KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 6: Explicitní aplikace a vyhodnocování

Učební materiál k přednášce 9. 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 6: Explicitní aplikace a vyhodnocování

Obsah lekce: V předchozích lekcích jsme popsali aplikaci procedur jakožto jednu z částí abstraktního interpretu. Část interpretu odpovědnou za aplikaci jsme označovali „Apply“. Stejně tak jsme popsali část interpretu provádějící vyhodnocování elementů a označovali jsme ji jako „Eval“. V této lekci ukážeme, že „Apply“ a „Eval“ lze bez újmy chápat jako procedury abstraktního interpretu, které mohou programátoři kdykoliv využít k přímé aplikaci procedur na seznamy hodnot a k vyhodnocování libovolných elementů v daném prostředí. Dále ukážeme, že prostředí je možné chápat jako element prvního řádu.

Klíčová slova: aplikace procedur, procedura `apply`, procedura `eval`, prostředí, vyhodnocování elementů.

6.1 Explicitní aplikace procedur

Nyní se budeme věnovat aplikaci procedur. V lekci 1 jsme uvedli, že aplikace procedur probíhá v momentě, kdy se první prvek seznamu vyhodnotil na proceduru. Procedura vzniklá vyhodnocením prvního prvku seznamu je aplikována s argumenty jimiž jsou elementy vzniklé *vyhodnocením* všech dalších prvků seznamu. Této aplikaci procedur můžeme říkat *implicitní aplikace,* protože je prováděna implicitně během vyhodnocování elementů (seznamů). V některých případech bychom ale mohli chtít provést *explicitní aplikaci* dané procedury s argumenty, které máme k dispozici ve formě seznamu – tedy *argumenty již máme k dispozici* a nechceme je získávat vyhodnocením (nějakých výrazů).

Předpokládejme pro ilustraci, že na symbol `s` máme navázaný seznam čísel, který jsme získali jako výsledek aplikace nějakých procedur. Z nějakého důvodu bychom třeba mohli chtít provést součet všech čísel ze seznamu (navázaného na) `s`. Kdybychom mohli v programu zaručit, že tento seznam bude mít vždy pevnou velikost (vždy stejný počet prvků), pak bychom mohli jeho prvky sečíst pomocí

`(+ (car s) (cadr s) (caddr s) ···)`.

Co kdybychom ale délku seznamu dopředu neznali? Pak bychom zřejmě předchozí výraz nemohli přesně napsat, protože bychom neznali počet argumentů, které bychom měli předat proceduře sčítání při její aplikaci. Další problém předchozího kódu je jeho *neefektivita.* Pro seznam délky n potřebujeme provést celkem $\frac{n(1+n)}{2}$ aplikací procedur `car` a `cdr` k tomu, abychom vyjádřili všechny argumenty pomocí `car`, `cadr`, `caddr`, a tak dále. Předchozí řešení tedy není ani univerzální ani efektivní.

Předchozí problém by bylo možné snadno vyřešit, kdybychom měli v jazyku Scheme k dispozici Apply (viz definici 2.12 na straně 49) jako primitivní proceduru vyššího řádu, které bychom mohli předat

(i) *proceduru* E, kterou chceme aplikovat,

(i) *seznam argumentů* $E_1, \ldots, E_n$, se kterými chceme proceduru E aplikovat,

a která by vrátila hodnotu aplikace $\mathrm{Apply}[E, E_1, \ldots, E_n]$.

V jazyku Scheme je taková primitivní procedura skutečně k dispozici. Část interpretu „Apply“, která je odpovědná za aplikaci procedur, je tedy přístupná programátorovi pomocí primitivní procedury vyššího řádu (vyššího řádu proto, že jedním z předaných argumentů je samotná procedura, kterou chceme aplikovat). Tato primitivní procedura je navázána na symbol `apply` a nyní si ji popíšeme.

Definice 6.1 (primitivní procedura `apply`). Primitivní procedura `apply` se používá s argumenty ve tvaru:

`(apply` ⟨*procedura*⟩ ⟨arg_1⟩ ⟨arg_2⟩ $\cdots$ ⟨arg_n⟩ ⟨*seznam*⟩`)`,

přitom ⟨arg_1⟩, …, ⟨arg_n⟩ jsou *nepovinné a mohou být vynechány.* Jelikož je `apply` procedura, je aplikována s argumenty v jejich vyhodnocené podobě. Při aplikaci procedura `apply` nejprve ověří, zda-li je element ⟨*procedura*⟩ primitivní nebo uživatelsky definovaná procedura. Pokud by tomu tak nebylo, ukončí se aplikace hlášením „CHYBA: První argument předaný `apply` musí být procedura.“ V opačném případě procedura `apply` *sestaví seznam hodnot* ze všech ostatních argumentů, takto: první prvek seznamu hodnot bude ⟨arg_1⟩, druhý prvek seznamu hodnot bude ⟨arg_2⟩, … n-tý prvek seznamu hodnot bude ⟨arg_n⟩ a další prvky seznamu hodnot budou tvořeny prvky ze seznamu ⟨*seznam*⟩. Pokud by poslední argument uvedený při

aplikaci `apply` (to jest argument ⟨*seznam*⟩) nebyl seznam, pak se aplikace ukončí hlášením „CHYBA: Poslední argument předaný `apply` musí být seznam.“ Výsledkem aplikace `apply` je hodnota vzniklá aplikací procedury ⟨*procedura*⟩ s argumenty jimiž jsou elementy ze *sestaveného seznamu hodnot*. ■

Z předchozí definice je tedy jasné, že `apply` musí být volána alespoň se dvěma argumenty, první z nich se musí vyhodnotit na proceduru (kterou aplikujeme) a poslední z nich se musí vyhodnotit na seznam argumentů, které chceme při aplikaci použít (seznam může být prázdný).

Nyní ukážeme několik vysvětlujících příkladů použití `apply`, praktické příklady budou následovat v dalších sekcích. Nejprve se zaměříme na použití `apply` pouze se dvěma argumenty, tedy bez $\langle arg_1 \rangle, \ldots, \langle arg_n \rangle$, viz definici 6.1. Vyhodnocení následujících výrazů vede na součet čísel (všimněte si použití `apply`):

```
(+ 1 2 3 4 5)                ⟹  15
(apply + (list 1 2 3 4 5))   ⟹  15
(apply + '(1 2 3 4 5))       ⟹  15
(apply + 1 2 3 4 5)          ⟹  „CHYBA: Poslední argument předaný apply musí být seznam.“
```

V posledním případě při aplikaci došlo k chybě, protože posledním argumentem předaným `apply` nebyl seznam (ale číslo `5`). Při použití `apply` se seznamem hodnot si ale musíme dát pozor na to, že hodnoty v seznamu (poslední argument `apply`) se používají jako argumenty při aplikaci a nejsou před aplikací vyhodnoceny. Všimněte si například rozdílu v následujících dvou aplikacích:

```
(apply + (list 1 (+ 1 2) 5))  ⟹  9
(apply + '(1 (+ 1 2) 5))      ⟹  „CHYBA: Pokus o sčítání čísla se seznamem.“
```

Uvědomte si, že v prvním případě v ukázce vznikl vyhodnocením výrazu `(list 1 (+ 1 2) 5)` tříprvkový seznam `(1 3 5)`, takže došlo k sečtení těchto hodnot. V druhém případě byla ale procedura sčítání aplikována se seznamem `(1 (+ 1 2) 5)` jehož druhým prvkem není číslo, takže aplikace sčítání na argumenty `1` (číslo), `(+ 1 2)` (seznam) a `5` (číslo) selhala.

Pokud se nyní vrátíme k našemu motivačnímu příkladu, tak bychom sečtení hodnot v seznamu navázaném na `s` provedli následovně:

```
(apply + s)
```

Použití `apply` je tedy vhodné v případě, kdy máme dány argumenty (se kterými chceme provést aplikaci nějaké procedury) v seznamu. Nemusíme přitom předávané argumenty nijak „ručně vytahovat“ ze seznamu (jak to bylo naznačeno na začátku této sekce). V případech, kdybychom neznali délku seznamu, protože by byla proměnlivá, by to stejně k uspokojivému řešení nevedlo.

Následující příklady ukazují další použití `apply`:

```
(apply +)                                  ⟹  „CHYBA: Chybí seznam hodnot.“
(apply + '())                              ⟹  0
(apply append '())                         ⟹  ()
(apply append '((1 2 3) () (c d) (#t #f))) ⟹  (1 2 3 c d #t #f)
(apply list '(1 2 3 4))                    ⟹  (1 2 3 4)
(apply cons (list 2 3))                    ⟹  (2 . 3)
(apply cons '(1 2 3 4))                    ⟹  „CHYBA: cons má mít dva argumenty.“
(apply min '(4 1 3 2))                     ⟹  1
```

Doposud jsme ukazovali pouze příklady použití `apply` se dvěma argumenty, to jest s procedurou a seznamem hodnot. Nyní si ukážeme příklady použití `apply` s více argumenty (viz definici 6.1). Nepovinné argumenty, které se nacházejí za předanou procedurou a před posledním seznamem (který musí být vždy přítomen) jsou při aplikaci předávány tak, jak jsou uvedeny. Následující příklad ukazuje několik možností sečtení čísel z daného seznamu za použití nepovinných argumentů `apply`:

```
(apply + 1 2 3 4 '())  ⟹  10
(apply + 1 2 3 '(4))   ⟹  10
(apply + 1 2 '(3 4))   ⟹  10
(apply + 1 '(2 3 4))   ⟹  10
(apply + '(1 2 3 4))   ⟹  10
```

V prvním případě byly nepovinné argumenty čtyři a povinný poslední seznam byl prázdný. V druhém případě byly nepovinné argumenty tři a poslední seznam byl jednoprvkový a tak dále. Uvědomte si, že nepovinné argumenty budou před aplikací `apply`, a tedy i před explicitní aplikací předané procedury, vyhodnoceny. Viz rozdíl mezi následujícími aplikacemi:

```
(apply + '((+ 1 2) 10))  ⟹  „CHYBA: Pokus o sčítání čísla se seznamem.“
(apply + (+ 1 2) '(10))  ⟹  13
```

Otázkou je, kdy použít tyto nepovinné argumenty. Používají se v případě, kdy potřebujeme aplikovat proceduru se seznamem hodnot, ale k tomuto seznamu ještě potřebujeme další hodnoty (zepředu) dodat. Kdybychom například chtěli sečíst všechny hodnoty v seznamu navázaném na `l` s číslem `1`, pak bychom to mohli udělat následujícími způsoby:

```
(define l '(10 20 30 40 50))
(apply + (cons 1 l))          ⟹  151
(+ 1 (apply + l))             ⟹  151
(apply + 1 l)                 ⟹  151
```

V prvním případě jsme ručně jedničku připojili jako nový první prvek seznamu hodnot. V druhém případě jsme přičtení jedničky provedli až po samotné aplikaci. V tomto případě to bylo možné, ale u některých procedur bychom to takto řešit nemohli (uvidíme dále). Poslední příklad byl nejkratší a nejčistší, využili jsme jeden nepovinný argument – přidání jedničky k seznamu argumentů za nás vyřeší procedura `apply`.

Následující příklad je o něco složitější na představivost, ale ukazuje praktičnost nepovinných argumentů, které lze předat proceduře `apply`. Uvažujme tedy aplikaci:

```
(apply map list '((a b c) (1 2 3)))  ⟹  ((a 1) (b 2) (c 3))
```

Při této aplikaci došlo k aplikování `map` jehož prvním argumentem byla procedura navázaná na `list`, druhý a třetí argument byly prvky seznamu `((a b c) (1 2 3))`. Předchozí aplikaci si tedy můžeme představit jako vyhodnocení následujícího výrazu:

```
(map list
     '(a b c)
     '(1 2 3))  ⟹  ((a 1) (b 2) (c 3))
```

Nyní by již výsledná hodnota měla být jasná. Procedura `map` má první argument jiného významu než ostatní argumenty, technicky bychom tedy nemohli provést trik jako byl v případě `+` proveden na třetím řádku v předchozí ukázce.

