KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 4: Tečkové páry, symbolická data a kvotování

Učební materiál k přednášce 26. října 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 4: Tečkové páry, symbolická data a kvotování

Obsah lekce: V této lekci se budeme zabývat vytvářením abstrakcí pomocí dat. Představíme tečkové páry pomocí nichž budeme vytvářet hierarchické datové struktury. Ukážeme rovněž, že tečkové páry bychom mohli plně definovat pouze pomocí procedur vyšších řádů. Dále se budeme zabývat kvotováním a jeho zkrácenou notací jenž nám umožní chápat symboly jako data. Jako příklad datové abstrakce uvedeme implementaci racionální aritmetiky.

Klíčová slova: datová abstrakce, konstruktory, kvotování, selektory, symbolická data, tečkové páry.

4.1 Vytváření abstrakcí pomocí dat

Doteď jsme vytvářeli abstrakce pomocí procedur: používali jsme procedury k vytváření složitějších procedur, přitom jsme se nemuseli zabývat tím, jak jsou jednodušší procedury vytvořeny (pokud fungují správně). Teď provedeme totéž s daty: budeme pracovat se *složenými* (*hierarchickými*) *daty* a budeme konstruovat procedury, které je budou zpracovávat – budou je brát jako argumenty nebo vracet jako výsledek své aplikace. Přitom to opět budeme provádět tak, aby bylo procedurám de facto jedno, jak jsou složená data konstruována z jednodušších. Nejprve si ukážeme několik motivačních příkladů, z nichž pak vyplyne potřeba vytvářet a zpracovávat *složená data*.

Příklad 4.1. Uvažujme, že bych chtěli napsat proceduru, která má nalézt kořeny kvadratické rovnice

$$ax^2 + bx + c = 0$$

na základě jejich koeficientů podle důvěrně známého vzorce

$$\frac{-b \pm \sqrt{D}}{2a}, \quad \text{kde } D = b^2 - 4ac.$$

Takové kořeny jsou v oboru komplexních čísel dva: x_1 a x_2 (popřípadě jeden dvojitý). Proceduru pro výpočet kořenů, která má tři argumenty reprezentující koeficienty kvadratické rovnice, bychom mohli napsat například následujícím způsobem. Kód je ponechán neúplný, protože nám zatím schází prostředek, jak vrátit oba kořeny současně:

```
(define koreny
  (lambda (a b c)
    (let* ((diskr (- (* b b) (* 4 a c)))
           (koren1 (/ (+ (- b) (sqrt diskr)) (* 2 a)))
           (koren2 (/ (- (- b) (sqrt diskr)) (* 2 a))))
      teď bychom chtěli vrátit dvě hodnoty současně
     )))
```

V tuto chvíli se samozřejmě nabízí celá řada „hloupých řešení“, jak obejít problém vracení dvou hodnot. Mohli bychom například proceduru rozšířit o další parametr, který by určoval, který kořen máme vrátit. Kód by pak vypadal následovně:

```
(define koreny
  (lambda (a b c p)
    (let ((diskr ((- (* b b) (* 4 a c)))))
       (/ ((if p + -) (- b) (sqrt diskr)) 2 a))))
```

Předchozí řešení problému není šťastné, kdybychom totiž dále chtěli pracovat s oběma kořeny, museli bychom proceduru aplikovat dvakrát a pokaždé by docházelo k vyhodnocování téhož kódu se stejnými argumenty (jedná se třeba o výpočet diskriminantu). Jiným takovým řešením je namísto jedné procedury `koreny` mít dvě procedury `koren1` a `koren2`, přitom by každá vracela jeden z kořenů. Zavádění dvou procedur namísto jedné ale znesnadňuje údržbu kódu a stává se zdrojem chyb. Navíc by nám to nijak nepomohlo, pořád by to vedlo k problému násobného vyhodnocování téhož kódu. V případě dvojice kořenů kvadratické rovnice není tento problém zas až tak markantní. Jeho význam bude zřejmější v následujícím příkladě.

Příklad 4.2. Uvažme, že bychom ve Scheme neměli racionální aritmetiku, a chtěli bychom navrhnout systém na provádění aritmetických operací nad racionálními čísly (sčítání, odčítání, násobení, dělení a tak dále). Chtěli bychom například implementovat proceduru `r+`, která bere jako parametry dvě racionální čísla a vrací jejích součet. Z pohledu primitivních dat uvažujeme racionální číslo jako dvě celá čísla – čitatele a jmenovatele. S touto dvojicí potřebujeme zacházet jako s jednou hodnotou. Pro bližší ilustraci uvedeme neúplnou definici procedury `r+` s vloženým textem poukazujícím na potřebu používat hierarchická data:

```
(define r+
  (lambda (cit1 jmen1 cit2 jmen2)
            proceduře bychom chtěli předávat jen dva argumenty = dvě racionální čísla
     (let* ((cit (+ (* cit1 jmen2) (* cit2 jmen1)))
            (jmen (* jmen1 jmen2))
            (delit (gcd cit jmen))))
       teď bychom chtěli vrátit dvě hodnoty současně: cit a jmen
     )))
```

Obdobně jako v předchozím příkladě bychom mohli mít zvlášť proceduru na výpočet čitatele a zvlášť proceduru na výpočet jmenovatele, nebo pomocí dalšího argumentu určovat, kterou složku racionálního čísla (čitatele nebo jmenovatele) má procedura vrátit. Ve většině výpočtů ale budeme potřebovat i čitatele i jmenovatele. Navíc kdybychom neustále pracovali se dvěma hodnotami namísto jedné a pro každou operaci bychom museli mít dvě procedury (vracející čitatele a jmenovatele z výsledné hodnoty). Program by se stal velmi brzy nepřehledným. Tato podvojnost procedur a čísel by také velice znesnadňovala udržování kódu a stala by se potenciálním zdrojem chyb.

Příklad 4.3. V dalším příkladě se snažíme navrhnout program pracující s geometrickými útvary, ve kterém bychom chtěli pracovat s pojmy jako bod, úsečka, kružnice a tak dále. Bod v rovině je z pohledu primitivních dat dvojice reálných čísel; úsečku můžeme reprezentovat (třeba) jako dvojici bodů; kružnici (třeba) jako bod (střed) a reálné číslo (poloměr). Rozumným požadavkem na takový geometrický program by bylo mít v něm například k dispozici procedury pro výpočet různých průsečíků: třeba procedura na výpočet průsečíku dvou úseček bude vracet bod. Při vytváření programu bychom záhy narazili na potřebu „slepit“ dvě souřadnice dohromady, abychom je mohli považovat za bod; „slepit“ dva body dohromady a vytvořit tak reprezentaci úsečky a podobně. Z pohledu tohoto příkladu je tady patrné, že jako programátoři bychom měli mít k dispozici prostředky, které nám umožní reprezentovat abstraktnější data než jen „numerické hodnoty“. Konec konců, prakticky všechny soudobé programy pracují s mnohem složitějšími daty než jsou jen čísla.

Z těchto tří motivačních příkladů vyplývá zřejmý požadavek: Obdobně jako jsme v lekci 2 vytvářeli nové procedury, potřebujeme teď vytvářet *větší datové celky z primitivních dat* a pracovat s nimi jako by se jednalo o *jeden element*. Potřebujeme tedy vytvářet *abstrakce pomocí dat*. Jakmile to budeme moci udělat, můžeme například přestat uvažovat racionální číslo jako dvě čísla – čitatele a jmenovatele – a můžeme začít s ním pracovat na vyšší úrovni abstrakce jako s celkem, to jest s elementem reprezentujícím „racionální číslo“. Stejně tak v ostatních motivačních příkladech můžeme pracovat s elementy reprezentující „dvojice kořenů kvadratické rovnice“, „bod“, „úsečka“, „kružnice“, a tak dále.

