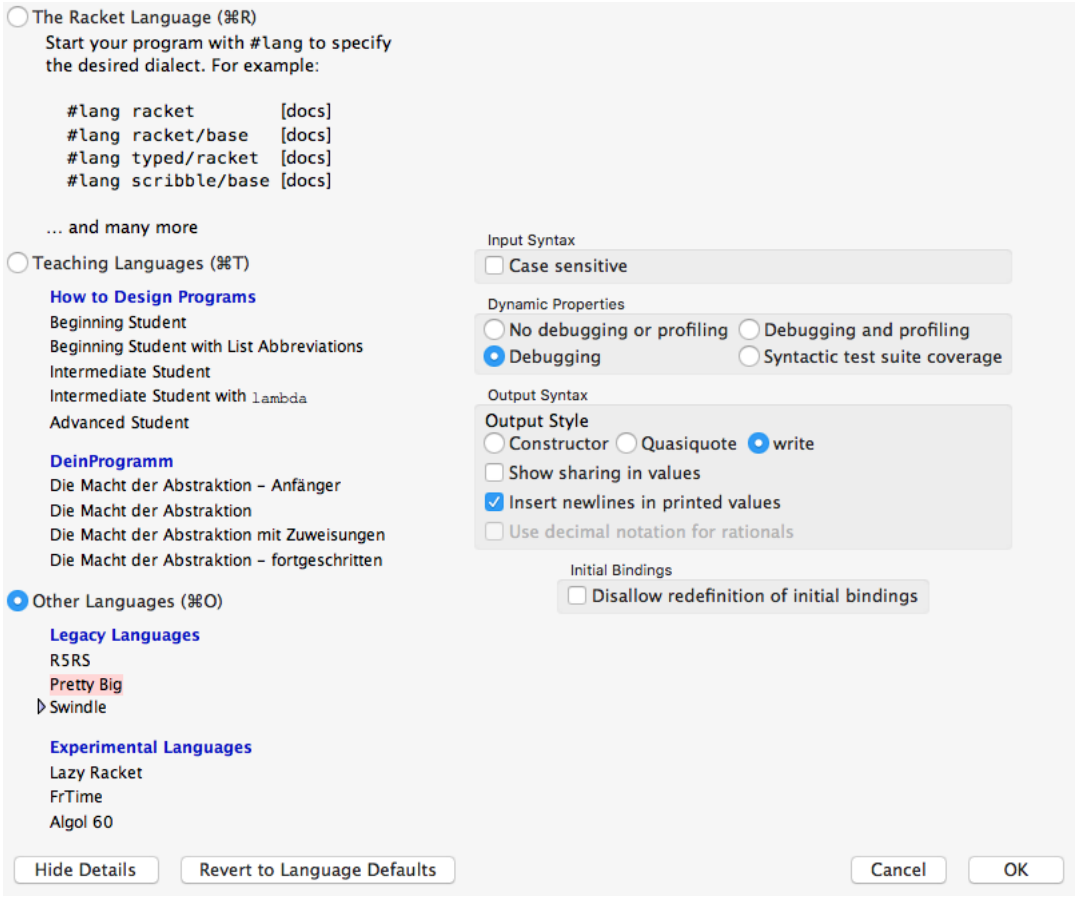
KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 1 ◇ poznámky k přednášce

8. Páry a seznamy

Než začnete, změňte si nastavení jazyka v DrRacket podle následujícího obrázku. Bez něj by nefungovalo, co budeme dnes a v budoucnu programovat. Nastavení funguje i na předchozí přednášky.



Hlavní změny způsobené novým nastavením jsou

1. Čísla psaná s desetinnou tečkou jsou automaticky nepřesná, ostatní jsou přesná. Není nutné používat notaci `#i`.
2. Pravdivostní hodnoty `#true` a `#false` lze zadávat a tisknou se zkráceně jako `#t` a `#f`.
3. Definice v příkazovém řádku lze opakovat. To je dobré k výukovým a testovacím účelům, v reálném programu by se definice opakovat neměly.
4. A především: procedury `cons`, `car` a `cdr`, které budeme používat dále, budou fungovat tak, jak potřebujeme.

1 Datové struktury a datová abstrakce

Procedury, které jsme dosud programovali, pracovaly s jednoduchými daty: čísly. V programech je ale potřeba používat i data složená. Hlavním důvodem je opět potřeba abstrakce, tentokrát **datové abstrakce**.