Na aplikaci procedur na seznamy argumentů se lze dívat jako na způsob *agregace elementů seznamu do jediné hodnoty*. Aplikaci pomocí `apply` používáme v případě, kdy máme vytvořený seznam hodnot, třeba nějakým předchozím výpočtem, a chceme na tento celý seznam aplikovat jednu proceduru. Při aplikaci (tímto způsobem) je potřeba pamatovat na to, že prvky v seznamu předaném `apply` jsou aplikované proceduře předány „jak leží a běží“, tedy bez dodatečného vyhodnocování.

6.2 Použití explicitní aplikace procedur při práci se seznamy

V minulé lekci jsme ukázali řadu procedur pro práci se seznamy. Pro některé z těchto procedur jsme ukázali, jak bychom je mohli naprogramovat, kdybychom je v jazyku Scheme implicitně neměli. Nyní budeme v této problematice pokračovat a zaměříme se přitom na využití `apply`.

Prvním problémem bude stanovení *délky seznamu*. V předchozí lekci jsme ukázali proceduru `length`, která pro daný seznam vrací počet jeho prvků. Nyní ukážeme, jak tuto proceduru naprogramovat. Předpokládejme, že máme na symbol `s` navázaný nějaký seznam, třeba:

```
(define s '(a b c d e f g))
s                             ⟹  (a b c d e f g)
```

Jak napsat výraz, jehož vyhodnocením bude délka seznamu navázaného na `s`? Jde nám přitom o to, nalézt obecné řešení. To jest, když změníme seznam `s`, vyhodnocením výrazu by měla být délka modifikovaného seznamu. Délku seznamu získáme tak, když *sečteme počet jeho prvků*. Při výpočtu délky by tedy zřejmě mělo hrát roli *sčítání*. Nejde však o sčítání elementů nacházejících se v seznamu (což nemusí být ani čísla, jako v našem případě), ale o jejich „počet“. V seznamu se na každé jeho pozici nachází *jeden* element. Na základě seznamu `s` tedy můžeme vytvořit seznam stejné délky tak, že každý prvek ze seznamu `s` zaměníme za `1`:

```
(map (lambda (x) 1) s)  ⟼  (1 1 1 1 1 1 1)
```

K záměně jsme použili mapování konstantní procedury vracející pro každý argument číslo `1`. Nyní již stačí aplikovat na vzniklý seznam proceduru sčítání. Takže délku seznamu navázaného na `s` spočítáme pomocí:

```
(apply + (map (lambda (x) 1) s))  ⟼  7
```

Zopakujme si ještě jednou použitou myšlenku: klíčem k vypočtení délky seznamu bylo „sečíst tolik jedniček, kolik bylo prvků ve výchozím seznamu“. Potřebovali jsme tedy udělat jeden mezikrok, kdy jsme z daného seznamu vytvořili seznam stejné délky obsahující samé jedničky, pak již jen stačilo aplikovat sčítání. Uvědomte si zde dobře roli `apply`. Procedura sčítání je aplikována na seznam „neznámé délky“ (délku seznamu se teprve snažíme zjistit) obsahující hodnoty, se kterými chceme proceduru sčítání aplikovat; například tedy výraz:

```
(+ (map (lambda (x) 1) s))  ⟼  „CHYBA: Nelze sčítat seznam.“
```

by tedy nebyl nic platný. Podle výše uvedeného postupu tedy můžeme naprogramovat proceduru pro výpočet délky seznamu jak je tomu v programu 6.1. Všimněte si, že procedura funguje skutečně stejně tak

Program 6.1. Výpočet délky seznamu.

```
(define length
  (lambda (l)
    (apply + (map (lambda (x) 1) l))))
```

jako procedura `length`, kterou jsme představili v předchozí lekci a to včetně mezního případu. To jest, délka prázdného seznamu je nula, viz následující příklady:

```
(length '(a b c d))      ⟼  4
(length '(a (b (c)) d))  ⟼  3
(length '())             ⟼  0
```

Na tomto příklady je dobře vidět, že pokud provedeme při programování správný myšlenkový rozbor problému, nemusíme se zabývat ošetřování okrajových případů, což často vede ke vzniku nebezpečných chyb, které nejsou vidět.

Pokud někteří čtenáři doposud pochybovali o užitečnosti definovat procedury jako jsou sčítání a násobení „bez argumentů“, tak nyní vidíme, že je to velmi výhodné. Kdyby nebylo sčítání definováno pro prázdný seznam (to jest `(+)` ⟼ `0`), tak bychom v proceduře `length` v programu 6.1 museli ošetřovat speciální případ, kdy bude na argument `l` navázán prázdný seznam. Kdybychom ošetření neprovedli, `length` by počítala pouze délky neprázdných seznamů. Při některých aplikacích `length` v programu by bylo nutné dělat dodatečné testy prázdnosti seznamu – což by vše jen komplikovalo kód.

Nyní ukážeme užitečnou a často používanou proceduru (vyššího řádu), která provádí *filtraci elementů v seznamu podle jejich vlastnosti* (predikát jednoho argumentu), která je dodaná spolu se seznamem formou argumentu. Pro objasnění si nejprve ukážeme několik příkladů, na kterých uvidíme, jak bychom chtěli proceduru používat:

```
(filter even? '(1 2 3 4 5 6))                    ⟹  (2 4 6)
(filter (lambda (x) (<= x 4)) '(1 2 3 4 5 6))    ⟹  (1 2 3 4)
(filter pair? '(1 (2 . a) 3 (4 . k)))            ⟹  ((2 . a) (4 . k))
(filter (lambda (x)
          (not (pair? x)))
        '(1 (2 . a) 3 (4 . k)))                  ⟹  (1 3)
(filter symbol? '(1 a 3 b 4 d))                  ⟹  (a b d)
```

V prvním případě procedura z daného seznamu čísel vyfiltrovala všechna sudá čísla, v druhém případě byla vyfiltrována čísla menší nebo rovna čtyřem, v třetím případě byly ze seznamu vyfiltrovány páry, v dalším případě vše kromě párů a v posledním případě symboly. Proceduru `filter` tedy chceme používat se dvěma argumenty: prvním je predikát jednoho argumentu a druhým je seznam. Chceme, aby procedura vrátila seznam, ve kterém budou právě ty prvky z výchozího seznamu, pro které je daný predikát pravdivý (přesněji: výsledek jeho aplikace je cokoliv kromě `#f`).

Při implementaci filtrace tedy musíme vyřešit problém, jak vynechávat prvky v seznamu. Zde bychom si mohli pomoci aplikací procedury pro *spojování seznamů*, protože při spojování nehrají roli „prázdné seznamy“, ve výsledku spojení jsou vynechány. Můžeme tedy říct, že výsledkem filtrace je spojení jednoprvkových seznamů obsahujících prvky z původního seznamu splňující danou vlastnost. V prvním kroku nám tedy z výchozího seznamu stačí vytvořit nový seznam, ve kterém budou: (i) všechny prvky splňující danou vlastnost obsaženy v jednoprvkových seznamech a (ii) místo ostatních prvků zde budou prázdné seznamy. K vytvoření tohoto seznamu můžeme použít `map`. Uvažujme následující seznam navázaný na `s` a vlastnosti reprezentovanou predikátem navázaným na `even?`:

```
(define s '(1 3 2 6 1 7 4 8 9 3 4))
s                                    ⟹  (1 3 2 6 1 7 4 8 9 3 4)
```

Pak následujícím mapováním získáme:

```
(map (lambda (x)
       (if (even? x)
           (list x)
           '()))
     s)  ⟹  (() () (2) (6) () () (4) (8) () () (4))
```

Na takto vytvořený seznam již jen stačí aplikovat `append`:

```
(apply append
       (map (lambda (x)
              (if (even? x)
                  (list x)
                  '()))
            s))  ⟹  (2 6 4 8 4),
```

což je výsledek filtrace sudých čísel z výchozího seznamu. Hlavní myšlenkou filtrace tedy bylo zřetězení jednoprvkových, případně prázdných, seznamů obsahující filtrované prvky (jednoprvkové seznamy obsahovaly právě prvky splňující vlastnost). Obecnou filtrační proceduru tedy můžeme naprogramovat zobecněním předchozího principu. Viz proceduru `filter` v programu 6.2. Podotkněme, že stejně jako tomu bylo v případě `length` máme naším přístupem opět automaticky vyřešen mezní případ filtrace prvků z prázdného seznamu. Viz příklady:

```
(filter even? '())           ⟹  ()
(filter (lambda (x) #f) '()) ⟹  ()
```

Filtrace je ve funkcionálních jazycích velmi oblíbená. Skoro každý funkcionální programovací jazyk je vybaven nějakou filtrační procedurou vyššího řádu. Pokud ne, lze ji snadno naprogramovat jako tomu bylo v programu 6.2. S pomocí filtrace lze naprogramovat celou řadu užitečných procedur. V programu 6.3 máme příklady dvou z nich. Procedura `remove` je vlastně jen „nepatrnou modifikací“ předchozí filtrační procedury, která spočívá v tom, že prvky splňující danou vlastnost jsou ze seznamu odstraňovány místo toho aby byly ponechávány. S pomocí `filter` již můžeme `remove` naprogramovat snadno – stačí, abychom

Program 6.2. Filtrace prvků seznamu splňujících danou vlastnost.

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

Program 6.3. Odstraňování prvků seznamu a test přítomnosti prvku v seznamu.

```
(define remove
  (lambda (f l)
    (filter (lambda (x)
              (not (f x)))
            l)))

(define member?
  (lambda (elem l)
    (not (null? (filter
                 (lambda (x)
                   (equal? x elem))
                 l)))))
```

totiž aplikovali `filter` s vlastností reprezentující negaci vlastnosti pro `remove`, viz program 6.3. Druhou procedurou v programu 6.3 je predikát `member?` testující přítomnost daného prvku v seznamu. Myšlenka této procedury je založena na tom, že daný prvek E je obsažen v seznamu, pokud vyfiltrováním prvků vlastnosti „prvek *je roven* E“ vznikne neprázdný seznam (to jest musí být v něm aspoň jeden prvek *roven* E). Viz příklady použití:

```
(member? 'a '())           ⟼  #f
(member? 'a '(1 2 3 4))    ⟼  #f
(member? 'a '(1 2 a 3 4))  ⟼  #t
```

V předchozí lekci jsme ukázali proceduru `list-ref`, která pro daný seznam a danou pozici (číslo) vrací prvek na dané pozici. Nyní si můžeme ukázat, jak lze pomocí filtrace danou proceduru naprogramovat, viz program 6.4. Myšlenka je v tomto případě následující. Procedura `list-ref` si nejprve s použitím procedury

Program 6.4. Procedura vracející prvek na dané pozici v seznamu.

```
(define list-ref
  (lambda (l index)
    (let ((indices (build-list (length l) (lambda (i) i))))
      (cdar
       (filter (lambda (cell) (= (car cell) index))
               (map cons indices l))))))
```

`build-list` vytvoří pomocný seznam indexů `(0 1 2 3` $\cdots$, který je stejně dlouhý jako vstupní seznam.

Pomocí mapování je potom vytvořen pomocný seznam párů ve tvaru `(`⟨*index*⟩ `.` ⟨*prvek*⟩`)`, přitom ⟨*prvek*⟩ je právě prvek seznamu na pozici ⟨*index*⟩. Pak už jen stačí vyfiltrovat z tohoto seznamu prvky (respektive prvek, bude jediný) s vlastností „první prvek páru (to jest index) má hodnotu danou argumentem `index`“. Nakonec stačí jen z tohoto páru vybrat druhý prvek, což je element na dané pozici.