Abychom docílili datové abstrakce, musíme odpovědět na dvě otázky:

1. Jak implementovat abstraktní data pomocí konkrétní (fyzické) datové reprezentace?
   Neboli potřebujeme prostředek, který nám umožní z *primitivních dat* vytvářet *složená data*.
2. Jak odstínit program od této konkrétní datové reprezentace?
   Když budeme pracovat se složenými daty, budeme s nimi chtít zacházet jako s pojmem, který reprezentují. Nebude nás zajímat jak a kde jsou data konkrétně uložena. Například v případě kořenů kvadratických rovnic chceme mít k dispozici procedury jako „vytvoř řešení“, „vrať první kořen z řešení“, „vrať druhý kořen z řešení“, a další procedury, které pracují s kořeny. Z pohledu uživatelů těchto procedur nás už ale nebude zajímat jakým způsobem a kde jsou hodnoty uloženy.

   Kvůli odstínění programu od konkrétní datové reprezentace proto pro každá složená data vytváříme:

konstruktory – procedury sloužící k vytváření složených dat z jednodušších, s vytvořenými daty dále pracujeme jako s jednotlivými elementy;

selektory – procedury umožňují přistupovat k jednotlivým složkám složených dat.

4.2 Tečkové páry

Ve Scheme máme k dispozici elementy jazyka nazývané *tečkové páry* (též jen *páry*). Tečkový pár je vytvořen spojením dvou libovolných elementů do uspořádané dvojce. Prvky v páru pak nazýváme *první prvek páru* a *druhý prvek páru*. Pár je zkonstruován primitivní procedurou `cons`, procedury pro přístup k jeho jednotlivým prvkům jsou navázané na symboly `car` a `cdr`. Z pohledu terminologie v předchozí sekci je `cons` *konstruktor páru* (vytváří pár z jednoduchých dat) a `car` a `cdr` jsou *selektory* (umožňují přistupovat k prvkům párů). Viz následující upřesňující definici.

Definice 4.4. Procedura `cons` se používá se dvěma argumenty

`(cons` ⟨*první prvek*⟩ ⟨*druhý prvek*⟩`)`

a vrací pár obsahující tyto dva argumenty ⟨*první prvek*⟩ a ⟨*druhý prvek*⟩ jako svoje prvky.
Máme-li pár ⟨*pár*⟩, můžeme extrahovat tyto části pomocí primitivních procedur `car` a `cdr`. Procedury `car` a `cdr` se používají s jedním argumentem

`(car` ⟨*pár*⟩`)`,
`(cdr` ⟨*pár*⟩`)`.

Procedura `car` jako výsledek své aplikace vrací první prvek předaného páru. Procedura `cdr` jako výsledek své aplikace vrací druhý prvek předaného páru. V případě, že by `car` nebo `cdr` byly aplikovány s elementem, který není pár, pak dojde k chybě „CHYBA: Argument předaný proceduře musí být pár“. ■

Poznámka 4.5. Původ jmen *cons*, *car* a *cdr*: Jméno procedury `cons` znamená „construct“ (vytvoř). Jména `car` a `cdr` pocházejí z původní implementace LISPu na počítači IBM 704. Tento stroj měl adresovací schéma, které povolovalo odkazovat se na místa v paměti nazvané „address“ a „decrement“. Zkratka „car“ znamená „Contents of Address part of Register“ a zkratka „cdr“ (čteno „kudr“) znamená „Contents of Decrement part of Register,“ pro detaily viz například [SICP].

Příklad 4.6. Uvažujme, že na symboly `par1` a `par2` navážeme dva páry následujícím způsobem:

```
(define par1 (cons 1 2))
(define par2 (cons par1 3))
```

(a) Výraz `(car par1)` se vyhodnotí na číslo `1`, protože na symbol `par1` je navázán pár, který má první prvek číslo `1` (a jeho druhým prvkem je číslo `2`). Aplikací procedury `car` dostaneme toto číslo.

(b) Výsledkem vyhodnocení výrazu `(cdr par2)` bude číslo `3`. Na symbol `par2` je navázán pár, jehož druhý prvek je číslo `3`, které získáme aplikací procedury `cdr`. Výsledkem vyhodnocení výrazu `(car par2)` bude pár obsahující jako svůj první prvek jedničku a jako druhý prvek dvojku. Z příkladu je myslím jasné, že prvky párů mohou být obecně jakékoliv elementy, tedy i další páry.

(c) Vyhodnocení výrazu `(car (car par2))` bude vypadat následovně: Jelikož je na symbol `car` navázána procedura, vyhodnotíme předaný argument – tedy seznam `(car par2)`. Vyhodnocením tohoto seznamu dostaneme pár, který má první prvek číslo `1` a druhý prvek je číslo `2` – vyhodnocení tohoto podvýrazu bylo již popsáno v bodě (b). Aplikací procedury `car` na tento pár dostaneme číslo `1`. Obdobně číslo `2` dostaneme jako výsledek vyhodnocení `(cdr (car par2))`.

(d) Při vyhodnocení výrazu `(cdr (cdr par2))` dostaneme chybu „CHYBA: Argument není pár“, protože výsledek vyhodnocení výrazu `(cdr par2)` je číslo `3`, tedy druhá (vnější) aplikace `cdr` selže, viz komentář v definici 4.4.

Při složitějších strukturách párů se může stát, že sekvence volání `car` a `cdr` budou dlouhé a nepřehledné, proto jsou ve Scheme k disposici složeniny. Například místo `(car (cdr p))` můžeme psát jen `(cadr p)`.

Ve Scheme jsou složeniny až do hloubky čtyři. Například na symboly `caaaar`, `cadadr` a `cddaar` budou ještě navázány složené selektory, ale na třeba na `cddadar` již ne. Složenin je tedy celkem 28.

Před tím než uvedeme další příklad podotkněme, že složeniny bychom mohli vytvářet sami:

```
(define caar (lambda (p) (car (car p))))
(define cadr (lambda (p) (car (cdr p))))
(define cdar (lambda (p) (cdr (car p))))
(define cddr (lambda (p) (cdr (cdr p))))
(define caaar (lambda (p) (car (caar p))))
  ⋮
```

Příklad 4.7. Uvažujme pár, který navážeme na symbol `par` vyhodnocením následujícího výrazu:

```
(define par (cons (cons 10 (cons 20 30)) 40)).
```

(a) Výraz `(caar par)` se vyhodnotí na číslo deset. Jde o totéž jako bychom psali `(car (car par))`.

(b) Výraz `(cdar par)` se vyhodnotí na číslo 2.

(c) Vyhodnocení výrazu `(caaar par)` skončí chybou „CHYBA: Argument není pár“.

(d) V předchozích příkladech jsme vždy používali selektory na páry, které jsme předtím navázali na symboly pomocí speciální formy `define`. Samozřejmě, že `define` se samotnými páry nic společného nemá a selektory bychom mohli použít přímo na páry vytvořené konstruktorem `cons` bez provádění jakýchkoliv vazeb, například:

```
(caar (cons (cons 10 20) (cons 30 40)))  ⟹  10
(cdar (cons (cons 10 20) (cons 30 40)))  ⟹  20
(cadr (cons (cons 10 20) (cons 30 40)))  ⟹  30
(cddr (cons (cons 10 20) (cons 30 40)))  ⟹  40
```

Než si ukážeme několik příkladů na použití tečkových párů, popíšeme, jak vypadá *externí reprezentace tečkových párů*, a také si ukážeme, jak se páry znázorňují graficky, pomocí tak zvané *boxové notace*.