Uveďme si příklad (vychází ze starší úlohy na cvičení). Při psaní jakékoliv procedury, která pracuje s geometrickými body, musíme zatím body zadávat pomocí dvou parametrů — jejich první a druhé souřadnice:

```
(define (point-distance A-x A-y B-x B-y)
  (sqrt (+ (sqr (- A-x B-x))
           (sqr (- A-y B-y)))))
```

Pokud chceme vypočítat vzdálenost dvou bodů, musíme je tedy zadat po souřadnicích:

```
> (point-distance 2 -1 5 3)
5
```

Použití procedury by bylo jistě jednodušší, kdyby umožňovala místo souřadnic bodů zadávat přímo body. Kdyby například proměnné `A` a `B` obsahovaly dva body (ať už vytvořené jakkoli), napsali bychom prostě

```
> (point-distance A B)
5
```

Toto a další vlastnosti bodů, které ještě uvedeme, znamená, že chceme, aby body byly **elementy prvního řádu** tak, jak jsme o nich hovořili v jedné z minulých přednášek.

Současně by body byly *datovými strukturami*, protože by se skládaly z více jednoduchých hodnot (obsahovaly by informaci o dvou číslech: souřadnicích x a y).

Aniž si zatím řekneme, jak přesně by body byly implementovány, ukážeme si, jak by se s nimi dalo pracovat.

Nový bod bychom vytvořili pomocí procedury `point`:

```
> (define A (point 2 -1))
> (define B (point 5 3))
```

Jednotlivé souřadnice bodů bychom zjišťovali pomocí procedur `point-x` a `point-y`:

```
> (point-x A)
2
> (point-y B)
3
```

Proceduře `point` říkáme *konstruktor* bodu, procedurám `point-x` a `point-y` *selektory*.

Pomocí uvedených procedur můžeme přepsat proceduru `point-distance` tak, aby pracovala s našimi body:

```
(define (point-distance A B)
  (sqrt (+ (sqr (- (point-x A) (point-x B)))
           (sqr (- (point-y A) (point-y B))))))
```

a otestovat ji:

```
> (point-distance A B)
5
> (point-distance (point 2 5) A)
6
```

Díky konstruktoru `point` můžeme psát procedury, které vracejí nové body. Například následující procedura vrací bod ve středu úsečky dané koncovými body:

```
(define (segment-center A B)
  (point (/ (+ (point-x A) (point-x B)) 2)
         (/ (+ (point-y A) (point-y B)) 2)))
```

Test:

```
> (point-x (segment-center A B))
3½
```

A můžeme také napsat proceduru, která daný bod hezky vytiskne:

```
(define (display-point pt)
  (begin (display "[")
         (display (point-x pt))
         (display ",")
         (display (point-y pt))
         (display "]")))
```

Například vyhodnocení výrazu `(display-point (segment-center A B))` (pokud proměnné `A` a `B` stále obsahují hodnoty, které jsme do nich před chvílí uložili) pak vytiskne `[7/2,1]`.

Všimněme si, že k tomu, abychom s body mohli pracovat, nepotřebujeme vědět, jak jsou konstruktor `point` a selektory `point-x` a `point-y` napsané. To je důležitý moment, který je dán tím, že používáme datovou abstrakci. Nezajímá nás, *co* to bod je, stačí, že víme, *jak* se s ním pracuje. To nám

1. zjednodušuje práci (nemusíme se starat o to, jak jsou body reprezentovány, tedy o implementační detaily),
2. umožňuje v budoucnu v případě potřeby snadno přejít k jiné reprezentaci bodů,
3. zvyšuje čitelnost kódu (ukážeme za chvíli).

Samozřejmě je ale nutné, aby někdo vhodnou reprezentaci bodů navrhl a procedury `point`, `point-x` a `point-y` naprogramoval. Je možná trochu překvapující, že s našimi dosavadními znalostmi je možné to udělat.