Další zajímavou aplikací filtrace by mohla být procedura `list-indices`, která vlastně provádí opak toho, co procedura `list-ref`. Procedura akceptuje jako argumenty seznam a element a vrací *seznam pozic* (*indexů*), *na kterých se daný element v seznamu vyskytuje.* Obecně je toto řešení lepší než vracet například jen jednu (první) pozici, protože prvek se může v seznamu vyskytovat na víc místech. Proceduru máme uvedenu v programu 6.5. Princip jejího vytvoření je podobný jako u procedury `list-ref`. Opět si vytvoříme seznam

Program 6.5. Procedura vracející všechny pozice výskytu daného prvku.

```
(define list-indices
  (lambda (l elem)
    (let ((indices (build-list (length l) (lambda (i) i))))
      (remove null?
              (map (lambda (x id)
                     (if (equal? x elem)
                         id
                         '()))
                   l
                   indices)))))
```

pomocných indexů a mapováním přes předaný seznam a seznam indexů vytváříme nový seznam, který bude obsahovat buď indexy (v případě že na dané pozici prvek je), nebo prázdné seznamy. Z tohoto meziproduktu již nám pak stačí odstranit prázdné seznamy a získáme tak seznam indexů reprezentujících pozice všech výskytů daného prvku. Viz příklady použití:

```
(list-indices '(a b b a c d a) 'a)  ⟹  (0 3 6)
(list-indices '(a b b a c d a) 'd)  ⟹  (5)
(list-indices '(a b b a c d a) 10)  ⟹  ()
(list-indices '() 'a)               ⟹  ()
```

V sekci 5.8 jsme implementovali procedury pro vytváření matic, a také procedury pro jejich sčítání a odčítání. Nyní tuto sadu rozšíříme o transpozici matice `matrix-transpose` a násobení dvou matic `matrix-*`.

```
(define matrix-transpose
  (lambda (m)
    (apply map list m)))
```

Proceduru map jsme aplikovali na matici, která je reprezentovaná jako seznam seznamů. Mapováním procedury `list` na seznamy představující řádky, dostáváme seznam seznamů s čísly ve stejných sloupcích. Tento seznam můžeme považovat za transponovanou matici.

Násobení matic bychom mohli implementovat následovně. Abychom pracovali jen s řádky, transponujeme druhou matici použitím `matrix-transpose`. Každý řádek `x` první matice skalárně vynásobíme s každým řádkem `y` transponované druhé matice a dostaneme prvek výsledné matice:

```
(define matrix-*
  (lambda (m1 m2)
    (let ((t (matrix-transpose m2)))
      (map (lambda (x)
             (map (lambda (y)
                    (apply + (map * x y)))
                  t))
           m1))))
```

V sekci 5.8 jsme implementovali i selekce a projekce nad databázovými tabulkami. Tyto tabulky byly reprezentovány pomocí seznamů, jejichž prvky byly stejně dlouhé seznamy – řádky. Řádky při selekci byly přitom voleny na základě indexů těchto řádků. Pomocí filtrování můžeme vybírat řádky na základě predikátu. Procedura pro selekci (výběr řádků) bude vytvořena s využitím procedury `filter`:

```
(define selection
  (lambda (table property)
    (filter (lambda (x)
              (apply property x))
            table)))
```

Procedura selekce tak bude fungovat pro libovolný počet sloupců. Viz příklad použití:

```
(define mesta
  '((Olomouc 120 3 stredni)
    (Prostejov 45 2 male)
    (Prerov 50 3 male)
    (Praha 1200 8 velke)))

(selection mesta
           (lambda (jmeno p-obyvatel p-nadrazi velikost)
             (and (>= p-obyvatel 50)
                  (not (equal? velikost 'male)))))
  ⟹  ((olomouc 120 3 stredni)
       (praha 1200 8 velke))
```

6.3 Procedury s nepovinnými a libovolnými argumenty

Řada primitivních procedur, se kterými jsme se doposud setkali, umožňovala mít při jejich aplikaci některé argument nepovinné. Například procedura `map` musela mít k dispozici jako argument proceduru a seznam a volitelně jako nepovinné argumenty ji mohly být předány ještě další seznamy. Některé primitivní procedury, jako například `+`, `*` a `append` mohly být aplikovány dokonce s libovolným počtem argumentů, včetně žádného argumentu. V této sekci si ukážeme, jak lze vytvářet uživatelsky definované procedury s nepovinnými argumenty nebo s libovolnými argumenty.

Nejprve ukážeme, jak je možné vytvořit procedury, které mají několik povinných argumentů, které musejí být vždy uvedeny, a kromě nich mohou být předány další nepovinné argumenty. Platí podmínka, že *nepovinné argumenty* lze uvádět až *za všemi povinnými*. Při psaní λ-výrazů jejichž vyhodnocením mají vzniknout procedury pracující s nepovinnými argumenty, píšeme místo tradiční specifikace seznamu argumentů

$$(\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle)\,,$$

kterou jsme používali doposud, seznam argumentů ve tvaru

$$(\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle\ .\ \langle zbytek\rangle)\,,$$

kde $\langle param_1\rangle, \langle param_2\rangle, \ldots, \langle param_n\rangle, \langle zbytek\rangle$ jsou vzájemně různé symboly. To jest kromě povinných formálních argumentů (zapsaných jako dosud), jsme pomocí tečky „ . “ oddělili poslední symbol $\langle zbytek\rangle$. Přísně vzato, struktura argumentů zapsaná v tomto tvaru již *není seznam*, protože druhý prvek jeho „posledního páru“ není prázdný seznam. Úkol argumentu $\langle zbytek\rangle$ je následující. Při aplikaci procedury vzniklé vyhodnocením λ-výrazu se hodnoty všech povinných argumentů naváží na symboly $\langle param_1\rangle, \ldots, \langle param_n\rangle$ (jako doposud). Pokud byly navíc při aplikaci použity další argumenty, pak je vytvořen seznam všech těchto dodatečných argumentů a při aplikaci procedury je tento seznam navázaný na symbol $\langle zbytek\rangle$. Pokud tedy žádné nepovinné argumenty nebyly předány, na $\langle zbytek\rangle$ bude navázaný prázdný seznam.

Následující příklad demonstruje použití nepovinných argumentů:

```
((lambda (x y . rest) (list x y rest)) 1 2)       ⟹  (1 2 ())
((lambda (x y . rest) (list x y rest)) 1 2 3)     ⟹  (1 2 (3))
```

```
((lambda (x y . rest) (list x y rest)) 1 2 3 4 5)  ⟹  (1 2 (3 4 5))
((lambda (x y . rest) (list x y rest)) 1)          ⟹  „CHYBA: Chybí argument.“
```

V předchozích případech jsme tedy definovali proceduru, která měla dva povinné argumenty (v proceduře reprezentované formálními argumenty `x` a `y`) a dále mohla mít nepovinné argumenty, jejichž seznam byl při aplikaci navázaný na symbol `rest`. V prvním případě byly předány právě dva povinné argumenty, takže seznam nepovinných argumentů byl prázdný. V druhém případě již seznam nepovinných argumentů obsahoval jeden prvek. V třetím případě bylo předáno celkem pět argumentů, takže seznam nepovinných argumentů obsahoval poslední tři z nich.

Příklad použití nepovinných argumentů je v programu 6.6. Procedura `find` provádí podobnou činnost

Program 6.6. Test přítomnosti prvku v seznamu s navrácením příznaku.

```
(define find
  (lambda (elem l . not-found)
    (cond ((member? elem l) elem)
          ((null? not-found) #f)
          (else (car not-found)))))
```

jako procedura `member?` z programu 6.3 na straně 146 (`find` je, jak vidíme, dokonce naprogramovaná pomocí `member?`). Procedura `find` má dva povinné argumenty, prvním z nich je element, druhým je seznam. Procedura slouží k rozhodování, zda-li se daný element nachází v seznamu hodnot. V případě nalezení je ale vrácen samotný element (to se může v některých případech hodit), v případě nenalezení je vráceno standardně `#f`. Co kdybychom ale chtěli v seznamu hledat element „nepravda“, to jest samotné `#f`? Pak bychom vždy tak jako tak dostali jako výsledek aplikace `#f` (v případě nalezení i nenalezení prvku v seznamu). Problém bychom mohli napravit tak, že proceduře budeme předávat nepovinný argument, jehož hodnota bude vrácena v případě, kdy prvek nalezen nebude. Pokud nepovinný argument nebude uveden, pak při nenalezení prvku vrátíme standardní `#f`. Viz ukázky použití procedury:

```
(find 'a '(a b c d))                    ⟹  a
(find 'x '(a b c d))                    ⟹  #f
(find 'a '(a b c d) 'prvek-nenalezen)   ⟹  a
(find 'x '(a b c d) 'prvek-nenalezen)   ⟹  prvek-nenalezen
(find #f '(a b c d))                    ⟹  #f
(find #f '(a b #f d))                   ⟹  #f
(find #f '(a b c d) 'prvek-nenalezen)   ⟹  prvek-nenalezen
(find #f '(a b #f d) 'prvek-nenalezen)  ⟹  #f
```

Všimněte si, že procedura `find` pracuje de facto pouze s jedním nepovinným argumentem. Zbytek nepovinných argumentů, které by byly při její aplikaci předány v seznamu navázaném na symbol `not-found`, je procedurou ignorován.

Nyní obrátíme naši pozornost na problematiku předávání libovolného počtu argumentů. V předchozí notaci musela mít každý procedura aspoň jeden povinný argument, protože výraz `( . rest)` by nebyl syntakticky správně. Co když ale potřebujeme definovat proceduru, která může mít jakýkoliv počet argumentů. Z praxe takové procedury známe a víme o tom, že „libovolné argumenty“ jsou užitečné (vzpomeňme například jen proceduru `append`).

Uživatelsky definované procedury, které mají mít libovolný počet argumentů, vznikají vyhodnocením λ-výrazů, ve kterých je místo seznamu formálních argumentů uveden *jediný symbol*. Na tento jediný symbol bude při aplikaci navázán *seznam všech argumentů*. Viz příklady pro ilustraci:

```
((lambda args (list 'predano args)) 1 2 3 4 5 6)  ⟹  (predano (1 2 3 4 5 6))
((lambda args (list 'predano args)))              ⟹  (predano ())
((lambda args (reverse args)) 1 2 3 4 5 6)        ⟹  (6 5 4 3 2 1)
```

V prvních dvou příkladech byla aplikována procedura, která jako výsledek vrátila dvouprvkový seznam: na jeho první pozici byl symbol `predano` a na druhé pozici byl seznam všech předaných argumentů. V třetím případě jsme viděli ukázku procedury, která dané argumenty vrátí v obráceném seznamu.

V programu 6.7 je uvedena procedura `+2m` provádějící součet čtverců přes libovolné argumenty. V druhé

Program 6.7. Součet druhých mocnin.

```
(define
  +2m
  (lambda values
    (apply + (map (lambda (x) (* x x)) values))))
```

lekci jsme v programu 2.1 na straně 50 definovali proceduru na výpočet součtu dvou čtverců jako jednu z prvních procedur vůbec. V programu 6.7 se tedy nachází její zobecnění pracující s libovolným počtem argumentů. Pomocí mapování je ze seznamu čísel vytvořen seznam jejich druhých mocnin a pomocí aplikace sčítání je získána výsledná hodnota. Vše opět funguje i v mezním případě, kdy je procedura `+2m` zavolána bez argumentu. Viz následující příklady:

```
(+2m)       ⟼  0
(+2m 2)     ⟼  4
(+2m 2 3)   ⟼  13
(+2m 2 3 4) ⟼  29
```

Následující procedura provádí spojení libovolně mnoha seznamů v opačném pořadí:

```
(define
  rev-append
  (lambda lists
    (reverse (apply append lists))))
```

Proceduru můžeme použít následovně:

```
(rev-append)                          ⟼  ()
(rev-append '(a b))                   ⟼  (b a)
(rev-append '(a b) '(c d))            ⟼  (d c b a)
(rev-append '(a b) '(c d) '(1 2 3))   ⟼  (3 2 1 d c b a)
```

V předchozí lekci jsme ukázali konstruktor seznamu `list`. Nyní je ale jasné, že pokud máme k dispozici aparát pro předávání libovolného množství argumentů pomocí jejich seznamu, pak lze proceduru `list` snadno naprogramovat tak, jak je to uvedeno v programu 6.8. V tomto programu je procedura `list`

Program 6.8. Vytvoření konstruktoru seznamu.

```
(define list
  (lambda list list))
```

definována jako procedura akceptující libovolné argumenty, která vrací seznam těchto argumentů, což je přesně to, co provádí `list` představený v předchozí lekci.