Externí reprezentace páru majícího první prvek $\langle A \rangle$ a druhý prvek $\langle B \rangle$ je následující: `(`$\langle A \rangle$ `.` $\langle B \rangle$`)`. Z této notace pro externí reprezentaci tečkových párů je asi jasné, proč se o nich říká, že jsou „*tečkové*“. Viz příklad:

```
(cons 1 2))  ⟹  (1 . 2)
```

V případě, že druhý prvek páru je zase pár, se používá *zkrácený zápis*. V tomto zkráceném zápisu se vynechávají závorky náležející tomu vnitřnímu páru a tečka náležející vnějšímu. Třeba pár `(`$\langle A \rangle$ `.` `(`$\langle B \rangle$ `.` $\langle C \rangle$`))` můžeme zkráceně zapsat jako pár `(`$\langle A \rangle$ $\langle B \rangle$ `.` $\langle C \rangle$`)`.

Příklad 4.8. V tomto příkladu vidíme externí reprezentace párů:

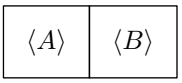
```
(cons 10 20)                      ⟹  (10 . 20)
(cons (cons 10 20) 30)            ⟹  ((10 . 20) . 30)
(cons 10 (cons 20 30))            ⟹  (10 . (20 . 30)) = (10 20 . 30)
(cons (cons 10 (cons 20 30)) 40)  ⟹  ((10 . (20 . 30)) . 40) = ((10 20 . 30) . 40)
(cons 10 (cons (cons 20 30) 40))  ⟹  (10 . ((20 . 30) . 40)) = (10 (20 . 30) . 40)
(cons (cons 10 20) (cons 30 40))  ⟹  ((10 . 20) . (30 . 40)) = ((10 . 20) 30 . 40)
```

Naopak následující výrazy nejsou externí reprezentace tečkových párů:

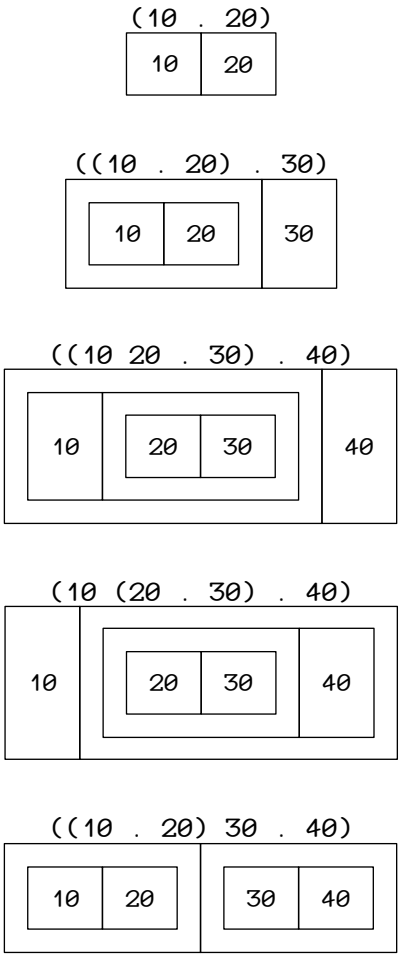
`(10 . ),( . 20),(10 . 20 . 30)` a `(10 . 20 30)`.

Nyní si ukážeme grafické znázornění tečkových párů – *boxovou notaci*. Pár je zobrazen jako obdélníček (box) rozdělený na dvě části. Do levé části boxu se zakresluje první prvek páru a do pravé druhý prvek páru. Tedy pár `(`$\langle A \rangle$ `.` $\langle B \rangle$`)` vypadá v boxové notaci tak, jak je znázorněno na obrázku 4.1.

Obrázek 4.1. Boxová notace tečkového páru.



Obrázek 4.2. Tečkové páry z příkladu 4.8 v boxové notaci



Příklad 4.9. Tečkové páry z příkladu 4.8 v boxové notaci najdeme na obrázku 4.2.

Příklad 4.10. Přirozeně nemusíme konstruovat páry jen z čísel a párů, jako v doposud uvedených příkladech. Můžeme vytvářet páry z jakýchkoli elementů:

```
(cons + -)              ⟼  („procedura sčítání“ . „procedura odčítání“)
(cons and not)          ⟼  („speciální forma and“ . „procedura negace“)
(cons (if #f #f) #t)    ⟼  („nedefinovaná hodnota“ . #f)
```

> Páry jsou elementy prvního řádu. Pokud se vrátíme k definici 2.17 na straně 55, pak můžeme jasně vidět, že páry je možno pojmenovat, předávat jako argumenty procedurám, vracet jako výsledky aplikace procedur a mohou být obsaženy v hierarchických datových strukturách (páry mohou být opět obsaženy v jiných párech).

Poslední, co zbývá k tečkovým párům říct, je to, jak vlastně páry zapadají do Read a Eval částí REPL cyklu. Existenci párů jsme v lekci 1 při definici symbolických výrazů zatajili. Ve skutečnosti i externí reprezentace řady tečkových párů je čitelná readerem [4]. V tuto chvíli můžeme obohatit množinu přípustných symbolických výrazů jazyka Scheme o speciální výrazy, které se shodují s externí reprezentací některých tečkových párů. Rozšíříme definici 1.6 ze strany 19 o následující nový bod:

- Jsou-li $e, f, e_1, e_2, \ldots, e_n$ symbolické výrazy $(n \geq 0)$, pak

  $(e\ \ e_1\ \ e_2\ \cdots\ e_n\ .\ f)$

  je symbolický výraz. Pokud je $n = 0$, symbolický výraz $(e\ .\ f)$ nazveme *pár*.

Teď nám ale dochází k nepříjemné dvojznačnosti: S-výraz $(e\ .\ f)$ teď můžeme považovat za tříprvkový seznam obsahující výraz e, symbol „.“ a výraz f, a také za tečkový pár obsahující výrazy e a f. Aby k této dvojznačnosti nedocházelo, je potřeba zpřísnit definici symbolu o následující rys: Samostatný znak „.“ *není symbol*. S-výraz $(e\ .\ f)$ tak může být jedině pár.

Další nejednoznačnost je v terminologii. Musíme si ujasnit, že i když nazýváme *symbolický výraz pár* i *element pár* shodně, to jest „pár“, jedná se o dvě různé věci. Pár jako *symbolický výraz* určuje jak může vypadat část programu (je to syntaktický pojem), kdežto pár jako element reprezentuje konkrétní hodnoty, se kterými pracuje interpret při vyhodnocování, jedná se tedy o sémantický pojem těsně spjatý s (abstraktním) interpretem jazyka Scheme.

Řekli jsme si tedy, že páry jsou čitelné readerem. Co jsme ale ještě neřekli je, jak se vyhodnocují. Podle toho, jak jsme nadefinovali Eval v definici 1.21 na straně 26, by se měl pár podle bodu (D) vyhodnotit sám na sebe. Podotkněme nyní, že z praktického hlediska se interprety jazyka Scheme dávají na vyhodnocování párů jinak. Zatím vyhodnocování párů ponecháme stranou a popíšeme je až v další sekci.

Na konci této sekce uvádíme několik zajímavých příkladů:

Příklad 4.11. V následujícím kódu vytváříme páry, které potom navazujeme na symboly. Nejprve na symbol `a` navážeme jeden pár a pomocí něj potom vytvoříme druhý pár a navážeme jej na symbol `b`. Pokud provedeme změnu vazby symbolu `a`, pár navázaný na symbolu `b` se (pochopitelně) nezmění:

```
(define a (cons 10 20))
(define b (cons (car a) 30))
b  ⟼  (10 . 30)
(define a (cons 5 -5))
b  ⟼  (10 . 30)
```

To vyplývá ze sémantiky speciální formy `define`, viz definici 1.26 na straně 1.26 a z faktu, že `cons` je procedura. Při vyhodnocení výrazu `(cons (car a) 30)` je aplikována procedura `cons` na vyhodnocení výrazů `(car a)` a `30` – to jest na čísla `10` a `30`. Výsledkem aplikace procedury `cons` je pak pár `(10 . 30)`, a je navázán na symbol `b`. Předefinováním hodnoty navázané na symbol `a` pomocí `define` se pak pár navázaný na symbol `b` nezmění.