Body lze reprezentovat pomocí lexikálních uzávěrů, o kterých jsme mluvili na minulé přednášce. Každý uzávěr totiž obsahuje informaci o prostředí, ve kterém vznikl. A do tohoto prostředí můžeme informaci o souřadnicích uložit. Samotný uzávěr by pak byla procedura, která na základě svého parametru hodnotu příslušného symbolu vrátí:

```
(define (point x y)
  (lambda (selector)
    (if selector x y)))

(define (point-x pt)
  (pt #true))

(define (point-y pt)
  (pt #false))
```

2 Pár jako nejjednodušší datová struktura

I když je možné použít k reprezentaci bodů lexikální uzávěry, Scheme poskytuje jednodušší a přímočařejší řešení: tzv. tečkové páry.

Definition 1 (Tečkový pár)**.** *Tečkový pár* (stručně *pár*) je datová struktura, která se skládá ze dvou složek, nazývaných (z historických důvodů) *car* a *cdr*. Tečkové páry se zapisují do kulatých závorek, složky jsou odděleny tečkou.

K vytvoření nového páru slouží procedura `cons`:

```
> (cons 1 2)
(1 . 2)
```

Aplikace procedury `cons` na dva argumenty vrátí jako výsledek tečkový pár se složkou *car* rovnou prvnímu a složkou `cdr` druhému argumentu. Procedura `cons` je tedy konstruktorem tečkových párů.

Ke zjištění složek párů slouží procedury `car` a `cdr`:

```
> (car (cons 3 4))
3
> (cdr (cons 3 4))
4
```

Jsou to tedy selektory tečkových párů.

Poznamenejme, že ve funkcionálním programování jednou vytvořené datové struktury už neměníme. Proto nepotřebujeme *mutátory* tečkových párů, tedy procedury, které by existujícímu páru mohly změnit jeho složky.

Predikát `pair?` zjišťuje, zda je daná hodnota pár:

```
> (pair? 1)
#f
> (pair? (cons 1 1))
#t
```

Tečkové páry se hodí jako reprezentace bodů:

```
(define (point x y)
  (cons x y))

(define (point-x pt)
  (car pt))

(define (point-y pt)
  (cdr pt))
```

Když takto upravíme konstruktor a selektory bodů, budou všechny naše procedury, které pracují s body (tedy `point-distance`, `segment-center` a `display-point`, ale samozřejmě i libovolné další) fungovat stejně jako předtím, aniž bychom je museli měnit. To je druhá výhoda datové abstrakce, o které jsme si řekli dříve.

Jako další příklad si ukážeme, jak lze tečkové páry použít k reprezentaci zlomků. Zlomek (*fraction*) se skládá z čitatele (*numerator*) a jmenovatele (*denominator*).

Ty uložíme do složek *car* a *cdr* tečkového páru. Konstruktor a selektory mohou tedy vypadat takto:

```
(define (fraction n d)
  (cons n d))

(define (numer frac)
  (car frac))

(define (denom frac)
  (cdr frac))
```

Procedura na tisk:

```
(define (display-frac frac)
  (begin (display (numer frac))
         (display "/")
         (display (denom frac))))
```

Procedury na základní operace se zlomky:

```
(define (frac-+ x y)
  (fraction (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (frac-* x y)
  (fraction (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

Porovnávání zlomků: zlomky nemusí být zkrácené, takže nestačí porovnat čitatel a jmenovatel jednoho s čitatelem a jmenovatelem druhého. Můžeme si ale všimnout, že $\frac{a}{b} = \frac{c}{d}$ je totéž jako $ad = cb$.

```
(define (frac-equal? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

Vraťme se teď znovu k výhodám datové abstrakce: datová abstrakce

1. zjednodušuje práci (nemusíme se starat o implementační detaily),
2. umožňuje v budoucnu v případě potřeby snadno přejít k jiné reprezentaci,
3. zvyšuje čitelnost kódu.

Následující varianta procedury `frac-*` nepoužívá datovou abstrakci, ale jinak dělá přesně totéž co procedura původní:

```
(define (frac-* x y)
  (cons (* (car x) (cdr y))
        (* (car y) (cdr x))))
```

Můžeme vidět, že procedura postrádá všechny tři výhody datové abstrakce:

1. Abychom ji mohli napsat, museli jsme vědět, že zlomky jsou reprezentovány tečkovými páry. V původní verzi jsme to vědět nemuseli. (Původní verzi jsme mohli dokonce klidně napsat ještě před tím, než jsme se o konkrétní reprezentaci zlomků rozhodli! Tak jsme to udělali na začátku s procedurami pracujícími s body.)
2. Pokud bychom se v budoucnu rozhodli reprezentovat zlomky jinak (šlo by to například pomocí lexikálních uzávěrů, jak jsme si ukázali dříve), museli bychom proceduru přepsat. Původní verzi ne.
3. Zdrojový kód procedury není dobře čitelný. Není například hned jasné, co znamená `(car x)`. Vidíme sice, že jde o složku *car* páru `x`, ale nevidíme, jaký má význam. V původní verzi uvedené `(numer x)` nám jasně říká, že jde o čitatel zlomku.