Následující definice shrnuje, jak vypadá syntaxe λ-výrazů. V tomto ani v následující části učebního textu (týkající se imperativních rysů při programování) ji již nebudeme nijak rozšiřovat.

Definice 6.2 (λ-výraz s nepovinnými a libovolnými formálními argumenty). Každý seznam ve tvaru

`(lambda (`$\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_m\rangle$`)` $\langle výraz_1\rangle\ \langle výraz_2\rangle \cdots \langle výraz_k\rangle$`)`, nebo
`(lambda (`$\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle$ `.` $\langle zbytek\rangle$`)` $\langle výraz_1\rangle\ \langle výraz_2\rangle \cdots \langle výraz_k\rangle$`)`, nebo

`(lambda` ⟨*parametry*⟩ ⟨*výraz*$_1$⟩ ⟨*výraz*$_2$⟩ $\cdots$ ⟨*výraz*$_k$⟩`)`,

kde n, k jsou kladná čísla, m je nezáporné číslo, ⟨*param*$_1$⟩, ⟨*param*$_2$⟩, $\ldots$, ⟨*param*$_n$⟩, ⟨*zbytek*⟩ jsou vzájemně různé symboly, ⟨*parametry*⟩ je symbol, a ⟨*výraz*$_1$⟩, $\ldots$, ⟨*výraz*$_k$⟩ jsou libovolné výrazy tvořící *tělo*, se nazývá λ-*výraz* (*lambda výraz*). Symboly ⟨*param*$_1$⟩, $\ldots$, ⟨*param*$_n$⟩ se nazývají *formální argumenty* (někdy též *parametry*). Čísla m, n nazýváme *počet povinných formálních argumentů* (*parametrů*). Symbol ⟨*zbytek*⟩ se nazývá *formální argument* (*parametr*) *zastupující seznam nepovinných argumentů*. Symbol ⟨*parametry*⟩ se nazývá *formální argument* (*parametr*) *zastupující seznam všech argumentů*. ■

V jazyku Scheme je možné vytvářet uživatelsky definované procedury, které mají jakýkoliv počet argumentů, nebo mají některé argumenty povinné, vždy alespoň jeden, a ostatní argumenty jsou nepovinné. V obou případech jsou nepovinné argumenty předávány proceduře formou seznamu, který je navázaný na speciální formální argument.

Poznámka 6.3. Programovací jazyky mají různé způsoby, jak předat nepovinné argumenty. Jednou z oblíbených metod, kterou disponují například jazyky jako je Common LISP, PHP a další je předávání nepovinných argumentů, které jsou identifikovány svým jménem (tak zvaným *klíčem*).

6.4 Vyhodnocování elementů a prostředí jako element prvního řádu

Nyní se budeme věnovat primitivní proceduře, pomocí niž budeme schopni získat na žádost hodnotu vzniklou vyhodnocením elementu. Analogicky jako jsme v předešlých sekcích řekli, že „Apply“ je k dispozici programátorovi prostřednictvím primitivní procedury vyššího řádu `apply`, tak i „Eval“ bude uživateli k dispozici prostřednictvím primitivní procedury (vyššího řádu) `eval`. Pomocí této primitivní procedury budeme moci provádět *explicitní vyhodnocení elementů*. Veškeré doposud používané vyhodnocování bylo vždy *implicitní*.

Primitivní procedura `eval` je aplikována se dvěma argumenty z nichž druhý argument je nepovinný. Prvním (povinným) argumentem je *element*, který chceme vyhodnotit. Druhým nepovinným argumentem je *prostředí*, ve kterém chceme daný element vyhodnotit. Pokud není prostředí uvedeno, `eval` bude uvažovat vyhodnocení v globálním prostředí $\mathcal{P}_G$. Výsledkem aplikace `eval` pro dané argumenty je výsledek vyhodnocení elementu v prostředí. Z toho, co jsme teď řekli plyne, že argumenty předávané `eval` plně korespondují s argumenty pro „Eval“ tak, jak byla popsán v lekci 2, viz definici 2.7 na straně 47.

Uveďme si nyní nějaké příklady použití `eval`, zatím pouze s jedním argumentem jímž je element, který bude vyhodnocen:

```
(eval 10)          ⟼  10
(eval '+)          ⟼  „procedura sčítání čísel“
(eval '(+ 1 2))    ⟼  3
```

Předchozí tři příklady korespondují s body (A), (B) a (C) definice vyhodnocování, protože číslo se vyhodnotilo na sebe sama, symbol `+` se vyhodnotil na svou vazbu a seznam se vyhodnotil obvyklým postupem. Zde upozorněme na fakt, že `eval` je skutečně procedura, tedy před její aplikací jsou vyhodnoceny její argumenty. Proto jsme museli předat proceduře symbol `+` pomocí kvotování, stejně tak seznam `(+ 1 2)`. Kdybychom to neučinili, symbol `+` by se vyhodnotil na svou vazbu a proceduře `eval` by byla předána k vyhodnocení procedura. V tom případě by se dle bodu (D) procedura vyhodnotila na sebe sama:

```
(eval +)           ⟼  „procedura sčítání čísel“
((eval +) 1 2)     ⟼  3
```

V tomto bodu by nám asi mělo být jasné, proč jsme do definice vyhodnocování, viz definici 2.7 na straně 47, přidali bod (D). Doposud se během výpočtu vyhodnocovaly pouze elementy, které byly interními formami symbolických výrazů – těch, co jsme uvedli v programu. Pokud ale máme k dispozici evaluátor ve formě procedury `eval`, je možné mu předat libovolný element k vyhodnocení, tedy i element, který není interní formou žádného symbolického výrazu, jak je tomu například u procedur.

Pomocí `eval` je možné manipulovat s daty jako s programem. V předchozích lekcích jsme upozornili na fakt, že programy v jazyku Scheme lze chápat jako data. Interní formy seznamů jsou konstruovány pomocí párů. Pomocí `eval` tedy máme možnost vyhodnocovat datové struktury reprezentující „kusy programu“. To nám na jednu stranu dává obrovský potenciál, protože můžeme třeba *uživatelský vstup* transformovat na kód a spustit jej, což usnadňuje řadu operací. Na druhou stranu je použití `eval` krajně nebezpečné a mělo by být vždy odůvodněné.

V následujícím příkladu ukazujeme konstrukci dvou seznamů (data), která jsou použita „jako program“:

```
(eval (cons '+ (cons 1 (cons 2 '()))))  ⟹  3
(eval (cons + (cons 1 (cons 2 '()))))   ⟹  3
(cons + (cons 1 (cons 2 '())))          ⟹  („procedura sčítání čísel“ 1 2)
```

Všimněte si, že na druhém řádku byl zkonstruován seznam `(„procedura sčítání čísel“ 1 2)`, který začíná procedurou a dalšími prvky jsou dvě čísla. Oproti prvnímu řádku tedy nestojí na prvním místě seznamu symbol, ale přímo procedura. Této situace bychom nemohli dosáhnout, kdybychom nepoužívali `eval` explicitně.

V následující ukázce jsme vyhodnocením vytvořili proceduru a dále ji aplikovali. Jelikož `eval` s jedním argumentem vyhodnocuje elementy v globálním prostředí, bude prostředí vzniku této procedury právě globální prostředí:

```
(eval '(lambda (x) 10))                  ⟹  „konstantní procedura vracející 10“
((eval '(lambda (x) 10)) 20)             ⟹  10
(apply (eval '(lambda (x) 10)) 20 '())   ⟹  10
```

Vyhodnocení následujícího výrazu končí chybou

```
(let ((x 10))
  (eval '(+ x 1)))  ⟹  „CHYBA: Symbol x nemá vazbu.“,
```

protože seznam `(+ x 1)` byl vyhodnocen v globálním prostředí (ve kterém `x` nemá vazbu) a to i navzdory tomu, že `eval` jsme uvedli v `let`-bloku, kde měl symbol `x` vazbu.

Jako další příklad si uveďme následující proceduru vyššího řádu:

```
(define proc
  (lambda (c)
    (eval '(lambda (x) (+ x c)))))
```

Tato procedura při své aplikaci vrací novou proceduru, která byla ale vytvořena v globálním prostředí. To jest při aplikaci `proc` je sice předán argument navázaný na `c`, jeho vazba ale není viditelná z prostředí vzniku vrácené procedury. Kdybychom tuto vrácenou proceduru aplikovali, vazba symbolu `c` by byla hledána v globálním prostředí, viz ukázku:

```
((proc 10) 20)  ⟹  „CHYBA: Symbol c nemá vazbu.“
(define c 100)
((proc 10) 20)  ⟹  120
```

Kdybychom někdy potřebovali vyrábět proceduru vyhodnocením seznamu (třeba protože část procedury by byla dodána až během činnosti programu pomocí interakce s uživatelem), pak bychom mohli problém s vazbou volných symbolů vyřešit tak, že místo symbolů bychom do seznamu rovnou dosadili výsledky jejich vyhodnocení – jejich vazby v aktuálním prostředí, ve kterém `eval` aplikujeme. Viz následující ukázku:

```
(define proc
  (lambda (c)
    (eval (list 'lambda
                (list 'x)
                (list '+ 'x c)))))
```

V tomto případě procedura `proc` vrací proceduru, která vznikne vyhodnocením seznamu v globálním prostředí. V tomto případě jsem ale do seznamu místo symbolu `c` vložili hodnotu jeho vazby v lokálním prostředí procedury `proc`. Vytvořili jsme tak vlastně seznam ve tvaru

`(lambda (x) (+ x „hodnota c“))`

a ten byl vyhodnocen v globálním prostředí. Vzniklou proceduru již tedy můžeme používat bez nutnosti provádět globální definici a procedura má kýžený efekt:

`((proc 10) 20)` $\longmapsto$ `30`

> Z pohledu jazyka Scheme jsou *data* totéž co *program*. Program lze chápat jako data a data mohou být použita pomocí `eval` jako program. Při používání `eval` je však potřeba dbát velké obezřetnosti, protože jeho (nadměrné) používání Často znesnadňuje ladění programu. Chyby mohou vznikat za běhu programu, aniž by byly v programu „vidět“ na nějakém jeho konkrétním místě.

Našim dalším cílem bude naimplementovat procedury `forall` a `exists` reprezentující kvantifikátory $\forall$ (všeobecný kvantifikátor „pro každý . . . “) a $\exists$ (existenční kvantifikátor „existuje . . . “). Budou jako argument brát predikát o jednom argumentu ⟨*predikát*⟩ a seznam ⟨*seznam*⟩ a vracet pravdivostní hodnotu. V případě `forall` to bude pravda `#t`, pokud každý prvek seznamu ⟨*seznam*⟩ splňuje predikát ⟨*predikát*⟩ a jinak `#f`. Procedura pro existenční kvantifikátor bude vracet `#t`, pokud alespoň jeden prvek ze seznamu ⟨*seznam*⟩ splňuje predikát ⟨*predikát*⟩. V opačném případě bude vracet `#f`. Jelikož obě procedury si budou podobné, omezíme se v následujícím na rozbor procedury `forall` modeující všeobecný kvantifikátor.

Pomocí `map` a daného predikátu dostaneme z původního seznamu seznam pravdivostních hodnot určujících, zda-li prvek na dané pozici splňuje podmínku danou predikátem:

`(map` ⟨*predikát*⟩ ⟨*seznam*⟩`)`.

Nyní potřebujeme zjistit, jaké pravdivostní hodnoty seznam obsahuje. V případě, že by `and` byla procedura, mohli bychom toho dosáhnout pomocí procedury `apply`. Ale jelikož jde o speciální formu, obdržíme při případném pokusu o aplikaci chybu:

`(apply and '(#t #t #f))` $\longmapsto$ „CHYBA: Nesprávné použití speciální formy `and`“ .