[4] Někdy tomu tak ovšem být nemusí, vezměme si třeba tečkové páry z příkladu 4.10. Jejich externí reprezentace bude pro reader nečitelná, protože páry obsahují elementy s nečitelnou reprezentací (procedury, speciální formy a nedefinovanou hodnotu).

Příklad 4.12. Nadefinujeme proceduru, která vrací pár, jehož prvním prvkem je právě tato procedura:

```
(define a (lambda () (cons a #f)))
(a)                ⟼  („procedura“ . #f)
(car (a))          ⟼  „procedura“
((car (a)))        ⟼  („procedura“ . #f)
(car ((car (a))))  ⟼  „procedura“
  ⋮
```

Příklad 4.13. V lekci 2 jsme v příklady 2.11 na straně 68 definovali proceduru vyššího řádu `extrem` na vytváření procedur hledání extrémní hodnoty ze dvou argumentů vzhledem k nějakému uspořádání (viz program 2.11). Nyní můžeme vytvořit proceduru která vytváří dvě procedury současně – jednu na nalezení jednoho extrému a druhou na nalezení opačného extrému – a vrací je v páru:

```
(define extrem-pair
  (lambda (p?)
    (cons (extrem p?)
          (extrem (lambda (a b) (not (p? a b)))))))
```

Aplikací procedury `extrem` jednou na výchozí predikát a jednou na predikát vytvořený z výchozího pomocí negace podmínky dané tímto predikátem, jsme tedy dostali dvě procedury, ze kterých jsme pomocí `cons` vytvořili tečkový pár. Procedury `min` a `max` pomocí procedury `extrem-pair` můžeme definovat třeba takto:

```
(define extrems (extrem-pair <))
(define min (car extrems))
(define max (cdr extrems))
```

Poznámka 4.14. Vrátíme-li se nyní k motivačnímu příkladu s řešením kvadratické rovnice ze začátku této sekce, pak snadno nahlédneme, že bychom mohli chybějící tělo v proceduře `koreny` doplnit výrazem `(cons koren-1 koren-2)`. To by byl úplně nejjednodušší způsob, jak vrátit „dvě hodnoty současně.“ Z hlediska čistoty programu by již toto řešení bylo diskutabilní, jak uvidíme v další sekci.

4.3 Abstrakční bariéry založené na datech

V předchozí lekci jsme mimo jiné mluvili o vrstvách programu a o abstrakčních bariérách založených na procedurách. Abstrakční bariéry založené na datech jsou obecnější než abstrakční bariéry pomocí procedur, protože pro nás jsou procedury data (lze s nimi zacházet jako s daty). Procedury hrají ale i při datové abstrakci důležitou roli a tou je odstínění fyzické reprezentace dat od jejich uživatele.

Při vytváření abstrakčních bariér v programech bychom se tedy měli soustředit jako na procedury tak na data. Jednostranné zaměření na to či ono skoro nikdy nepřinese nic dobrého. Účelem datové abstrakce je oddělit data na nižší úrovni od abstraktnějších dat a mít pro ně vytvořené separátní konstruktory a selektory (pomocí nichž je abstrakční bariéra de facto vytvořena).

Opět zdůrazněme, že při vývoji programu je (i z pohledu dat) vhodné používat styl *bottom-up*, tedy pracovat a přemýšlet nad programem jako nad postupným obohacováním jazyka. V případě datové abstrakce jde o *rozšiřování jazyka* o *nové datové typy*. Pro ilustraci si uveďme následující příklad.

Příklad 4.15. Jako příklad použití abstraktních bariér uvádíme program 4.1 na výpočet kořenů kvadratické rovnice. Tento krátký program je rozdělen do tří vrstev, které jsou znázorněny na obrázku 4.3. Nejnižší vrstvu tvoří vrstva jazyka Scheme a konkrétní implementace tečkových párů. To jak tato implementace vypadá nás ve vyšších vrstvách nezajímá, pro nás jsou důležité konstruktory a selektory párů, které nám tato vrstva nabízí. Hned nad touto nejnižší vrstvou je vrstva implementace dvojice kořenů. Dvojici kořenů implementujeme pomocí tečkových párů, to pro nás ale v dalších vrstvách není důležité. Co je důležité jsou procedury `vrat-koreny`, `prvni-koren` a `druhy-koren` sloužící jako konstruktor a selektory pro dvojici kořenů. Dále již pracujeme jen s nimi. Všimněte si, že v procedurách `najdi-koreny` a `dvojnasobny?` neuvažujeme dvojice kořenů jako páry.

Například definici procedury `najdi-koreny` by bylo chybou namísto

```
(vrat-koreny ···
```

napsat

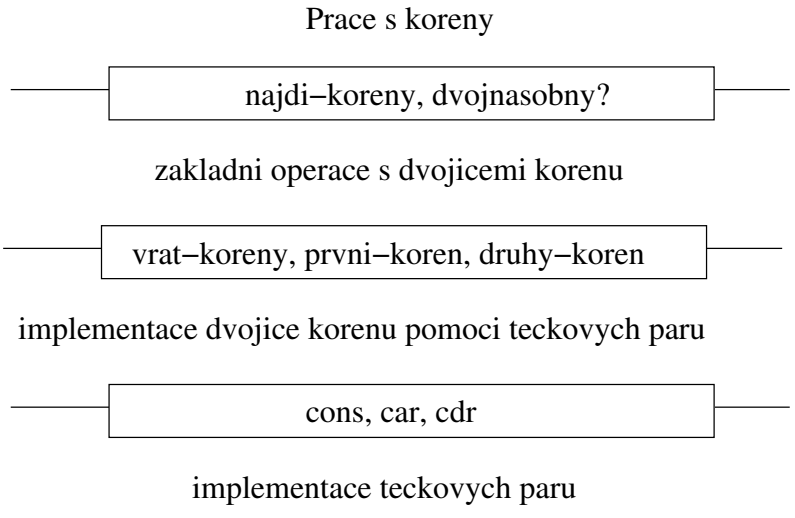
```
(cons ···,
```

protože by tím došlo k porušení pomyslné abstrakční bariéry. Na symboly `vrat-koreny` a `cons` je sice navázaná stejná procedura, ale při změně vrstvy „práce s kořeny“ (viz obrázek 4.3) bychom museli měnit i vyšší vrstvu. Třeba hned v úvodu další sekce ukážeme, jak bychom mohli zacházet s kořeny, kdybychom neměli k dispozici tečkové páry. Obdobným způsobem bychom mohli napsat i proceduru `vrat-koreny`.

Program 4.1. Příklad abstrakční bariéry: výpočet kořenů kvadratické rovnice.

```
(define vrat-koreny cons)
(define prvni-koren car)
(define druhy-koren cdr)

(define najdi-koreny
  (lambda (a b c)
     (let ((diskr (- (* b b) (* 4 a c))))
       (vrat-koreny (/ (- (- b) (sqrt diskr)) 2 a)
                    (/ (+ (- b) (sqrt diskr)) 2 a)))))

(define dvojnasobny?
  (lambda (koreny)
    (= (prvni-koren koreny)
       (druhy-koren koreny))))
```

Obrázek 4.3. Schéma abstrakčních bariér (NACRT OBRAZKU)



Pro každá data, která se v programu vyskytují, bychom měli vytvořit sadu nezávislých konstruktorů a selektorů. To vede ke snadné údržbě kódu a k snadným případným změnám implementace. Vždy je potřeba najít vhodné ekvilibrium mezi množstvím abstrakčních bariér a efektivitou programu (optima-

lizací programu týkající se jeho rychlosti, spotřeby systémových zdrojů a podobně). Tyto dvě kvality programu jdou leckdy proti sobě.

