Lokální definice (1)

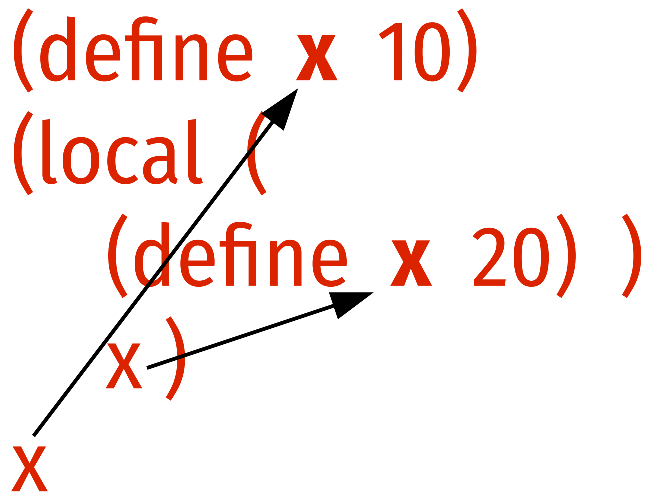
- syntaxe:
  (local (*seznam definic*) *výraz*)
- definice jsou dostupné pouze uvnitř příkazu local
- příklad:

```
(local (
   (define Pi 3.1415926)
   (define (plocha-kruhu r) (* Pi r r)) )
   (plocha-kruhu 10) )
Pi
```

Pi – běhová chyba, neznámé Pi

Lokální definice (2)

- libovolná definice může být lokální – konstanty, funkce, datové struktury,...
- pro rozsahy platnosti platí obvyklá pravidla – lokální definice zastiňuje definici přicházející zvenčí



Použití lokálních definic

- obvyklé využití: local je tělem funkce, zavádí lokální hodnoty nebo pomocné funkce
- příklad:

```
(define (plocha-kruhu r)
  (local (
    (define Pi 3.1415926) )
    (* Pi r r)))
```

- tento příklad je hloupý – proč?

Lokální pomocná funkce

spojit dva seznamy, vybírat vždy menší abs. hodnotu čísel

```
(define (absmin-seznam L1 L2)
  (local (
    (define (absmin X Y)
      (min (abs X) (abs Y))
    )
    (cond
      [(or (empty? L1) (empty? L2)) '()]
      [else (cons (absmin (car L1) (car L2))
                  (absmin-seznam (cdr L1) (cdr L2)))]
    )))
```

Problém podpůrné funkce

- předchozí příklad funguje, ale definuje pomocnou funkci opakovaně
- každé rekurzivní volání znamená novou definici **absmin** (její obsah se ale nijak nemění)
- trik – předefinovat sama sebe, použít local pro:
  - definice všech podpůrných funkcí
  - redefinice sama sebe (skutečné tělo)
  - tělem bude volání sama sebe (volá se předefinovaná verze)

Pomocná funkce znovu a lépe

```
(define (absmin-seznam L1 L2)
  (local (
    (define (absmin X Y)
      (min (abs X) (abs Y))
    )
    (define (absmin-seznam L1 L2)
          (cond
            [(or (empty? L1) (empty? L2) '()]
            [else (cons (absmin (car L1) (car L2))
               (absmin-seznam (cdr L1) (cdr L2)))]
          )) )
    (absmin-seznam L1 L2)))
```

pomocná funkce

redefinice

Řešení pro obalové funkce

```
(define (nejmensi L)
  (local (
    (define (nejmensi kandidat zbytek)
      (cond
        [(empty? zbytek) kandidat]
        [else (nejmensi (min kandidat (car zbytek))
                        (cdr zbytek))]
    )) )
  (nejmensi (car L) (cdr L))))
```

redefinice

- určení hodnoty shora dolů nepotřebuje obalovou funkci, lze definovat lokálně

Zvyšování efektivity (1)

- průměr z pozitivních čísel, první pokus

```
(define (vyber-kladne L)
  (cond
    [(empty? L) '()]
    [(>= 0 (car L)) (vyber-kladne (cdr L))]
    [else (cons (car L) (vyber-kladne (cdr L)))]
  ))
(define (secti L)
  (cond
    [(empty? L) 0]
    [else (+ (car L) (secti (cdr L)))]
  ))
```

Zvyšování efektivity (2)

```
(define (prumer-kladnych L)
  (cond
    [(empty? (vyber-kladne L)) #f]
    [else (/ (secti (vyber-kladne L))
             (length (vyber-kladne L)))]
  ))
```

- vyber-kladne se volá třikrát se stejným parametrem → stejným výsledkem
- rekurzivní funkce – každé volání dá spoustu práce

Zvyšování efektivity (3)