Potřebujeme tedy mít „and“ a „or“ jako procedury libovolně mnoha argumentů. Budeme se teď zabývat tímto problémem. Proceduru pro „and“ – `and-proc` bychom mohli implementovat například takto:

```
(define and-proc
  (lambda args
    (null? (remove (lambda (x) x) args))))
```

Nejdříve jsme použitím procedury `remove` ze seznamu argumentů navázaného na symbol `args` odstranili všechny prvky různé od `#f`, použitím procedury `remove` s procedurou identity. Poté jsme otestovali, zda je výsledný seznam prázdný. Pokud ano, znamená to, že žádný argument procedury `and-proc` nebyl nepravda `#f`, a tedy výsledkem aplikace procedury `and-proc` bude pravda. V opačném případě bude výsledkem `and-proc` nepravda.

Procedura `and-proc` je tedy implementací operace logické konjunkce:

```
(and-proc)                      ⟼ #t
(and-proc 1 2 3)                ⟼ #t  rozdíl oproti and: (and 1 2 3) ⟼ 3
(and-proc #t (< 1 2) #t)        ⟼ #f
```

Na `and-proc` je navázána procedura, nikoli speciální forma. Důsledkem toho je, jakým způsobem se vyhodnocují její argumenty:

`(and-proc #f nenavazany-symbol #f)` $\longmapsto$ „CHYBA: Symbol `nenavazany-symbol` nemá vazbu“

Pro nás je důležitější ta skutečnost, že `and-proc` jako procedura může být použita třeba jako argument procedury `apply`:

`(apply and-proc '(#t #t #t))` $\longmapsto$ `#t`

K implementaci `and-proc` bychom také mohli chytře použít speciální formu `and` a proceduru `eval`. Máme-li seznam argumentů, můžeme na jeho začátek přidat symbol `and` a výsledný seznam explicitně vyhodnotit použitím `eval`.

```
(define and-proc
  (lambda args
    (eval (cons 'and args))))
```

Analogicky můžeme vytvořit proceduru, která bude obdobou speciální formy `or`:

```
(define or-proc
  (lambda args
    (eval (cons 'or args))))

(and-proc 1 2 3 #f 10)        ⟹ #f
(and-proc (+ 1 2))            ⟹ 3
(and-proc  (if #f #t #t))     ⟹ #t
```

Poznámka 6.4. Tato implementace ale není úplně korektní a bude pracovat správně jen do té doby, dokud výsledky vyhodnocení argumentů budou pravdivostní hodnoty, čísla nebo jiné elementy, které se vyhodnocují samy na sebe. Podívejme se na to na následujícím příkladě, kde dostáváme opačné hodnoty pro stejný argument.

```
(and-proc '(if #f #t #f))  ⟹  #f
(and '(if #f #t #f))       ⟹  (if #f #t #f) tedy hodnota považovaná za pravdu.
```

Speciální forma `and` nám vrací čtyřprvkový seznam `(if #f #t #f)`. Ten vznikne vyhodnocením výrazu `'(if #f #t #f)` a je vrácen jako výsledek aplikace této formy, protože se jedná o poslední argument. Při aplikaci procedury `and-proc` dochází k nepříjemnému efektu, který není na první pohled zřejmý. Protože se jedná o proceduru, jsou její argumenty implicitně vyhodnoceny. Do seznamu jejich vyhodnocení je přidán symbol `and` a pak je vyhodnocen použitím speciální formy `eval`. Při aplikaci formy `and`, jak je popsána v definici 2.22 na straně 64, se postupně *vyhodnocují* argumenty. Argumenty jsou tak vlastně vyhodnoceny dvakrát. Vyhodnocením seznamu `(if #f #t #f)` dostaneme `#f`. Odtud výsledná hodnota. S procedurou `or-proc` to samozřejmě bude podobné.

Pomocí procedur `and-proc` a `or-proc` můžeme konečně naprogramovat proceduru univerzálního kvantifikátoru `forall` a proceduru existenčního kvantifikátoru `exists`.

```
(define forall
  (lambda (f l)
    (apply and-proc (map f l))))

(define exists
  (lambda (f l)
    (apply or-proc (map f l))))
```

Procedura univerzálního kvantifikátoru `forall` tedy vrací pravdu, pokud predikát platí pro všechny prvky v seznamu:

```
(forall even? '(1 2 3 4 5))  ⟹  #f
(forall even? '(2 4))        ⟹  #t
(forall even? '(1 3 5))      ⟹  #f
(forall even? '())           ⟹  #t
```

Všimněte si posledního případu: každý prvek prázdného seznamu splňuje jakoukoliv vlastnost triviálně (souhlasí s vlastnostmi všeobecného kvantifikátoru). Analogicky procedura existenčního kvantifikátoru `exists` vrací pravdu, pokud predikát platí alespoň pro jeden prvek v seznamu:

```
(exists even? '(1 2 3 4 5))  ⟹  #t
(exists even? '(2 4))        ⟹  #t
(exists even? '(1 3 5))      ⟹  #f
(exists even? '())           ⟹  #f
```

Naše kvantifikátory můžeme rozšířit na procedury více argumentů. Podobně jako u procedury `map` pak vstupní predikát musí přijímat tolik argumentů, kolik je vstupních seznamů. Predikát je pak aplikován na prvky na stejných pozicích.

```
(define forall
  (lambda (f . lists)
    (apply and-proc (apply map f lists))))

(define exists
  (lambda (f . lists)
    (apply or-proc (apply map f lists))))
```

Všeobecný kvantifikátor `forall` pak zjišťuje, zda všechny prvky na stejných pozicích v seznamech splňují tento predikát.

```
(forall (lambda (x y) (<= x y)) '(10 20 30) '(11 22 33))  ⟹  #t
(forall (lambda (x y) (<= x y)) '(10 23 30) '(11 22 33))  ⟹  #f
```

V předchozí ukázce bychom přísně vzato nemuseli používat λ-výrazy, stačilo by pouze uvést:

```
(forall <= '(10 20 30) '(11 22 33))  ⟹  #t
(forall <= '(10 23 30) '(11 22 33))  ⟹  #f
```

A podobně existenční kvantifikátor `exists` pro více seznamů vrací pravdu `#t`, jestliže prvky na stejných pozicích splňují daný predikát.

```
(exists (lambda (x y) (> x y)) '(10 20 30) '(11 22 33)) ⟹ #f
(exists (lambda (x y) (> x y)) '(10 23 30) '(11 22 33)) ⟹ #t
```

Teď se budeme zabývat procedurou `eval` se dvěma argumenty. Jak již bylo řečeno, prvním argumentem je element k vyhodnocení, druhým argumentem je *prostředí*, ve kterém má k vyhodnocení dojít. Zde se vlastně dostáváme do zajímavého bodu, protože pokud chceme, abychom pomocí `eval` vyhodnocovali elementy relativně vzhledem k prostředí, pak musí být prostředí v jazyku Scheme *elementem prvního řádu.* Vskutku, prostředí jsou de facto tabulky obsahující symboly a jejich vazby plus ukazatele na svého předchůdce. Nic nám tedy nebrání abychom tyto „tabulky“ chápali jako elementy jazyka Scheme. *Prostředí* je pro nás tedy *nový element jazyka.*

Proto, abychom mohli pracovat s prostředím jako s elementem, potřebujeme mít k dispozici nějaké primitivní procedury nebo speciální formy, které budou nějaká prostředí vracet. Nejprve budeme uvažovat speciální formu `the-environment`:

Definice 6.5 (speciální forma `the-environment`). Speciální forma `the-environment` se používá bez argumentu. Výsledkem její aplikace je prostředí, ve kterém byla aplikována (aktuální prostředí). ■

Před tím, než ukážeme příklad si uvědomme, proč je `the-environment` speciální forma a nikoliv procedura. Při aplikaci procedur nehraje roli prostředí, ve kterém byly procedury aplikovány, protože používáme lexikální rozsah platnosti. Procedury ani nemají možnost zjistit, v jakém prostředí byly aplikovány. Naproti tomu speciální formy řídí vyhodnocování svých argumentů, musí mít tedy prostředí své aplikace k dispozici. Speciální forma `the-environment` prostě udělá jen to, že toto prostředí vrátí jako výsledek. Viz příklady použití:

```
(the-environment)          ⟹  „globální prostředí“
((lambda (x)
   (the-environment))
 10)                       ⟹  „lokální prostředí procedury, kde x ↦ 10“
(let ((x 10))
  (the-environment))       ⟹  „lokální prostředí procedury, kde x ↦ 10“
```

V druhém případě si všimněte, že procedura vzniklá vyhodnocením λ-výrazu ve svém těle provede pouze aplikaci `the-environment`. Při aplikaci této procedury je vytvořeno lokální prostředí, v němž je na `x`

navázána hodnota 10 a toto prostředí je vráceno. V třetím případě se jedná o stejný případ, protože let--výraz je v podstatě jen zkrácením výrazu na druhém řádku.

Nyní již můžeme provést vyhodnocení výrazu v aktuálním prostředí:

```
(let ((x 10))
  (eval '(+ x 1) (the-environment)))  ⟹  11
```

Kdybychom v předchozím příkladu u eval neuvedli druhý argument, pak by byl výraz vyhodnocen v globálním prostředí a nastala by chyba. My jsme ale výraz vyhodnotili v aktuálním prostředí (v tele let-bloku), to jest v prostředí, kde je na x navázána hodnota.

Dále budeme v jazyku uvažovat proceduru environment-parent, která *pro dané prostředí vrátí jeho předka.* V případě, že je environment-parent aplikována na globální prostředí, které předka nemá, pak vrátí #f. Například použitím

```
(let* ((x 10)
       (y 20))
  (environment-parent (the-environment)))
```

bychom získali prostředí, ve kterém má vazbu symbol x, ale ve kterém nemá vazbu symbol y. Musíme si uvědomit, že speciální forma let* vytváří s každým symbolem nové prostředí a tato prostředí jsou do sebe zanořená, viz třetí lekci.

Procedura procedure-environment pro danou uživatelsky definovanou proceduru *vrátí prostředí jejího vzniku.* Například pomocí

```
(procedure-environment
 (let ((x 10))
   (lambda (y)
     (+ x y))))
```

získáme prostředí vzniku procedury vzniklé vyhodnocením uvedeného λ-výrazu. To je prostředí vytvořené pomocí let, ve kterém je na symbol x navázána hodnota 10. Máme-li k dispozici procedure-environment, pak bychom již nutně nemuseli mít the-environment, protože kdekoliv v programu pomocí

```
(procedure-environment (lambda () #f))
```

můžeme získat aktuální prostředí. Samozřejmě, že toto řešení je méně efektivní, protože při něm vždy vytvoříme novou proceduru jen proto, abychom posléze získali prostředí jejího vzniku.

Poslední pomocnou procedurou, kterou představíme, je procedura environment->list, která pro dané prostředí vrací seznam tečkových párů ve tvaru (⟨*symbol*⟩ . ⟨*vazba*⟩) obsahující všechny vazby v daném prostředí. Pomocí této procedury tedy budeme schopni „srozumitelně vypisovat“ obsah prostředí. Samotná prostředí nemají žádnou čitelnou externí reprezentaci. Viz příklad použití právě popsané procedury:

```
(environment->list
 (procedure-environment
  (let ((x 10)
        (z 20))
    (lambda (y)
      (+ x y)))))        ⟹  ((x . 10) (z . 20))

(environment->list
 ((lambda (x y)
    (the-environment))
  100 200))              ⟹  ((x . 100) (y . 100))
```

Použitím předchozích procedur spolu s env můžeme provádět vyhodnocování elementů v aktuálním prostředí i v prostředích vzniku daných procedur. Například následující ukázka demonstruje vyhodnocení seznamu v prostředí vzniku nějaké procedury.

```
(eval '(* x x)
      (procedure-environment
       (let ((x 10))
         (lambda (y) (+ x y)))))  ⟼  100
```

Procedura vzniklá vyhodnocením `(lambda (y) (+ x y))` není v předchozím příkladu vůbec aplikována.