Abychom měli zlomky vždy v základním tvaru (zkrácené), upravíme proceduru `fraction`, aby před vytvořením zlomku zadaný čitatel a jmenovatel zkrátila:

```
(define (fraction n d)
  (let ((div (gcd n d)))
    (cons (/ n div) (/ d div))))
```

(Procedura `gcd` počítá největší společný dělitel zadaných dvou čísel. Ve Scheme je k dispozici jako vestavěná procedura, umíme ji ale taky sami naprogramovat.)

Procedura `frac-equal?` se pak zjednoduší:

```
(define (frac-equal? x y)
  (and (= (numer x) (numer y))
       (= (denom x) (denom y))))
```

3 Páry jako základ složitějších datových struktur

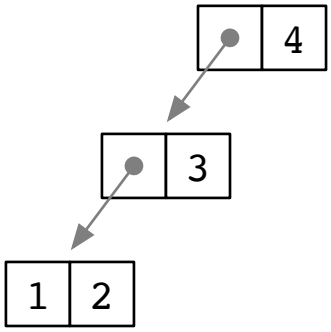
Páry mohou ve svých složkách *car* a *cdr* obsahovat i jiná data než čísla:

```
> (cons #true "Dobrý den")
(#t . "Dobrý den")
```

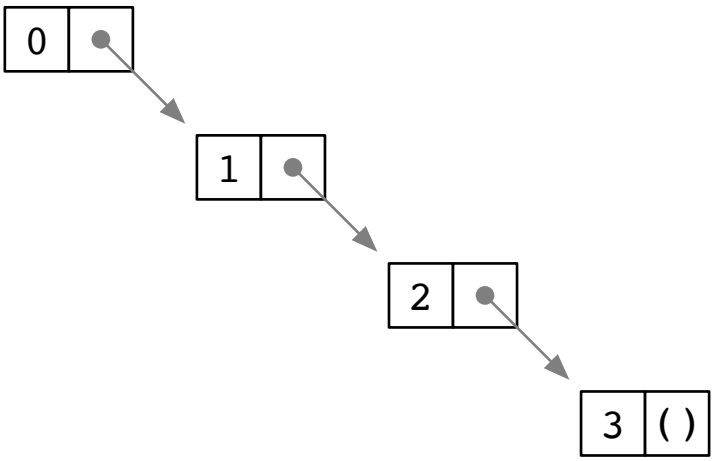
Mohou dokonce obsahovat i jiné páry:

```
> (cons (cons 1 2) 3)
((1 . 2) . 3)
> (cons (cons (cons 1 2) 3) 4)
(((1 . 2) . 3) . 4)
```

To může být zprvu těžko pochopitelné (do krabice se nevejdou dvě jiné krabice stejné velikosti). Můžeme si ale představit, že to, co do složek páru ukládáme, může být jen informace o tom, kde se obsah složky nachází (adresu). Tu si můžeme představit třeba jako šipku. Poslední uvedený pár si tedy můžeme představit takto:



Ve Scheme hrají zásadní roli páry uspořádané následujícím způsobem:



Jsou to páry, v jejichž `cdr` je vždy uložen jiný pár nebo zvláštní hodnota `()`. Tato hodnota se nazývá *prázdný seznam* a v rámci vyhodnocovacího procesu se vyhodnocuje sama na sebe:

```
> ()
()
```

Strukturu z posledního obrázku můžeme tedy vytvořit vyhodnocením výrazu

```
(cons 0 (cons 1 (cons 2 (cons 3 ()))))
```

Takovým strukturám se říká seznamy. Přesně řečeno je seznam definován takto:

Definition 2 (Seznam)**.** *Seznam* je buď prázdný seznam, nebo tečkový pár, jehož *cdr* je opět seznam.