- s výhodou lze využít lokální definici
- uložíme výsledek vyber-kladne a používáme

```
(define (prumer-kladnych L)
  (local ( (define kladne (vyber-kladne L)) )
    (cond
      [(empty? kladne) #f]
      [else (/ (secti kladne)
               (length kladne))]
    )
  ))
```

Krok stranou

- ještě lepší řešení – zavést obecnou funkci pro průměr seznamu (lze pak využít i jinde):

```
(define (prumer L)
   (cond
      [(empty? L) #f]
      [else (/ (secti L) (length L))]
   ))
(define (prumer-kladnych L)
   (prumer (vyber-kladne L)))
```

Lokální definice – shrnutí

- skrývá lokální konstrukce
- vyhýbá se opakovaným výpočtům a definicím (redefinice sama sebe)
- poskytuje přirozený mechanismus pro ukládání (průběžných) výsledků
- zvyšuje čitelnost kódu díky vhodně zvoleným identifikátorům
- stojí za zamyšlení, zda nenahradit voláním funkce s odpovídajícím parametrem

Rozsah platnosti identifikátorů

- **oblast:** část programu, kde identifikátor existuje
  - **globální identifikátory:** celý program
  - **identifikátory definované v local:** uvnitř local
  - **parametry funkcí:** tělo funkce
- **rozsah platnosti:** část programu, v níž výskyt identifikátoru znamená danou konkrétní definici
  - oblast bez vnořených oblastí stejnojmenných identifikátorů
  - lokální definice zastiňuje definici přicházející zvenčí

Příklad

```
(define x 20)
...
(local (
   (define x 10)
   (define (mocnina4 x) (* x x x x)) )
          )
   (mocnina x)
)
```

Funkcionální abstrakce

- opakování je častým zdrojem chyb
- je lépe se vyhnout vytváření podobných funkcí
- příklad: vyhledávání

```
(define (hledejcislo? N L)
   (cond  [(empty? L) #f]
          [(= N (car L)) #t]
          [else (hledejcislo? N (cdr L))]
   ))
```

různé typy hledání se liší jen zde
např string=? pro řetězce

Funkcionální parametry

- funkce může být parametrem jiné funkce
- zachází se s ní zcela normálně, jen se ve výrazech používá na prvním místě jako název funkce
- příklad: funkce, které předáme funkci dvou argumentů a dvě hodnoty, vydá výsledek dané funkce pro tyto hodnoty

```
(define (zavolej funkce x y)
  (funkce x y))
(zavolej + 10 36) ⟶ 46
```

Obecné hledání

- předejme kritérium jako argument

```
(define (hledej? kriterium? N L)
  (cond [(empty? L) #f]
        [(kriterium? N (car L)) #t]
        [else (hledej? kriterium? N (cdr L))]
  ))
```

- použití:

```
(hledej? = 10 SeznamCisel)
(hledej? string=? "Jan Veselý" SeznamJmen)
```

Použití obecného hledání

- volání je složitější – může se hodit si navíc definovat specializované funkce (pro čísla, řetězce,...), ale jako konkretizace obecné funkce:

```
(define (hledejcislo? N L)
   (hledej? = N L))
```

- sdílejí **společný vyhledávací mechanismus** → mají konzistentní chování
- lze předat i uživatelem definovanou funkci (např. porovnávající dvě datové struktury)

absmin-seznam ještě jinak

spojit dva seznamy, vybírat vždy menší abs. hodnotu čísel

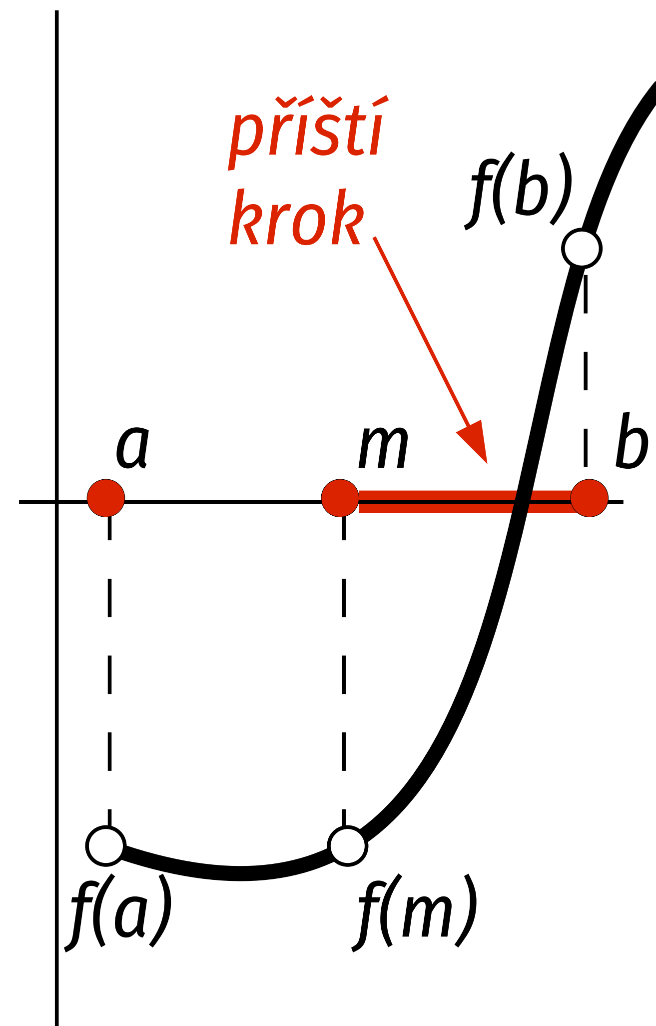
```
(define (kombinuj-seznamy fce L1 L2)
   (cond
      [(or (empty? L1) (empty? L2)) '()]
      [else (cons (fce (car L1) (car L2))
                  (kombinuj-seznamy fce (cdr L1) (cdr L2)))]
   ))

(define (absmin X Y) (min (abs X) (abs Y)))

(define (absmin-seznam L1 L2)
   (kombinuj-seznamy absmin L1 L2))
```