> Prostředí je v našem pojetí elementem prvního řádu. Na prostředí se taky můžeme dívat jako na speciální hierarchická data. Konstruktorem prostředí je aplikace uživatelsky definovaných procedur, protože při ní prostředí vznikají. Selektory prostředí jsou reprezentovány procedurami jako je `environment-parent` a podobně.

Pokud jsme o používání `eval` řekli, že je nebezpečné, pak bychom měli o používání `eval` s druhým argumentem (prostředím) říct, že je ještě mnohem víc nebezpečné a dát za to jeden velký tlustý vykřičník. Pomocí `eval` totiž můžeme vyhodnocovat elementy při jejichž vyhodnocení dojde k *vedlejšímu efektu*, například ke změně vazby nebo překrytí vazby v nějakém prostředí. Od toho okamžiku mohou začít některé procedury vykazovat „zvláštní chování“. Demonstrujme si vše na následujícím větším příkladu. Předpokládejme, že máme definovánu proceduru `aux-proc` následovně:

```
(define aux-proc
  (let ()
    (lambda (x)
      (+ x y))))
```

Procedura vznikla v prostředí, ve kterém nejsou žádné vazby, které vzniklo použitím `let` bez seznamu vazeb. Předkem tohoto prostředí je globální prostředí. Vyhodnocení následujícího výrazu pochopitelně končí chybou:

```
(aux-proc 10)  ⟼  „CHYBA: Symbol y nemá vazbu.“
```

Vyhodnocením následujícího výrazu:

```
(eval '(define y 20)
      (procedure-environment aux-proc))
```

došlo ke vzniku vedlejšího efektu, jímž byla definice nové vazby symbolu `y`. Výraz způsobující tuto definici jsme ale nevyhodnotili v globálním prostředí, nýbrž v prostředí vzniku procedury `aux-proc`. Takže v globálním prostředí symbol `y` zůstává i nadále bez vazby, ale aplikace `aux-proc` již proběhne bez chyby:

```
y              ⟼  „Symbol y nemá vazbu“
(aux-proc 10)  ⟼  30
```

To co jsme teď provedli byl z programátorského hlediska „extrémně nečistý trik“ (slangově *hack*), kdy jsme lokálnímu prostředí procedury, které by za normálních podmínek nebylo z globálního prostředí nijak dosažitelné, „vnutili“ novou vazbu symbolu. Přitom tato vazba nadále z venčí není na první pohled vidět, globální prostředí je nezměněno. Podíváme-li se nyní na definici procedury `aux-proc`, nikde tam symbol `y` pochopitelně nevidíme. Externí pozorovatel, který by nevěděl o naší „černé magii“, by si myslel, že aplikace `aux-proc` bude končit chybou, stejně tak, jak to bylo v původním případě.

Při dalším předefinování `y` v prostředí vzniku procedury, by se `aux-proc` opět začala chovat jinak:

```
(eval '(define y 200)
      (procedure-environment aux-proc))
(aux-proc 10)  ⟼  210
```

> Pokud to není nezbytně nutné, procedura `eval` by neměla být vůbec používána. Na druhou stranu, je-li její použití na místě a pokud může výrazně urychlit vývoj programu, pak jejímu použití nelze snad nic namítat. Proceduru bychom ale měli používat pokud možno tak, abychom vyhodnocováním výrazů co možná nejméně ovlivňovali činnost zbytku programu.

Podotkněme, že procedura `eval` je popsána v definici R^{5}RS jazyka Scheme, viz [R5RS]. V tomto reportu je popsána i verze `eval` se dvěma argumenty, ale pouze ve velmi omezené míře. Výše uvedená speciální forma `the-environment` a procedury `procedure-environment`, `environment-parent` a `environment->list` nejsou v R^{5}RS vůbec zahrnuty. Některé široce používané interprety jazyka Scheme ale podobnými procedurami skutečně disponují, například interpret Elk. V poslední lekci v této části učebního textu do paradigmat programování si ukážeme implementaci skutečného interpretu jazyka Scheme, ve kterém budeme mít všechny tyto speciální formy a procedury k dispozici.

Poznámka 6.6. Ve funkcionálních programovacích jazycích je procedura `eval`, nebo nějaký její ekvivalent, obvykle k dispozici. Totéž se nedá říct o jiných programovacích jazycích. V procedurálních jazycích se procedury provádějící evaluaci vyskytují jen minimálně. V těchto vyšších programovacích jazycích také v podstatě neplatí, že program a data jsou totéž (teoreticky to možná platí, ale prakticky nikoliv). Možnosti vyhodnocování jsou někdy mylně přičítány jen interpretům programovacích jazyků. Protipříkladem mohou být jazyk Common LISP, který je kompilovaný (i interpretovaný) a ve kterém je `eval` přítomen (prostředí zde ale není element prvního řádu).

Na závěr této sekce uvedeme několik příkladů použití procedury `apply` a speciální formy `eval` a odvozených procedur. Prvním z nich je přibližná rovnost vektorů. Vektory jsou representovány číselnými seznamy `v1` a `v2`. Argument `delta` je pak tolerance, ve které se může pohybovat rozdíl jednotlivých složek, aby ještě vektory byly uznány za rovné. To pomáhá předejít problémům se zaokrouhlovacími chybami, které vznikají při manipulaci s neexaktními čísly. Například na některých platformách může nastat situace, kde

```
(= (- 1.2 1) 0.2)  ⟼  #f.
```

Proceduru `vec-equal?` bychom mohli naimplementovat například takto:

```
(define vec-equal?
  (lambda (v1 v2 delta)
    (forall (lambda (x y)
              (< (abs (- x y)) delta))
            v1 v2)))
```

Teď uvádíme aplikace:

```
(vec-equal? '(0 1 3) '(0 1.2 3) 0.1)      ⟼  #f
(vec-equal? '(0 1 3) '(0 1.2 3.001) 0.3)  ⟼  #t
```

Asociativní pole je datová struktura, která se skládá z kolekce tak zvaných klíčů a kolekce hodnot. Každý klíč je přiřazen jedné hodnotě. Základní operace na této datové struktuře jsou: přiřazení hodnoty nějakému klíči (respektive změna stávajícího přiřazení), vyhledání hodnoty podle klíče a zrušení stávajícího přiřazení. My budeme asociativní pole reprezentovat seznamem přiřazení. Přiřazením přitom budeme rozumět pár, jehož první prvek je klíč a druhý prvek je přiřazená hodnota. Budeme implementovat první dvě zmíněné základní operace. Teď tedy přidání prvku do asociativního pole:

```
(define cons-assoc
  (lambda (key val assoc)
    (let ((cell (cons key val)))
      (if (exists (lambda (x) (equal? (car x) key))
                  assoc)
          (map (lambda (x)
                 (if (equal? (car x) key)
                     cell
                     x))
               assoc)
          (cons cell assoc)))))
```

Nejdříve jsme zjistili použitím procedury kvantifikátoru `exists`, jestli je k zadanému klíči `key` přiřazena hodnota. Jestliže ne, přidáme pár reprezentující přiřazení hodnoty ke klíči do asociativního pole pomocí

procedury `cons`. Pokud ale přiřazení k takovému klíči již v poli je, použijeme proceduru `map`, abychom nahradili původní přiřazení novým.

Druhou základní operací je selekce. Použitím procedury `filter` vybereme ty prvky seznamu reprezentující asociativní pole, které mají shodný klíč s požadovaným klíčem. Pokud výsledný je výsledný seznam prázdný, nebylo přiřazení nalezeno. V opačném případě vracíme hodnotu z tohoto přiřazení.

```
(define assoc-key
  (lambda (assoc key)
    (let ((found (filter (lambda (x)
                           (equal? (car x) key))
                         assoc)))
      (if (null? found)
          #f
          (cdar found)))))
```

Další příklad se týká opět datových tabulek. Je vlastně vylepšení reprezentace datových tabulek, kterou jsme zavedli v sekci 5.8, tak, aby každá tabulka měla „hlavičku“, ve které jsou jména atributů, které budeme používat při selekci a projekci. Toto je příklad takové tabulky:

```
(define mesta
  '((jmeno p-obyvatel p-nadrazi velikost)
    (Olomouc 120 3 stredni)
    (Prostejov 45 2 male)
    (Prerov 50 3 male)
    (Praha 1200 8 velke)))
```

Procedura selekce bude brát jako argumenty tabulku a výraz reprezentující podmínku. V tomto výrazu používáme jména z hlavičky tabulku. Příklad použití může být třeba tento:

```
(selection mesta
           '(and (>= p-obyvatel 50)
                 (not (equal? velikost 'male))))
```

Nejdůležitější rys použitý v této procedure je vytvoření λ-výrazu vyhodnocením výrazu

```
(list 'lambda head condition)
```

a jeho následné vyhodnocení procedurou `eval`. Tato procedura a tělo tabulky – to jest tabulka bez hlavičky – pak budou vstupními argumenty pro proceduru `filter`, která pak provede vyfiltrování požadovaných řádků.

```
(define selection
  (lambda (table condition)
    (let* ((head (car table))
           (body (cdr table))
           (property (eval (list 'lambda head condition))))
      (filter (lambda (x)
                (apply property x))
              body))))
```

Toto naše řešení má ale mouchu. A to právě v použití procedury `eval`. Jde o to, že tato procedura, je-li použita s jedním argumentem, vyhodnocuje tento svůj argument v globálním prostředí. A z toho důvodu nebude fungovat následující kód, kde se ve výrazu, který reprezentuje podmínku odvoláváme na lokálně vázaný symbol.

```
(let ((x 50))
  (selection mesta
             '(and (>= p-obyvatel x)
                   (not (equal? velikost 'male))))) ⟹ „CHYBA: Symbol x nemá vazbu.“
```

Tento problém bychom samozřejmě mohli vyřešit použitím procedury `eval` se dvěma argumenty. Druhým argumentem bude aktuální prostředí, to musíme předat jako další argument při spuštění selekce:

```
(define selection
  (lambda (table env condition)
    (let* ((head (car table))
           (body (cdr table))
           (property (eval (list 'lambda head condition) env)))
      (filter (lambda (x)
                (apply property x))
              body))))
```

Viz příklad použití:

```
(let ((x 50))
  (selection mesta
             (the-environment)
             '(and (>= p-obyvatel x)
                   (not (equal? velikost 'male)))))
```

V jednom z dalších dílů textu si ukážeme mnohem elegantnější řešení tohoto problému.

6.5 Reprezentace množin a relací pomocí seznamů

V této sekci ukážeme reprezentaci konečných množin a relací pomocí seznamů, a procedur pro manipulaci s nimi. Na nich demonstrujeme použití procedur `apply`, `eval` a procedur implementovaných v této sekci. Také na nich ukážeme filtraci a vytváření procedur s nepovinnými argumenty a s libovolným počtem argumentů.