Seznamy se ve Scheme zapisují tak, jak jsme zvyklí. Proto výše uvedený seznam Scheme vytiskne takto:

```
> (cons 0 (cons 1 (cons 2 (cons 3 ()))))
(0 1 2 3)
```

Jde ovšem jen o jeden z několika možných zápisů. Další je například `(0 . (1 . (2 . (3 .())))).`

Podle definice seznamu je *cdr* daného neprázdného seznamu vždy seznam:

```
> (cdr (cons 0 (cons 1 (cons 2 (cons 3 ())))))
(1 2 3)
```

Uvedený seznam lze tedy zapsat například i takto: `(0 . (1 2 3))` nebo takto: `(0 . (1 . (2 3))).`

Hodnoty uložené ve složkách *car* párů v seznamu se nazývají *prvky seznamu*. Počet prvků seznamu (a tedy počet párů, které seznam tvoří) se nazývá *délka* seznamu. Následující procedura počítá délku zadaného seznamu:

```
(define (length lst)
  (if (eq? lst ())
      0
      (+ 1 (length (cdr lst)))))
```

Test:

```
> (length ())
0
> (length (cons 1 (cons 2 ())))
2
```

Tato procedura vytváří seznam dané délky s daným prvkem:

```
(define (make-list len elem)
  (if (= len 0)
      ()
      (cons elem (make-list (- len 1) elem))))
```

Test:

```
> (make-list 30 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> (make-list 20 #false)
(#f #f #f #f #f #f #f #f #f #f #f #f #f #f #f #f #f #f #f #f)
> (make-list 20 ())
(() () () () () () () () () () () () () () () () () () () ())
```

Otázky a úkoly na cvičení

1. Je možné zavést body pomocí následujícího konstruktoru a selektorů?

   ```
   (define (point x y)
     (lambda (s)
       (s x y)))

   (define (point-x pt)
     (pt (lambda (x y) x)))

   (define (point-y pt)
     (pt (lambda (x y) y)))
   ```

   Zdůvodněte.

2. Napište predikát `right-triangle?`, který (podobně jako predikát z druhého cvičení) zjistí, zda zadaný trojúhelník je pravoúhlý. Predikát bude akceptovat jako argumenty vrcholy trojúhelníka jako body.

3. Napište proceduru `op-vertex`, která k bodům A a B najde bod C tak, že bod B je středem úsečky s vrcholy A a C.

4. Napište procedury na rozdíl a podíl zlomků.

5. Vylepšete procedury pro práci se zlomky tak, aby správně pracovaly i se zápornými hodnotami. (Ani zdaleka není nutné upravovat všechny.)

6. Napište predikát `list?`, který zjistí, zda jeho argument je seznam.

7. Napište proceduru `make-ar-seq-list`, která vytvoří seznam členů aritmetické posloupnosti se zadaným počátkem, koncem a diferencí:

```
> (make-ar-seq-list 10 20 2)
(10 12 14 16 18 20)
```

8. Napište proceduru `make-geom-seq-list` která vytvoří seznam členů geometrické posloupnosti s daným prvním členem, kvocientem a počtem členů:

```
> (make-geom-seq-list 5 2 10)
(5 10 20 40 80 160 320 640 1280 2560)
```

9. Napište proceduru `build-list` která vytvoří seznam zadané délky n vzniklý aplikací zadané procedury na čísla $0, 1, 2, \cdots, n-1$:

```
> (build-list 10 sqr)
(0 1 4 9 16 25 36 49 64 81)
> (build-list 20 (lambda (n) 0))
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> (build-list 20 identity)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)
> (build-list 10 (lambda (n) (cons n n)))
((0 . 0) (1 . 1) (2 . 2) (3 . 3) (4 . 4) (5 . 5) (6 . 6) (7 .
7) (8 . 8) (9 . 9))
```

10. Napište proceduru `draw-list` pro želví grafiku, která bude jako argument akceptovat seznam, jehož prvky budou páry. *Car* každého páru znamená délku kreslené čáry, *cdr* úhel otočení želvy. Procedura postupně projde celý seznam a na základě údajů v každém páru nejprve nakreslí čáru a pak otočí želvu. Příklad: vyhodnocení výrazu

```
(draw-list (make-list 4 (cons 100 90)))
```

by mělo vést k nakreslení čtverce o délce hrany 100.