define matrix-map
  (lambda (m1 m2 f)
    (map (lambda (row1 row2)
           (map f row1 row2))
         m1 m2)))
```

Procedury sčítání a odčítání matic můžeme pomocí procedury `matrix-map` nadefinovat například takto:

```
(define matrix-+
  (lambda (m1 m2)
    (matrix-map m1 m2 +)))

(define matrix--
  (lambda (m1 m2)
    (matrix-map m1 m2 -)))
```

Shrnutí

V této lekci jsme zavedli seznamy jako struktury konstruované pomocí párů a speciálního elementu jazyka – – prázdného seznamu. Upřesnili jsme externí reprezentaci seznamů. Vysvětlili jsme, jaký je vztah mezi seznamy chápanými jako data a seznamy chápanými jako symbolické výrazy. Interní reprezentací seznamů (symbolických výrazů) jsou právě seznamy konstruované z párů. Naopak, externí reprezentace seznamů (elementů jazyka), které mají čitelnou externí reprezentaci, je ve tvaru symbolických výrazů (seznamů). Ukázali jsme několik typických procedur pro manipulaci se seznamy. Zaměřili jsme se především na konstruktory seznamů, procedury pro spojování seznamů, obracení seznamů, mapovací proceduru a další. Pozornost jsme věnovali i typovému systému jazyka Scheme. Vysvětlili jsme, jaký je rozdíl mezi statickým a dynamickým typováním; silně a slabě typovanými jazyky; a bezpečným a nebezpečným typováním. Věnovali jsme se i vybraným technických aspektům souvisejícím s konstrukcí interpretů jazyka Scheme. Konkrétně jsme se zabývali problémem automatické správy paměti. V závěru lekce jsme ukázali několik příkladů demonstrující praktickou práci se seznamy.

Pojmy k zapamatování

- seznam, prázdný seznam, hlava seznamu, ocas seznamu,
- kvotování seznamu,
- mapování procedury přes seznam, reverze seznamu, spojování seznamů,
- správa paměti, garbage collector, algoritmus mark & sweep,
- typový systém,
- silně/slabě typovaný jazyky,
- staticky/dynamicky typovaný jazyky,
- bezpečně/nebezpečně typovaný jazyky,

Nově představené prvky jazyka Scheme

- procedury `append`, `build-list`, `list`, `map`, `read`, `reverse`
- predikáty `boolean?`, `list?`, `null?`, `number?`, `pair?`, `procedure?`, `symbol?`

Kontrolní otázky

1. *Co je prázdný seznam?*
2. *Jak jsou definovány seznamy?*
3. *Jak jsme změnili boxovou notaci?*
4. *Jak jsme změnili tečkovou notaci?*
5. *Jaký je ve Scheme vztah mezi programy a daty?*
6. *Co je to garbage collector?*
7. *Popište algoritmy práce garbage collectoru.*
8. *Jak jsme naprogramovali seznamy pamatující si vlastní délku?*
9. *Jaký je rozdíl mezi silně a slabě typovaným jazykem?*
10. *Jaký je rozdíl mezi staticky a dynamicky typovaným jazykem?*
11. *Jaký je rozdíl mezi bezpečně a nebezpečně typovaným jazykem?*

Cvičení

1. Bez použití interpretu vyhodnoťte následující výrazy:

```
(cons 1 (cons (cons 2 '()) (cons 3 '())))          ⟼
(cons (cons '() '()) '())                          ⟼
(caddr '(1 2 3))                                   ⟼

(list '1 2 3)                                      ⟼
(list '(1 2 3))                                    ⟼
(list list)                                        ⟼
(list (+ 1 2))                                     ⟼
(list '+ 1 2)                                      ⟼
(list + 1 2)                                       ⟼
(list '(+ 1 2))                                    ⟼

(reverse '(1 2 3))                                 ⟼
(reverse '((1 2 3)))                               ⟼
(reverse '(1 (2 3) 4))                             ⟼

(build-list 5 -)                                   ⟼
(build-list 3 list)                                ⟼
(build-list 0 (lambda(x) nedef-symbol))            ⟼
(build-list 1 (lambda (x) '()))                    ⟼

(map number? '(1 (2 3) 4))                         ⟼
(map + '(1 2 3))                                   ⟼
(map + '(2 4 8) '(1 3 9))                          ⟼
(map (lambda (x) (cons 1 x)) '(2 4 8) '(1 3 9))    ⟼
(map (lambda (x) '()) '(1 2 3))                    ⟼
(map (lambda (x) nenavazany-symbol) '())           ⟼

(append 1 2)                                       ⟼
(append 1 '(2))                                    ⟼
(append '(1) '(2))                                 ⟼
(append (append))                                  ⟼
(append '(1 2))                                    ⟼
```

```
(append)                                          ⟼
(append '())                                      ⟼
(append '(1 2) '(3 4) (list 5))                   ⟼
(append '(list 5))                                ⟼

'(map (lambda (x) '()) 10)                        ⟼
(quote quote)                                     ⟼
(quote (quote (1 2 3)))                           ⟼
('quote (1 2 3))                                  ⟼
(car ''10)                                        ⟼
```

2. Naprogramujte proceduru o třech argumentech n, a, d, která vrací seznam prvních n členů aritmetické posloupnosti $\{a + nd\}_{n=0}^{\infty}$
3. Naprogramujte proceduru o třech číselných argumentech n, a a q, která vrací seznam prvních n členů geometrické posloupnosti $\{aq^n\}_{n=0}^{\infty}$.
4. Naprogramujte konstruktor diagonální čtvercové matice a pomocí něho konstruktor jednotkové matice.
5. Napište obdobu procedury `map`, která do mapované procedury dává nejen prvek seznamu ale i jeho index. Například:
   `(map cons '(a b c))` ⟼ `((a . 0) (b . 1) (c . 2))`
   `(map 2-ze-2 '(a b c))` ⟼ `(0 1 2)`

Úkoly k textu

1.

2.

3.

Řešení ke cvičením

1. (1 (2) 3), ((())), 3
   (1 2 3), ((1 2 3)), (*procedura*), (3), (+ 1 2), (*procedura* 1 2), ((+ 1 2))
   (3 2 1), ((1 2 3)), (4 (2 3) 1)
   (0 -1 -2 -3 -4), ((0) (1) (2)), (), (())
   (#t #f #t), (1 2 3), (3 7 17), (() () ()), (), (1 2), ()
   chyba, chyba, (1 2), (), (1 2), (), (), (1 2 3 4 5), (list 5)
   (map (lambda (x) '()) 10), quote, (quote (1 2 3)), chyba, quote
2. ```
   (define (arit first diff n)
     (build-list n (lambda (i) (+ first (* i diff)))))
   ```
3. ```
   (define (geom first quot n)
     (build-list n (lambda (i) (* first (expt quot i)))))
   ```
4. ```
   (define build-diag-matrix
     (lambda (diag-l)
       (let ((n (length diag-l)))
         (build-matrix n n
                       (lambda (x y)
                         (if (= x y)
                             (list-ref diag-l x)
                             0))))))

   (define build-unit-matrix
     (lambda (n)
   ```
   ```

```
    (build-diag-matrix
     (build-list n (lambda (i) 1)))))

(define map-index
  (lambda (f l)
    (map f l (build-list (length l) +))))
```