Za množinu budeme považovat seznam bez duplicit. V tomto seznamu pro nás bude důležitý pouze výčet prvků, nikoli jejich pořadí. Prázdná množina je reprezentovaná prázdným seznamem:

```
(define the-empty-set '())
```

Počet prvků množiny (kardinalita) se bude shodovat s délkou seznamu:

```
(define card
  (lambda (set)
    (length set)))
```

Procedura `make-set` vytváří množinu výběrem těch prvků z množiny ⟨*universe*⟩, které mají vlastnost ⟨*prop?*⟩. Jedna se o jednoduché použití procedury `filter`:

```
(define make-set
  (lambda (prop? universe)
    (filter prop? universe)))
```

Přidání prvku do množiny. Potřebujeme nejdříve zkontrolovat, jestli přidávaný prvek už v množině není. V případě, že ne, přidáme prvek. Jinak je výsledkem původní množina. Tak se zabrání možnému vzniku duplicit.

```
(define cons-set
  (lambda (x set)
    (if (in? x set)
        set
        (cons x set))))
```

V proceduře tedy potřebujeme predikát `in?`, který by rozhodoval, zda je daný element prvkem v seznamu. To zjistíme tak, že vyfiltrujeme prvky, které jsou rovny (při porovnání pomocí `equal?`), a otestujeme, jestli je výsledný seznam neprázdný.

```
(define in?
  (lambda (x set)
    (not (null? (filter (lambda (y) (equal? x y)) set)))))
```

Též bychom mohli použít proceduru `exists` a s její pomocí zjišťovat, zda je alespoň jeden prvek množiny roven danému elementu. Kód by pak vypadal následně:

```
(define in?
  (lambda (x set)
    (exists (lambda (p) (equal? x p)) set)))
```

Další procedura `set?` bude testovat, jestli je její argument množina. Tedy ověří, že se jedná o seznam a pak pro každý prvek tohoto seznamu zjistíme, jestli je počet jeho výskytů roven jedné. Ke zjištění počtu výskytů je využita procedura `occurrences`, kterou napíšeme vzápětí.

```
(define set?
  (lambda (elem)
    (and (list? elem)
         (forall (lambda (x)
                   (= (occurrences x elem) 1))
                 elem))))
```

Teď tedy k pomocné proceduře `occurrences`, která se používá se dvěma argumenty – elementem ⟨*elem*⟩ a seznamem ⟨*l*⟩. Tato procedura vrací počet výskytů prvku ⟨*elem*⟩ v seznamu ⟨*l*⟩. Můžeme ji realizovat například tak, že vyfiltrujeme ze seznamu ⟨*l*⟩ všechny prvky, které jsou shodné s elementem ⟨*elem*⟩. Kód takové implementace by vypadal takto:

```
(define occurrences
  (lambda (elem l)
    (length (filter (lambda (x) (equal? x elem)) l))))
```

Jinou možností je vytvořit aplikací procedurou `map` zaměnit prvky shodné s elementem ⟨*elem*⟩ za jedničky a ostatní za nuly. Na takto vzniklý seznam pak aplikujeme proceduru sčítání čísel. Než se začneme zabývat množinovými operacemi, vytvoříme si dvě pomocné procedury `map-index` a `map-tail`. Jedná se o obdoby procedury `map` (pro jeden seznam). Procedura `map-index` předává vstupní proceduře nejen prvky zadaného seznamu, ale také jejich indexy. Mapovaná procedura tedy bude přijímat dva argumenty – prvním z nich bude prvek z původního seznamu, druhým index tohoto prvku.

```
(define map-index
  (lambda (f l)
    (let ((indices (build-list (length l) (lambda (i) i))))
      (map f l indices))))
```

Procedura `map-index` je naprogramována tak, že vytvoří seznam indexů pomocí `build-list`, podobně jako v programu 6.4, to jest jako v implementaci procedury `list-ref`. Poté namapujeme předanou proceduru na původní seznam a seznam indexů. Viz příklady použití:

```
(map-index cons '(a b c d))              ⟹  ((a . 0) (b . 1) (c . 2) (d . 3))
(map-index (lambda (x y) x) '(a b c d))  ⟹  (a b c d)
(map-index (lambda (x y) y) '(a b c d))  ⟹  (0 1 2 3)
```

Mapovací procedura `map-tail` bude předávat mapované proceduře místo jednotlivých prvků podseznamy. Místo prvního prvku, bude předán celý seznam, namísto druhého seznam bez prvního prvku, a tak dále. Až konečně namísto posledního prvku seznamu bude předán jednoprvkový seznam obsahující poslední prvek.

Tyto podseznamy budeme získávat pomocí procedury `list-tail`. Ta se používá se dvěma argumenty – – prvním je seznam, a druhým číslo ⟨*i*⟩. Výsledkem aplikace je pak seznam bez prvních ⟨*i*⟩ prvků. Viz příklady požití:

```
(list-tail '(1 2 3 4 5) 1)  ⟹  (2 3 4 5)
(list-tail '(1 2 3 4 5) 3)  ⟹  (4 5)
(list-tail '(1 2 3 4 5) 5)  ⟹  ()
(list-tail '(1 2 3 4 5) 7)  ⟹  „CHYBA: Seznam má příliš malý počet prvků.“
```

Pomocí této procedury a `map-index` je implementace procedury `map-tail` velmi přímočará:

```
(define map-tail
  (lambda (f l)
    (map-index (lambda (x i)
                 (f (list-tail l i)))
               l)))
```

Viz příklad použití procedury:

```
(map-tail (lambda (x) x) '(a b c d))  ⟹  ((a b c d) (b c d) (c d) (d))
```

Nyní uděláme proceduru `list->set`, která bude konvertovat seznam na množinu. Tento seznam bude jejím jediným argumentem a procedura z něj odstraní duplicitní prvky.

```
(define list->set
  (lambda (l)
    (apply append
           (map-tail (lambda (x)
                       (let ((head (car x))
                             (tail (cdr x)))
                         (if (in? head tail)
                             '()
                             (list head))))
                     l))))
```

Nyní se zaměříme na operace s množinami – na sjednocení množin (`union`), průnik množin (`intersect`) a množinový rozdíl. Implementace procedury pro sjednocení množin je velice jednoduchá. Množiny, tedy seznamy, spojíme aplikací procedury `append`. V takto vzniklém seznamu se ale mohou vyskytovat duplicitní prvky – ty odstraníme použitím `list->set`.

```
(define union
  (lambda (set-A set-B)
    (list->set (append set-A set-B))))
```

Průnik množin `intersect` bude pro nás aplikací `make-set`. Universem bude sjednocení množin vytvořené pomocí procedury `union`, požadovanou vlastností pak bude přítomnost prvku (zjištěná predikátem `in?`) v obou množinách:

```
(define intersect
  (lambda (set-A set-B)
    (make-set (lambda (x)
                (and (in? x set-A)
                     (in? x set-B)))
              (union set-A set-B))))
```

Tyto dvě uvedené množinové operace můžeme sjednotit do procedury vyššího řádu `set-operation`:

```
(define set-operation
  (lambda (prop)
    (lambda (set-A set-B)
      (filter (lambda (x) (prop x set-A set-B))
              (list->set (append set-A set-B))))))
```

Pomocí `set-operation` můžeme definovat operace sjednocení a průniku množin.

```
(define union (set-operation (lambda (x A B) (or (in? x A) (in? x B)))))
(define intersect (set-operation (lambda (x A B) (and (in? x A) (in? x B)))))
```

Nebo třeba operaci množinového rozdílu:

```
(define minus (set-operation (lambda (x A B) (and (in? x A) (not (in? x B))))))
```

```
(union '(10 20 30) '(20 30 40))      ⟼  (10 20 30 40)
(intersect '(10 20 30) '(20 30 40))  ⟼  (20 30)
(minus '(10 20 30) '(20 30 40))      ⟼  (10)
```

Dále předchozí reprezentace množin použijeme k reprezentaci binárních relací na množinách. Připomeňme, že binární relace na množině je podmnožinou druhé kartézské mocniny této množiny. Kartézská mocnina množiny A je množina uspořádaných dvojic $\langle a, b\rangle$ takových, že první prvek a i druhý prvek b patří do množiny A. My budeme uspořádanou dvojici reprezentovat tečkovým párem (což se přímo nabízí). Vytvoříme si proto separátní konstruktory a selektory pro uspořádanou dvojici:

```
(define make-tuple cons)
(define 1st car)
(define 2nd cdr)
```

Procedura `cartesian-square` bude pro množinu vracet její druhou kartézskou mocninu.

```
(define cartesian-square
  (lambda (set)
    (apply append
           (map (lambda (x)
                  (map (lambda (y)
                         (make-tuple x y))
                       set))
                set))))
```

Výraz `(map (lambda (y) (make-tuple x y)) set)` vytvoří seznam všech dvojic, jejichž prvním prvkem je hodnota navázaná na symbol `x` a durhým prvkem je prvek z množiny `set`. Vnějším použitím procedury `map`

```
(map (lambda (x)
  (map (lambda (y)
         (make-tuple x y))
       set))
     set)
```

dostáváme seznam obsahující pro každé x z množiny `set` seznam všech párů kódujících uspořádané dvojice $\langle x, y\rangle$, kde y patří do množiny `set`. A tyto seznamy spojíme použitím procedury `append`.

Nyní můžeme přikročit k definici konstruktoru relace. Procedura `make-relation` bude procedurou dvou argumentů; prvním z nich bude vlastnost reprezentovaná predikátem dvou argumentů a druhým množina nad kterou bude relace definovaná.

```
(define make-relation
  (lambda (prop? universe)
    (filter (lambda (x)
              (prop? (1st x) (2st x)))
            (cartesian-square universe))))
```

Vytvořili jsme tedy druhou kartézskou mocniny množiny `universe` a z ní jsme aplikaci procedury `filter` vybrali ty dvojice, které splňují vlastnost `prop?`. Následují příklady použití tohoto konstruktoru.

```
(define u '(0 1 2 3 4 5))
(make-relation (lambda (x y) #f) u)
  ⟼  ()
```

```
(make-relation (lambda (x y) (= x y)) u)
  ⟾  ((0 . 0) (1 . 1) (2 . 2) (3 . 3) (4 . 4) (5 . 5))
(make-relation (lambda (x y) (= (+ x 1) y)) u)
  ⟾  ((0 . 1) (1 . 2) (2 . 3) (3 . 4) (4 . 5))
(make-relation (lambda (x y) (= (modulo (+ x 1) (length u)) y)) u)
  ⟾  ((0 . 1) (1 . 2) (2 . 3) (3 . 4) (4 . 5) (5 . 0))
(make-relation (lambda (x y) (< x y)) u)
  ⟾  ((0 . 1) (0 . 2) (0 . 3) (0 . 4) (0 . 5) (1 . 2) (1 . 3) (1 . 4) (1 . 5) (2 . 3) (2 . 4)
      (2 . 5) (3 . 4) (3 . 5) (4 . 5))
```

Protože relace jsou speciální množiny, můžeme na ně aplikovat procedury, které jsme výše vytvářeli pro množiny bez jakýchkoli změn:

```
(define r1 (make-relation (lambda (x y) (= (modulo (+ x 1) (length u)) y)) u))
(define r2 (make-relation (lambda (x y) (< x y)) u))

(union r1 r2)
  ⟾  ((5 . 0) (0 . 1) (0 . 2) (0 . 3) (0 . 4) (0 . 5) (1 . 2) (1 . 3)
      (1 . 4) (1 . 5) (2 . 3) (2 . 4) (2 . 5) (3 . 4) (3 . 5) (4 . 5))
(intersect r1 r2)
  ⟾  ((0 . 1) (1 . 2) (2 . 3) (3 . 4) (4 . 5))
(minus r1 r2)
  ⟾  ((5 . 0))
(minus r2 r1)
  ⟾  ((0 . 2) (0 . 3) (0 . 4) (0 . 5) (1 . 3) (1 . 4) (1 . 5) (2 . 4) (2 . 5) (3 . 5))
```

Níže budeme potřebovat proceduru, která pro libovolné dvě relace vrátí jejich nejmenší společné univerzum. Použitím procedury `map` pro každou z těch dvou relací vytvoříme seznam všech prvků vyskytujících se v prvních prvcích dvojic a seznam všech prvků vyskytujících se v druhých prvcích dvojic. A tyto čtyři seznamy spojíme aplikací procedury `append`. Z výsledného seznamu pak vytvoříme množinu pomocí procedury `list->set`:

```
(define get-universe
  (lambda (rel-R rel-S)
    (list->set (append (map 1st rel-R) (map 2st rel-R)
                       (map 1st rel-S) (map 2nd rel-S)))))
```