Hledání kořene f(x)=0

- metoda půlení intervalů
- parametry
  - spojitá funkce 1 proměnné f(x)
  - čísla a (levý okraj intervalu) a b (pravý) taková, že f(a) a f(b) mají opačná znaménka
- rozpůlíme interval a pokračujeme polovinou, kde jsou v krajních bodech opačná znaménka



Hledání kořene – kód

```
(define PRESNOST 0.001)

(define (hledej-koren f levy pravy)
    (local
        ((define stred (/ (+ levy pravy) 2)))
     (cond
        [(> PRESNOST (abs (f stred))) stred]
        [(> 0 (* (f levy) (f stred)))
                (hledej-koren f levy stred)]
        [else   (hledej-koren f stred pravy)]
    )))
```

Filtrování seznamu

- obvyklá úloha: vybrat ze seznamu prvky vyhovující určité podmínce

```
(define (filtr podminka? L)
   (cond     [(empty? L) '()]
             [(podminka? (car L))
                (cons (car L) (filtr podminka? (cdr L)))]
             [else (filtr podminka? (cdr L))]
   ))
```

- použití: vybrat pozitivní hodnoty

```
(filtr positive? SeznamCisel)
```

Výhody abstrakce

- **flexibilita**
  - abstraktní funkce lze použít k řešení řady podobných úloh
  - program je kratší a snadněji pochopitelný
- **konzistence**
  - ve všech případech se chovají konzistentně (minimalizováno riziko překlepů)
  - změny/vylepšení se promítnou do chování všech
- abstrakce je lepší než kopírování + změny

Generické/polymorfní funkce

- naše filtrovací funkce zpracuje seznam libovolného typu (i směs typů) – vše závisí na podmínce
- výsledkem je seznam stejného typu získaný výběrem těch hodnot, které vyhověly podmínce
- funkce, které **jednotně zpracovávají různé datové typy**, jsou označovány jako generické nebo polymorfní

Rozšířené popisy

- komentář popisující typy argumentů a výsledku funkce potřebuje rozšíření
- seznam hodnot určitého typu:
  ; hledejcislo?: číslo **(listof číslo)** -> boolean
- funkcionální argumenty:
  ; hledej?: **(X Y -> boolean)** X (listof Y) -> boolean
  zároveň obsahuje polymorfismus – konkrétní typy zastupují písmena, stejné písmeno znamená nutnost stejného typu

Zkoumání možností (1)

- někdy lze generickou funkci použít i v nečekaných situacích
- příklad: zpracováváme seznam s cílem získat určitou hodnotu (např. sečíst prvky seznamu)
  - pro prázdný seznam použijeme nějakou základní hodnotu
  - jinak zkombinujeme první prvek seznamu s výsledkem rekurzivního volání sama sebe na zbytek seznamu

Zkoumání možností (2)

```
(define (fold zaklad kombinuj L)
  (cond  [(empty? L) zaklad]
         [else (kombinuj
                 (car L)
                 (fold zaklad kombinuj (cdr L)))]
  ))
```

- (define (**soucet** L) (fold 0 + L))
- (define (**soucin** L) (fold 1 * L))
- (define (**???** L1 L2) (fold L2 cons L1))

Zkoumání možností (3)

- ; vložení čísla do vzestupně uspořádaného seznamu

```
(define (insert N L)
  (cond
    [(empty? L) (list N)]
    [(<= N (car L)) (cons N L)]
    [else (cons (car L) (insert N (cdr L)))]
  ))
```

- (define (**insertsort** L) (fold '() insert L))
- generickou funkci lze mnohdy využít i v netušených případech

vytvořeno s podporou
projektu ESF