4.4 Implementace tečkových párů pomocí procedur vyšších řádů

V této sekci se budeme zabývat tím, zda-li je nutné k vytvoření elementů zapouzdřující dvojice hodnot uvažovat nové elementy jazyka (tečkové páry) tak, jak jsme to dělali doposud, nebo jestli není možné je modelovat s tím, co už máme v jazyku Scheme k dispozici. V této sekci ukážeme, že páry lze plně implementovat pouze s využitím procedur vyšších řádů a lexikálního rozsahu platnosti.

Nejdříve problematiku demonstrujeme na příkladu 4.1 s kořeny kvadratické rovnice z úvodu lekce. Poté tuto myšlenku zobecníme tak, že pomocí procedur vyšších řádů naimplementujeme vlastní tečkové páry.

Zopakujme, že chceme vytvořit proceduru, která by pro kvadratickou rovnici

$$ax^2 + bx + c = 0$$

zadanou jejími koeficienty a, b a c spočítala její dva kořeny a předpokládejme při tom, že nemáme k dispozici `cons`, `car` a `cdr` o kterých jsme doposud hovořili. Kód z příkladu 4.1 doplníme tak, jak je to ukázáno v programu 4.2. Do těla `let*`-bloku (které v kódu z úvodu lekce chybělo) jsme dali výraz:

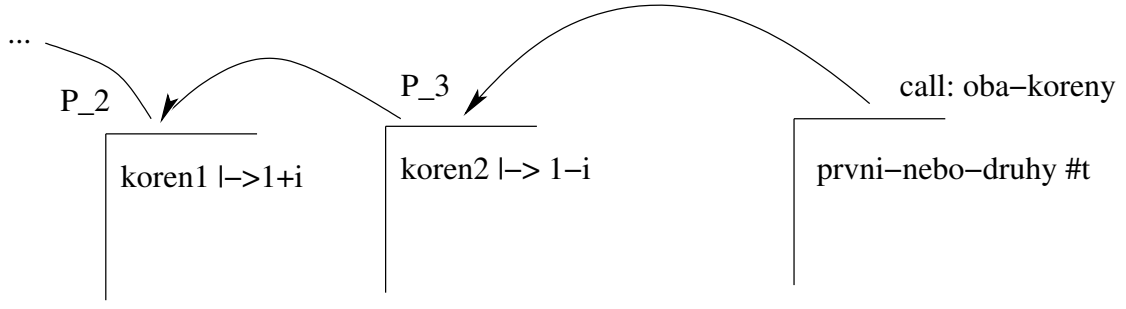
Program 4.2. Implementace procedury `koreny` pomocí procedur vyšších řádů.

```
(define koreny
  (lambda (a b c)
      (let* ((diskr (- (* b b) (* 4 a c)))
             (koren1 (/ (+ (- b) (sqrt diskr)) (* 2 a)))
             (koren2 (/ (- (- b) (sqrt diskr)) (* 2 a))))
        (lambda (prvni-nebo-druhy)
          (if prvni-nebo-druhy koren1 koren2)))))
```

```
(lambda (prvni-nebo-druhy)
  (if prvni-nebo-druhy
      koren1
      koren2))
```

Ten je při aplikaci procedury `koreny` vyhodnocen v posledním prostředí $\mathcal{P}_3$ (viz obrázek 4.4) vytvořeném vyhodnocením `let*`-bloku. Tedy v prostředí, ve kterém jsou známy vazby na symboly `koren1` a `koren2`. V tomto prostředí $\mathcal{P}_3$ nám tedy vzniká procedura, která na základě pravdivostní hodnoty, na kterou je aplikovaná, vrací buďto element navázaný na symbol `koren1` nebo na `koren2`.

Obrázek 4.4. Vznik prostředí při aplikaci procedury z příkladu 4.2 (NACRT OBRAZKU)



```
(koreny 1 -2 2)                          ⟼   procedura

(define oba-koreny (koreny 1 -2 2))
(oba-koreny #t)                          ⟼   1+i
(oba-koreny #f)                          ⟼   1-i
```

Při aplikaci procedury navázané na symbol `oba-koreny` s různými pravdivostními hodnotami jako argumenty tedy dostáváme jednotlivé kořeny. Je tomu skutečně tak, protože v prostředí vzniku procedury jsou symboly `koren1` a `koren2` navázané na hodnoty vypočtené již při aplikaci procedury `koreny`. Podle toho také můžeme vytvořit selektory `prvni-koren` a `druhy-koren`. Ty budou proceduru zapouzdřující oba kořeny aplikovat na pravdivostní hodnoty, podle toho, který z nich chceme:

```
(define prvni-koren (lambda (oba) (oba #t)))
(define druhy-koren (lambda (oba) (oba #f)))
(prvni-koren oba-koreny)   ⟼   1+i
(druhy-koren oba-koreny)   ⟼   1-i
```

Nyní z tohoto příkladu vytáhneme základní myšlenku – možnost uchovat vazby dvou symbolů v prostředí, které mohou reprezentovat „dvě hodnoty pohromadě“, a možnost přístupu k těmto prvkům pomocí procedur vyšších řádů. Ukážeme tedy, jak naimplementovat tečkové páry pouze s využitím procedur vyšších řádů a s využitím vlastností lexikálního rozsahu platnosti. Bez něj by dále uvedené úvahy o implementaci párů neměly význam (při použití dynamického rozsahu platnosti bychom velmi brzy narazili na fatální problémy). Naše implementace nám zároveň dá odpověď na fundamentální otázku, zda jsou páry definovatelné pomocí procedur vyšších řádů (odpověď bude kladná).

Pár v naší reimplementaci tedy bude procedura, která pro různé argumenty vrací různé prvky páru. Nově vytvářený `cons` bude procedurou vyššího řádu, která bude náš pár vytvářet. Podívejme se na následující implementaci procedury `cons`:

```
(define cons
  (lambda (x y)
     (lambda (k)
        (if k x y))))
```

Při její aplikaci bude vytvořeno prostředí $\mathcal{P}$, ve kterém budou existovat vazby na symboly `x` a `y`. Toto prostředí pak bude prostředím vzniku procedury $\langle$`(k)`, `(if k x y)`, $\mathcal{P}\rangle$. Výsledkem aplikace procedury `cons` tedy bude procedura, která v závislosti na jejím jednom argumentu `k` bude vracet buďto hodnotu navázanou na symbol `x` nebo `y`. Přitom `x` a `y` nejsou mezi formálními argumenty vrácené procedury, ta má pouze formální argument `k`. To jest vazby symbolů `x` a `y` se budou při vyhodnocování těla vrácené procedury hledat v nadřazeném prostředí a to je právě prostředí jejího vzniku, tedy $\mathcal{P}$. Při aplikaci této procedury s různými pravdivostními hodnotami dostáváme jednotlivé složky páru.