Tuto proceduru můžeme zobecnit na proceduru libovolného počtu argumentů. `get-universe` tak bude vracet nejmenší společné univerzum pro jakýkoli počet relací. Procedura, která vznikne vyhodnocením λ-výrazu

```
(lambda (x) (append (map 1st x) (map 2nd x))),
```

vrací seznam prvků vyskytujících se v prvních prvcích dvojic v relaci, na kterou je aplikována. Tuto proceduru mapujeme na seznam relací `relations`. Na seznam seznamů, který bude výsledkem tohoto mapování, aplikujeme proceduru `append` a vytvoříme tak jeden seznam. Z toho vytvoříme množinu procedurou `list->set`, viz následující kód:

```
(define get-universe
  (lambda relations
    (list->set (apply append
                      (map (lambda (x)
                             (append (map 1st x) (map 2nd x)))
                           relations)))))
```

Zde uvádíme aplikace procedury `get-universe` na různý počet relací:

```
(get-universe)                                  ⟾  ()
(get-universe '())                              ⟾  ()
(get-universe '() '() '())                      ⟾  ()
```

```
(get-universe '() '((a . b)) '())                    ⟹  (a b)
(get-universe '((10 . 20) (20 . 30)) '((a . b)) '())  ⟹  (10 20 30 a b)
```

K relaci R můžeme uvažovat inverzní relaci R^{-1}, to jest relaci $\{\langle b, a\rangle \mid \langle a, b\rangle \in R\}$. Vytvoříme proceduru `invert-relation`, která k dané relaci vrací inverzní relaci:

```
(define invert-relation
  (lambda (rel-R)
    (map (lambda (x)
           (make-tuple (2nd x) (1st x)))
         rel-R)))
```

Použitím procedury `map` jsme na každý prvek seznamu, který reprezentuje relaci aplikovali proceduru, která převrací pořadí prvků v uspořádané dvojici. Následuje ukázka použití procedury `invert-relation`.

```
(invert-relation '())  ⟹  ()
(invert-relation r1)   ⟹  ((1 . 0) (2 . 1) (3 . 2) (4 . 3) (5 . 4) (0 . 5))
```

Relace lze také skládat. Máme-li binární relace R a S na množině M, jejich složením rozumíme takovou binární relaci $R \circ S$ na množině M, že platí $\langle x, y\rangle \in R \circ S$, právě když existuje prvek $z \in M$ tak, že máme: $\langle x, z\rangle \in R$ a $\langle z, y\rangle \in S$. Procedura `compose-relations` bere dva argumenty, kterými jsou relace, a vrací relaci jejich složení.

```
(define compose-relations
  (lambda (rel-R rel-S)
    (let ((universe (get-universe rel-R rel-S)))
      (make-relation
       (lambda (x y)
         (exists (lambda (z)
                   (and (in? (make-tuple x z) rel-R)
                        (in? (make-tuple z y) rel-R)))
                 universe))
       universe))))
```

Implementace této procedury je vlastně přímým přepisem uvedeného předpisu. Nejdříve jsme pomocí procedury `get-universe` získali univerzum M a nad tímto univerzem jsme vytvořili relaci aplikací procedury `make-relation`. Vstupním argumentem této procedury byl mimo univerza predikát realizující vlastnost „existuje $z \in M$: $\langle x, z\rangle \in R$ a $\langle z, y\rangle \in S$“. V něm jsme použili proceduru existenčního kvantifikátoru `exists` a predikát `in?`. Nyní tedy ukážeme aplikaci této procedury:

```
(compose-relations r1 r2)
   ⟹  ((0 . 2) (1 . 3) (2 . 4) (3 . 5) (4 . 0) (5 . 1))
(compose-relations r2 r1)
   ⟹  ((1 . 3) (1 . 4) (1 . 5) (2 . 4) (2 . 5) (3 . 5) (0 . 2) (0 . 3) (0 . 4) (0 . 5))
```

O binární relaci R na množině M říkáme, že je *reflexivní*, pokud pro každé $x \in M$ platí $\langle x, x\rangle \in R$. Predikát `reflexive?`, který zjišťuje zda je relace reflexivní bychom mohli napsat například tak, jak je uvedeno níže. Opětně jde o přímý přepis předpisu.

```
(define reflexive?
  (lambda (rel-R)
    (forall (lambda (x)
              (in? (make-tuple x x) rel-R))
            (get-universe rel-R))))
```

A zde uvádíme aplikace:

```
(reflexive? (make-relation (lambda (x y) #f) u))                    ⟹  #t
(reflexive? (make-relation (lambda (x y) (= x y)) u))               ⟹  #t
(reflexive? (make-relation (lambda (x y) (= (+ x 1) y)) u))         ⟹  #f
(reflexive? (make-relation (lambda (x y) (= (modulo x 2) (modulo y 2))) u))  ⟹  #t
(reflexive? (make-relation (lambda (x y) (< x y)) u))               ⟹  #f
```

Binární relace R na množině M je *symetrická*, pokud pro všechna $x, y \in M$ platí, že pokud $\langle x, y\rangle \in R$, pak $\langle y, x\rangle \in R$:

```
(define symmetric?
  (lambda (rel-R)
    (let ((universe (get-universe rel-R)))
      (forall (lambda (x)
                (forall (lambda (y)
                          (if (in? (cons x y) rel-R)
                              (in? (cons y x) rel-R)
                              #t))
                        universe))
              universe))))
```

Zase šlo o přímý přepis uvedené definice. Zde uvádíme použití:

```
(symmetric? (make-relation (lambda (x y) #f) u))                          ⟹  #t
(symmetric? (make-relation (lambda (x y) (= x y)) u))                     ⟹  #t
(symmetric? (make-relation (lambda (x y) (= (+ x 1) y)) u))               ⟹  #f
(symmetric? (make-relation (lambda (x y) (= (modulo x 2) (modulo y 2))) u))  ⟹  #t
(symmetric? (make-relation (lambda (x y) (< x y)) u))                     ⟹  #f
```

Poslední vlastností relací jíž se budeme zabývat je tranzitivita. Binární relace R na množině M je tranzitivní, pokud pro všechna $x, y, z \in M$ platí, že pokud $\langle x, y\rangle \in R$ a $\langle y, z\rangle \in R$, pak $\langle x, z\rangle \in R$:

```
(define transitive?
  (lambda (rel-R)
    (let ((universe (get-universe rel-R)))
      (forall (lambda (x)
                (forall (lambda (y)
                          (forall (lambda (z)
                                    (if (and (in? (cons x y) rel-R)
                                             (in? (cons y z) rel-R))
                                        (in? (cons x z) rel-R)
                                        #t))
                                  universe))
                        universe))
              universe))))
```

Použití:

```
(transitive? (make-relation (lambda (x y) #f) u))                         ⟹  #t
(transitive? (make-relation (lambda (x y) (= x y)) u))                    ⟹  #t
(transitive? (make-relation (lambda (x y) (= (+ x 1) y)) u))              ⟹  #f
(transitive? (make-relation (lambda (x y) (= (modulo x 2) (modulo y 2))) u))  ⟹  #t
(transitive? (make-relation (lambda (x y) (< x y)) u))                    ⟹  #t
```

Shrnutí

Zabývali jsme se explicitní aplikací procedur a vyhodnocováním elementů. Představili jsme proceduru `apply` pomocí niž je možné aplikovat procedury s danými argumenty. Explicitní aplikaci procedur lze použít k agregaci více elementů do jednoho výsledku pomocí aplikace procedury. Ukázali jsme, jak lze s pomocí `apply` naprogramovat některé často používané procedury. Typickou úlohou, kterou lze vyřešit pomocí `apply` je filtrace prvků seznamu. V další části jsme se zabývali problematikou vytváření uživatelsky definovatelných procedur s nepovinnými a s libovolnými argumenty. Provedli jsme rozšíření λ-výrazů, které již si ponecháme (rozšíření bylo definitivní). Dále jsme ukázali proceduru `eval` a další sadu speciálních forem a procedur pro manipulaci s prostředím. Uvedli jsme, že prostředí lze bez újmy chápat jako element

jazyka Scheme, dokonce jako element prvního řádu. Upozornili jsme na úskalí při používání `eval` související se vznikem těžko odhalitelných chyb v programech. Nakonec jsme ukázali reprezentaci množin a relací pomocí seznamů, na které jsme demonstrovali použití procedur `apply` a `eval`.

Pojmy k zapamatování

- implicitní aplikace procedury, explicitní aplikace procedury,
- implicitní vyhodnocení elementů, explicitní vyhodnocení elementů,
- filtrace,
- nepovinné argumenty, libovolné argumenty,
- prostředí jako element prvního řádu.

Nově představené prvky jazyka Scheme

- speciální forma `the-environment`,
- procedury `apply`, `eval`, `environment-parent`, `procedure-environment`, `environment->list`

Kontrolní otázky

1. *Jaký je rozdíl mezi implicitní a explicitní aplikací procedury?*
2. *Jaké argumenty může mít procedura* `apply` *a jaký mají význam?*
3. *V kterých případech je nutné použít* `apply`*?*
4. *Jak se zapisují formální argumenty procedur s nepovinnými argumenty?*
5. *Jak se zapisují formální argumenty procedur s libovolnými argumenty?*
6. *Jaké omezení platí při použití nepovinných argumentů?*
7. *Jaké znáte speciální formy a procedury pro práci s prostředím?*
8. *Jak se provádí explicitní vyhodnocování elementů?*

Cvičení

1. Naprogramujte proceduru na zpracovaní seznamu podle vzoru. Vzorem se myslí seznam o stejném počtu prvků, který obsahuje:
   - symbol `del`, pak je prvek na odpovídající pozici smazán
   - symbol `ins`, pak je prvek na odpovídající pozici ponechán
   - procedury, pak je prvek seznamu na odpovídající pozici nahrazen aplikací této procedury na tento prvek.

   Viz příklad volání:

   ```
   (format '(1 2 3 4 5) (list 'del even? even? 'ins (lambda (x) (+ x 1))))
      ⟹ (#t #f 4 5)
   ```
2. Naprogramujte proceduru vracející seznam mocnin čísla k (od 0-té až po $(n-1)$-té).
3. Naprogramujte proceduru konverze `binary->decimal`, které binární číslo, reprezentované číselným seznamem obsahujícím `0` a `1`, převede na číslo (Scheme-ovské).
4. Implementujte pro reprezentaci množin uvedenou v sekci 6.5:
   - operaci symetrického rozdílu
   - predikát rovnosti množin
   - predikát zjišťující, zda je zadaný seznam množinou

Úkoly k textu

1. Naprogramujte relace reprezentované páry $\langle univerzum \rangle$. $\langle vlastnost \rangle$. Kde $\langle univerzum \rangle$ seznam, a $\langle vlastnost \rangle$ je predikát představující funkci příslušnosti.
2. Implementujte reprezentace zobrazení jako speciálních relací.

Řešení ke cvičením

1. ```
   (define format
     (lambda (l pattern)
       (apply append
              (map (lambda (atom pat)
                     (cond ((equal? pat 'del) '())
                           ((equal? pat 'ins) (list atom))
                           ((procedure? pat) (list (pat atom)))
                           (else #f)))
                   l pattern))))

   (format '(1 2 3 4 5) (list 'del even? even? 'ins (lambda (x) (+ x 1))))
   ```
2. ```
   (define (power-list k n)
     (apply map *
            (map
             (lambda (r)
               (build-list n (lambda (i)
                               (if (<= i r)
                                   1
                                   k))))
             (build-list n (lambda (i) i)))))
   ```
3. ```
   (define (binary->decimal bin)
     (apply + (map * (reverse bin) (power-list 2 (length bin)))))
   ```
4. ```
   (define sminus
     (set-operation
      (lambda (x A B)
        (or (and (in? x A) (not (in? x B)))
            (and (in? x B) (not (in? x A)))))))

   (define sminus
     (lambda (set-A set-B)
       (union (minus set-A set-B)
              (minus set-B set-A))))
   ```

   - ```
     (define set-equal?
       (lambda (set-A set-B)
         (null? (sminus set-A set-B))))
     ```
   - ```
     (define set?
       (lambda (elem)
         (apply and-proc
                (map-tail (lambda (x)
                            (not (in? (car x) (cdr x))))
                          elem))))

     (define set?
       (lambda (elem)
         (equal? (list->set elem) elem)))
     ```