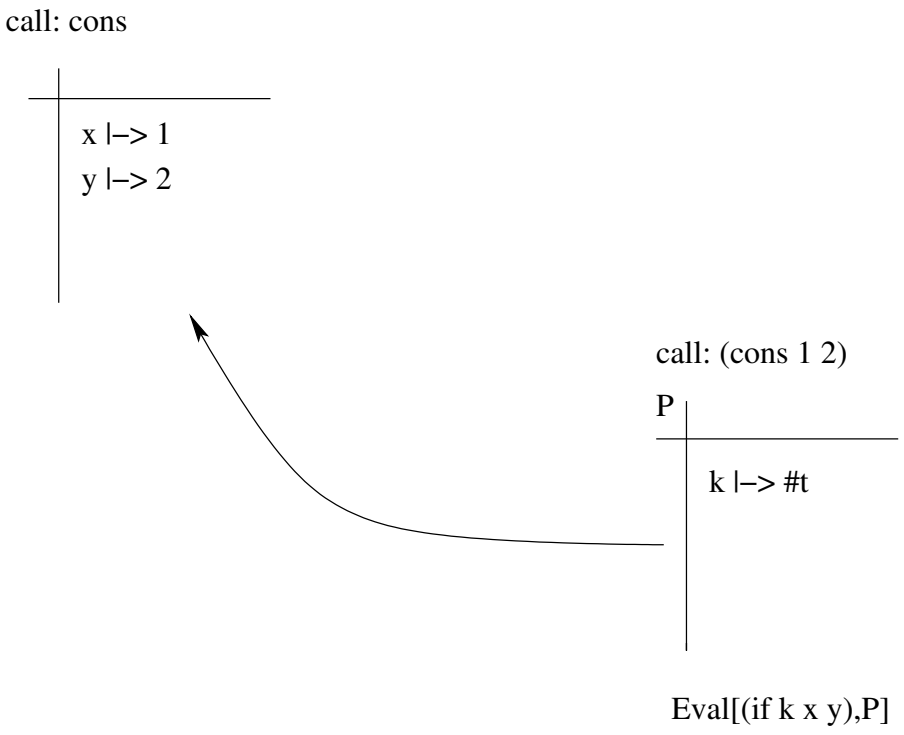
```
((cons 1 2) #t)   ⟼   1
((cons 1 2) #f)   ⟼   2
```

Selektory `car` a `cdr` zavolají takto vzniklou proceduru s různým argumentem. Jde vlastně o totéž jako v programu 4.4. Přehledně je to znázorněno na obrázku 4.5.

```
(define car (lambda (p) (p #t)))
(define cdr (lambda (p) (p #f)))

(define p (cons 2 3))
p   ⟼   „procedura“

(car p)   ⟼   2
(cdr p)   ⟼   3
```

Můžeme udělat ještě jinou reimplementaci, která je z hlediska funkcionálního programovaní čistější. Provedeme vlastně totéž, ale *s použitím procedur projekce*. Z lekce 2 známe procedury projekce, vracející jeden z jejich

Obrázek 4.5. Prostředí vznikající při použití vlastní implementace párů (NACRT OBRAZKU)



argumentů. Pomocí těchto procedur můžeme jednoduše implementovat konstruktor a selektory párů, viz program 4.3. Výsledkem aplikace procedury `cons` bude procedura, v jejímž prostředí vzniku budou na

Program 4.3. Implementace tečkových párů pomocí procedur vyšších řádů.

```
(define cons
  (lambda (x y)
     (lambda (proj)
       (proj x y))))

(define 1-z-2 (lambda (x y) x))
(define 2-z-2 (lambda (x y) y))

(define car (lambda (p) (p 1-z-2)))
(define cdr (lambda (p) (p 2-z-2)))
```

symboly `x` a `y` navázané argumenty se kterými byla `cons` aplikována což jsou, stejně jako v předchozím případě, elementy z nichž chceme „vytvořit pár“. Tato procedura nebude brát jako argument pravdivostí hodnotu, nýbrž projekci. Selektory `car` a `cdr` pak opětně ve svém těle volají tuto proceduru s různými projekcemi.

Opět se můžeme vrátit k abstrakčním bariérám založeným na datech. V předchozí podkapitole jsme uvedli, že bariéry založené na datové abstrakci jsou de facto obecnější než bariéry založené na procedurální abstrakci. Když si nyní uvědomíme, že hierarchická data lze plně vyjádřit pouze pomocí procedur vyšších řádů, ihned dostaneme, že bariéry založené na datové abstrakci jsou „stejně silné“ jako bariéry založené na procedurální abstrakci. Podotkněme, že toto pozorování vlastně říká, že nemá příliš smysl

odlišovat od sebe procedurální a datovou abstrakci – můžeme se bez újmy bavit pouze o jediné abstrakci (datové = procedurální). V drtivé většině programovacích jazyků si však takový „luxus“ dovolit nemůžeme, protože v nich nelze s programy (respektive s procedurami) zacházet jako s daty ani tak nelze činit obráceně. Právě popsaný pohled na datovou a procedurální abstrakci je silným rysem řady funkcionálních jazyků, především pak dialektů LISPu k nimž patří i jazyk Scheme.

4.5 Symbolická data a kvotování

Doposud jsme všechna složená data, která jsme používali, konstruovali až na výjimky pomocí párů z čísel. Na čísla se lze dívat jako na *primitivní*, nedělitelná neboli *nehierarchická data.* V této sekci rozšíříme vyjadřovací sílu našeho jazyka uvedením možnosti pracovat s libovolnými *symboly* jako s daty. Nejprve řekněme, k čemu by nám to mohlo být dobré. Symboly slouží při programování jako „jména“ a v programech je leckdy potřeba pracovat se jmény jako s daty. Pomocí symboly můžeme například označovat některé části hierarchických datových struktur (uvidíme v dalších lekcích) nebo mohou sloužit přímo jako zpracovávaná data. Například symboly `Praha`, `New-York`, `Prostejov` a další můžeme chápat jako „jména měst“, symboly `red`, `green`, `blue`, ... můžeme v programech chápat jako symbolická označení barev a tak dále.

Se symboy můžeme pracovat jako s daty díky speciální formě `quote`, která zabraňuje vyhodnocování svého (jediného) argumentu.

Definice 4.16 (speciální forma `quote`). Speciální forma `quote` se používá s jedním argumentem

`(quote` $\langle arg \rangle$`)`.

Výsledkem aplikace této speciální formy je pak $\langle arg \rangle$. Jinými slovy, speciální forma `quote` vrací svůj argument v nevyhodnocené podobě. Přesněji, označíme-li Q speciální formu navázanou na `quote`, pak

$$\mathrm{Apply}(Q, \langle arg \rangle) := \langle arg \rangle.$$ ■

Zde máme příklady aplikace speciální formy `quote`.

```
(quote 10)       ⟼  10
(quote blah)     ⟼  blah
(quote (a . b))  ⟼  (a . b)
```

Poznámka 4.17. (a) Je naprosto zřejmé, že na symbol `quote` musí být navázána speciální forma. Kdyby to byla procedura, muselo by dojít k vyhodnocení jejího argumentu, jak vyplývá z definice 1.21.

(b) Všimněte si, že je zbytečné kvotovat čísla. Stejně tak je zbytečné kvotovat jakýkoli jiný element, který se vyhodnocuje sám na sebe. V takovém případě se totiž vyhodnocený element rovná nevyhodnocenému. Konkrétně třeba výraz `(quote 10)` se vyhodnotí na číslo `10`, ale číslo `10` se vyhodnotí samo na sebe – tedy na číslo `10`.

(c) Slovo „quote“ znamená v angličtině „uvozovka“. Použití kvotování je analogické s použitím uvozovek v přirozeném jazyce. Porovnejte například věty: (i) Zobraz x; a (ii) Zobraz „x“.

Pro zkrácení zápisu je ve Scheme možné namísto `(quote` $\langle arg \rangle$`)` psát jen `'`$\langle arg \rangle$. Tedy můžeme psát:

```
'blah     ⟼  blah
'10       ⟼  10
'(a . b)  ⟼  (a . b)
```

Poznámka 4.18. Apostrof `'` je takzvaný *syntaktický cukr* pro `quote` a neměli bychom jej chápat ani jako samostatný symbol ani jako rozšíření syntaxe jazyka Scheme[5]. Z našeho pohledu je to v podstatě jen

[5]Při konstrukci konkrétního interpretu jazyka Scheme je však potřeba tento symbol načítat a brát jej v potaz. Reader musí být uzpůsoben k tomu, aby mohl zkrácené výrazy nahrazovat seznamy s `quote`. Takže někdo by v tuto chvíli mohl namítat, že apostrof *je* součástí syntaxe jazyka Scheme. U konkrétních interpretů tomu tak je. Z pohledu abstraktního interpretu Scheme však budeme apostrof uvažovat „mimo jazyk“ a tím pádem nebudeme muset opět rozšiřovat definici symbolických výrazů.

metasyntaktická značka (je „nad“ syntaxí jazyka Scheme), kterou zkracujeme delší výraz. *Syntaktický cukr* je *terminus technicus* užívající se pro rozšíření zápisu programu, které jej dělají snadnější („sladší“) pro použití člověkem. Syntaktický cukr dává člověku alternativní způsob psaní kódu, který je praktičtější tím, že je stručnější nebo tím, že je podobný nějaké známé notaci. V případě jednoduché uvozovky ve Scheme vlastně jde o obojí. Zápis je stručnější a podobný použití uvozovek v přirozeném jazyce.

Pozor! Je opět nutné rozlišovat symbol samotný a element navázaný na symbol. Jde o dvě zcela odlišné věci, jak již dobře víme z předchozích lekcí. Nyní se tento kontrast ale zvětšuje, protože symboly jsou pro nás nejen jedny ze symbolických výrazů, ale díky `quote` již může pracovat se symboly jako s elementy jazyka. Například v následujícím kódu navážeme na symbol `plus` různé elementy – jednou proceduru navázanou na symbol `+`:

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

a podruhé, s použitím kvotování, symbol `+`:

```
(define plus '+)
(plus 1 2)  ⟼  „CHYBA: + není procedura ani speciální forma.“
plus        ⟼  +
```

Příklad 4.19. Pomocí `define` můžeme zavést symboly, které se vyhodnocují na sebe sama:

```
(define blah 'blah)
blah  ⟼  blah
```

4.6 Implementace racionální aritmetiky

V této sekci naimplementujeme vlastní racionální aritmetiku. Jedná se o komplexní příklad demonstrující práci s páry, datovou abstrakcí a abstrakčními bariérami.

Nejprve vytvoříme konstruktor `make-r` a selektory `numer` a `denom`. První verze konstruktoru bude vypadat jednoduše – prostě ze dvou argumentů představujících jmenovatele a čitatele vytvoříme tečkový pár.

```
(define make-r (lambda (x y) (cons x y)))
```

Podle toho pak také budou vypadat selektory: `numer` bude vracet první prvek tohoto páru a `denom` jeho druhý prvek.

```
(define numer (lambda (x) (car x)))
(define denom (lambda (x) (cdr x)))
```

Nyní naimplementujeme procedury pro základní aritmetické operace s racionálními čísly: procedury `r+` a `r-` pro sčítání a odčítání. Další procedury `r*` a `r/` budou součástí úkolů na konci lekce. Procedury jsou přímým přepisem předpisu

$$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm bc}{bd}$$

Všimněte si, že v kódu už nebudeme používat procedury `cons`, `car` a `cdr`, protože se nacházíme za abstrakční bariérou a nepracujeme již s páry (s konkrétní reprezentací), ale s racionálními čísly (s abstrakcí).

```
(define r+
  (lambda (x y)
    (make-r (+ (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y)))))

(define r-
```

```
  (lambda (x y)
    (make-r (- (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y)))))
```

Též můžeme naimplementovat celou řadu predikátů. Protože by se jejich kódy téměř nelišily, vytvoříme je pomocí procedury vyššího řádu:

```
(define make-rac-pred
  (lambda (p?)
    (lambda (a b)
      (p? (* (numer a) (denom b))
          (* (numer b) (denom a))))))
```

Touto procedurou vytvoříme vlastní predikáty na porovnávání racionálních čísel:

```
(define r< (make-rac-pred <))
(define r> (make-rac-pred >))
(define r= (make-rac-pred =))
(define r<= (make-rac-pred <=))
(define r>= (make-rac-pred >=))
```

```
(r< (make-r 1 2) (make-r 2 3))  ⟹  #t
(r> (make-r 1 2) (make-r 2 3))  ⟹  #f
```

Dále naprogramujeme procedury `r-max` a `r-min` jako analogie procedur `max` a `min` z lekce 2. Můžeme k tomu směle využít proceduru vyššího řádu `extrem`. Tuto proceduru jsme definovali v programu 2.11 na straně 68.

```
(define r-max (extrem r>))
(define r-min (extrem r<))
```

Máme tedy procedury na výběr většího či menšího racionálního čísla:

```
(r-max (make-r 1 2) (make-r 2 3))  ⟹  (2 . 3)
(r-min (make-r 1 2) (make-r 2 3))  ⟹  (1 . 2)
```

Nyní uděláme změnu v konstruktoru racionálního čísla, tak, aby zlomek při vytváření automaticky krátil. Tedy čitatele i jmenovatele vydělí jejich největším společným jmenovatelem. Zbytek kódu (ostatní procedury) se nezmění.

```
(define make-r
  (lambda (x y)
    (let ((g (gcd x y)))
      (cons (/ x g) (/ y g)))))
```

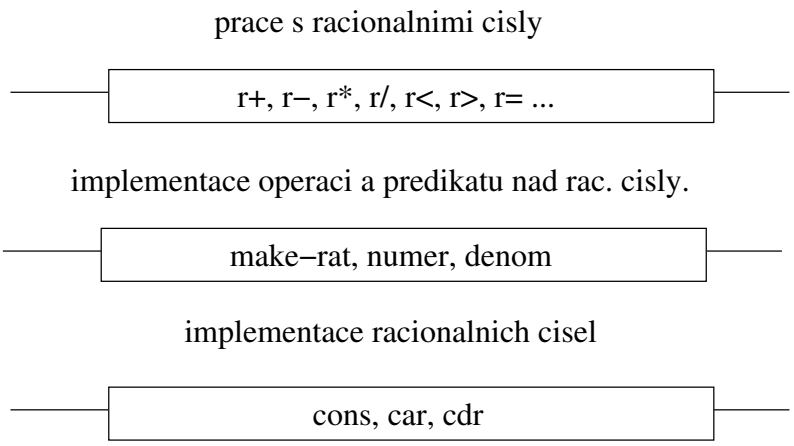
Teď provedeme reimplementaci `cons`, `car` a `cdr` tak, jak je to v programu 4.3. Dále pro vypisování doděláme proceduru `r->number`. To proto, že po reimpelemtaci budou racionální čísla procedury, jejichž externí reprezentace nám neprozradí jejich složení.

```
(define r->number
  (lambda (x)
    (/ (numer x) (denom x))))
```

Všimněte si, že zbytek programu bude fungovat, aniž bychom jej museli měnit. Modifikovali jsme nejspodnější vrstvu programu (viz obrázek 4.6) bez nutnosti měnit vrstvy, které jsou nad ní. Ve vyšších vrstvách je změna neznatelná. Například procedury `r-max` a `r-min` budou pracovat bez jakékoli změny:

```
(r->number (r-max (make-r 1 2) (make-r 2 3)))  ⟹  2/3
(r->number (r-min (make-r 1 2) (make-r 2 3)))  ⟹  1/2
```

Obrázek 4.6. Vrstvy v implementaci racionální aritmetiky



A teď ještě jednou změníme procedury navázané na symboly `cons`, `car` a `cdr`. Tentokrát tak, že zaměníme pořadí prvků v páru. Takže například výraz `(cons 1 2)` se nevyhodnotí na pár `(1 . 2)`, ale na pár `(2 . 1)`. Tedy opět změníme nejspodnější vrstvu programu.

```
(define kons cons)
(define kar car)
(define kdr cdr)

(define cons (lambda (x y) (kons y x)))

(define car (lambda (p) (kdr p)))
(define cdr (lambda (p) (kar p)))
```

Díky dodržování abstrakčních bariér se však tato změna neprojeví ve vyšších vrstvách:

```
(r->number (r-max (make-r 1 2) (make-r 2 3)))  ⟹  2/3
(r->number (r-min (make-r 1 2) (make-r 2 3)))  ⟹  1/2
```

Shrnutí

V této lekci jsme se zabývali vytvářením abstrakcí pomocí dat. Představili jsme si tečkové páry, pomocí nichž vytváříme hierarchické datové struktury. Rozšířili jsme symbolické výrazy o tečkové páry a doplnili Eval. Ukázali jsme rovněž, že tečkové páry bychom mohli plně definovat pouze pomocí procedur vyšších řádů. Dále jsme se zabývali kvotováním a jeho zkrácenou notací, jenž nám umožní chápat symboly jako data. Jako příklad datové abstrakce jsme uvedli implementaci racionální aritmetiky.

Pojmy k zapamatování

- datová abstrakce
- konstruktor
- kvotování
- selektor
- symbolická data
- tečkové páry

- syntaktický cukr
- konkrétní datová reprezentace

Nově představené prvky jazyka Scheme

- procedury `cons`, `car`, `cdr`
- speciální forma `quote`

Kontrolní otázky

1. *Co je datová abstrakce?*
2. *Co jsou páry?*
3. *Jak vypadá externí reprezentace párů?*
4. *Co je boxová notace?*
5. *Jak jsme změnili definici S-výrazu?*
6. *Co je to kvotování?*
7. *Jaká je zkrácená notace kvotování?*
8. *Je možné naimplementovat páry pomocí procedur vyšších řádů? Jak?*

Cvičení

1. Napište bez použití interpretru vyhodnocení následujících výrazů:

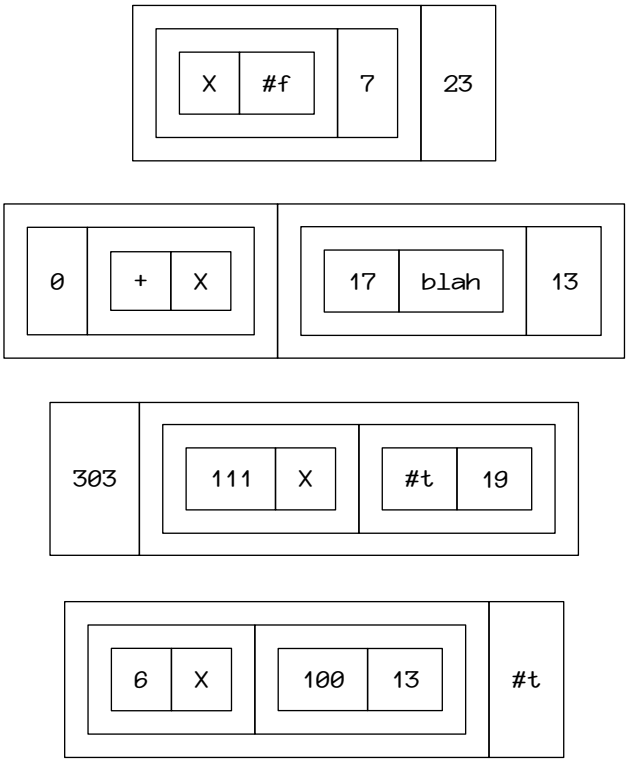
```
cons                                ⟼
(cons 1 2)                          ⟼
(cons car cdr)                      ⟼
(cdr (cons cdr cdr))                ⟼
'cons                               ⟼

(cons '(cons . cons) car)           ⟼
(cons lambda 'x)                    ⟼
('+ 1 2 3 4)                        ⟼
'10                                 ⟼
'(20 . 40)                          ⟼

(caar (cons (cons 1 2) 3))          ⟼
(cadr '((1 . 2) . 4))               ⟼
(car 'cons)                         ⟼
(cdr 1 2)                           ⟼
(define 'a 10)                      ⟼

(if 'symbol 10 20)                  ⟼
((car (cons and or)))               ⟼
```

2. Mějme hierarchická data znázorněná v boxové notaci na obrázku 4.7. Napište výrazy, jejichž vyhodnocením vzniknou.
3. Napište páry z obrázku 4.7 v tečkové notaci.
4. Napište sekvenci procedur `car` a `cdr` (respektive složeniny), kterými se v hierarchických strukturách z obrázku 4.7 dostaneme k prvku `X`.
5. Napište proceduru `map-pair`, která bere jako argumenty proceduru o jednom argumentu ⟨*proc*⟩ a pár ⟨*pár*⟩ a vrací pár výsledků aplikací procedury ⟨*proc*⟩ na prvky páru ⟨*pár*⟩. Příklady aplikace:

```
(map-pair - '(1 . 2))                          ⟼  (-1 . -2)
(map-pair (lambda (op) (op 3)) (cons + -))     ⟼  (3 . -3)
```

6. Upravte konstruktor racionálních čísel tak, aby jmenovatel byl vždy přirozené číslo. Například:

Obrázek 4.7. Boxová notace tečkových párů – zadání ke cvičení




```
(define r (make-r 3 -4))
(numer r) ⟹ -3                    (denom r) ⟹ 4
```
nikoli
```
(numer r) ⟹ 3                     (denom r) ⟹ -4
```

7. Napište procedury `r*` a `r/` na násobení a dělení racionálních čísel.

Úkoly k textu

1. Podobným způsobem jako jsme naimplementovali páry pomocí procedur vyšších řádů (tedy bez použití procedur `cons`, `car`, `cdr`) naimplementujte uspořádané trojice.

Řešení ke cvičením

1. procedura cons, (1 . 2), (procedura car . procedura cdr), procedura cdr, cons, ((cons . cons) procedura car), (specialni forma lambda . x), chyba, 10, (20 . 40) 1, chyba, chyba, chyba, chyba 10, #t
2.
```
(cons (cons (cons 'X #f) 7) 23)

(cons
 (cons 0 (cons '+ 'X))
 (cons (cons 17 'blah) 13))

(cons 303
      (cons (cons 111 'X)
            (cons #t 19)))

(cons (cons (cons 6 'X)
```

```
              (cons 100 13))
        #t)
```

3. ```
   (((X . #f) . 7) . 23)
   ((0 + . X) (17 . blah) . 13)
   (303 (111 . X) #t . 19)
   (((6 . X) 100 . 13) . #t)
   ```

4. `caaar`, `cddar`, `cdadr`, `cdaar`

5. ```
   (define map-pair
     (lambda (f p)
       (cons (f (car p)) (f (cdr p)))))
   ```

6. ```
   (define make-rac
     (lambda (n d)
       (let ((div ((if (< d 0) - +) (gcd n d))))
         (cons (/ n div) (/ d div)))))
   ```

7. ```
   (define r*                                  (define r/
     (lambda (a b)                               (lambda (a b)
       (make-r (* (numer a) (numer b))                     (make-r (* (numer a) (denom b))
               (* (denom a) (denom b))))))                         (* (denom a) (numer b)))))
   ```