

PARADIGMATA PROGRAMOVÁNÍ 2A

MAKRA I



VÝVOJ TOHOTO UČEBNÍHO MATERIÁLU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČR

Opakování (kvazikvotování)

``(1 2 3 4 5)` \Rightarrow `(1 2 3 4 5)`
``(1 (+ 2 3) 4 5)` \Rightarrow `(1 (+ 2 3) 4 5)`
``(1 ,(+ 2 3) 4 5)` \Rightarrow `(1 5 4 5)`
``(1 ((,(+ 2 3))) 4 5)` \Rightarrow `(1 ((5)) 4 5)`

`(define s `(a b c))`

``(1 ,s 2)` \Rightarrow `(1 (a b c) 2)`
``(1 ,@s 2)` \Rightarrow `(1 a b c 2)`

``(1 '2 3)` \Rightarrow `(1 (quote 2) 3)`
```(1 2 3)`  $\Rightarrow$  `(quote (1 2 3))`  
````(1 2 3)`  $\Rightarrow$  `(quasiquote (1 2 3))`

``(1 `(2 3))` \Rightarrow `(1 (quasiquote (2 3)))`
``(1 `',(+ 1 2) 3))` \Rightarrow `(1 (quasiquote ((unquote (+ 1 2)) 3)))`
``(1 ,`',(+ 1 2) 3))` \Rightarrow `(1 (3 3))`

Problém: chceme upravit `if` tak,
aby při absenci alternativního výrazu vracel `#f`

nyní máme:

`(if (= 1 2) 'blah)` \implies nedefinovaná hodnota

chceme:

`(new-if (= 1 2) 'blah)` \implies `#f`

nabízí se vyřešit pomocí nové procedury:

```
(define new-if  
  (lambda (elem1 elem2)  
    (if elem1 elem2 #f)))
```

při volání `new-if` je vždy vyhodnocen i druhý argument:

`(new-if #f blah-blah)` \implies Error

Potřebujeme: během vyhodnocování každý výraz tvaru

```
(new-if expr1 expr2)
```

nahradit výrazem:

```
(if expr1 expr2 #f)
```

bez toho aniž by se vyhodnocovaly *expr1* a *expr2*

Jinými slovy:

- potřebujeme zavést předpis, který bude provádět „transformaci kódu“
- **transformace** ... konkrétní část kódu je nahrazena jinou
- po transformaci proběhne **vyhodnocení transformovaného kódu**

```
(define x 11)
```

```
(new-if (even? x) (+ x 1))
```

↓ transformace

```
(if (even? x) (+ x 1) #f)
```

↓ vyhodnocení

```
#f
```

Jak se dívat na transformaci?

- můžeme si ji představit jako (klasickou) proceduru, které jsou předány argumenty v nevyhodnocené podobě
- transformace se v některých jazycích nazývá **makroexpenze**

;; transformační procedura pro `new-if`

```
(define new-if-trans  
  (lambda (test expr . alt)  
    (list 'if test expr  
          (if (null? alt)  
              #f  
              (car alt))))))
```

`(new-if-trans 'e1 'e2 'e3)` \Rightarrow `(if e1 e2 e3)`

`(new-if-trans 'e1 'e2)` \Rightarrow `(if e1 e2 #f)`

`(new-if-trans '(even? x) '(+ x 1))`

\Rightarrow `(if (even? x) (+ x 1) #f)`

kratší řešení pomocí kvazikvotování

```
(define new-if-trans  
  (lambda (test expr . alt)  
    `(if ,test  
        ,expr  
        (begin #f ,@alt))))
```

příklady transformace:

```
(new-if-trans 'expr1 'expr2 'expr3)  
⇒ (if expr1 expr2 (begin #f expr3))
```

```
(new-if-trans 'expr1 'expr2)  
⇒ (if expr1 expr2 (begin #f))
```

```
(new-if-trans '(even? x) '(+ x 1))  
⇒ (if (even? x) (+ x 1) (begin #f))
```

nouzové řešení `new-if`, které se již chová jak má
manuální spuštění transformační procedury
následné vyhodnocení transformovaného výrazu
`(eval (new-if-trans '(even? x) '(+ x 1)))`

Výhody řešení:

- pokud je první výraz nepravdivý, alternativní výraz není vyhodnocen
- touto konstrukcí (`new-if`) lze zastavit rekurzi

Nevýhody řešení:

- všechny předávané argumenty musíme explicitně kvotovat
- transformovaný výraz musíme ručně vyhodnotit pomocí `eval`
- `eval` ve většině interpretů pracuje jen v globálním prostředí
- volání je nepřehledné

problém s lexikálními vazbami

```
(let ((x 10))  
  (eval (new-if-transformer '(even? x) '(+ x 1))))  
⇒ error: x not bound
```

částečné řešení: použití (*the-environment*)
ve většině interpretů nebude fungovat

```
(let ((y 10))  
  (eval (new-if-transformer '(even? y) '(+ y 1))  
        (the-environment)))
```

navíc jsme se nezbavili nepřehledného kódu

Řešení problému: zavedení **maker**

MAKRA – dva základní pohledy na makra

1. POHLED: Makra jsou „rozšířením syntaxe jazyka“

- **makro** = dáno definicí svého transformačního předpisu
- po načtení výrazu (READ) je v něm provedena makroexpanze tuto fázi provádí tzv. *preprocesor*
- až po dokončení expanze všech maker nastává vyhodnocování výrazu
- nemá smysl uvažovat pojem „aplikace makra“
- takto na makra pohlíží většina PJ: C, DrScheme, Common LISP,...

Výhody přístupu:

- preprocesor a vlastní *eval* jsou zcela nezávislé
- preprocesor může být aktivován okamžitě po načtení výrazu
- umožňuje snadnou kompilaci kódu
(v kompilovaném kódu již pochopitelně „žádná makra nejsou“)

Nevýhody přístupu:

- makra jsou „mimo jazyk“ (často se zapisují odlišně, třeba v C)
- makra nejsou elementy prvního řádu

MAKRA – dva základní pohledy na makra

2. POHLED: Makra jsou „speciální elementy jazyka“

- **makro** = element jazyka obsahující ukazatel na transf. proceduru
- **transformační procedura** ... klasická procedura
- je potřeba rozšířit *eval*:
 - případ, kdy se první prvek seznamu vyhodnotí na makro
- makra jsou „uživatelsky definované speciální formy“
- takto na makra budeme pohlížet my (dále třeba PJ: M4, T_EX)

Výhody přístupu:

- makra jsou elementy prvního řádu
- s makry lze pracovat „jako s daty“,
 - mohou dynamicky vznikat/zanikat za běhu programu
- můžeme uvažovat koncept „anonymního makra“

Nevýhody přístupu:

- k makroexpansi dochází až při činnosti *eval*
- prakticky znemožňuje účinnou kompilaci kódu
- při neuváženém používání maker komplikuje ladění programu

Motivační příklady definice maker

```
(define-macro new-if
  (lambda (test expr . alt)
    (list 'if test expr
          (if (null? alt)
              #f
              (car alt))))))
```

```
(define-macro new-if
  (lambda (<test> <expr> . <alt>)
    (list 'if <test> <expr>
          (if (null? <alt>)
              #f
              (car <alt>))))))
```

Motivační příklady definice maker

;; new-if: pomocí kvazikvotování

```
(define-macro new-if
  (lambda (test expr . alt)
    `(if ,test
        ,expr
        (begin #f ,@alt))))
```

;; new-if: pomocí kvazikvotování

```
(define-macro new-if
  (lambda (<test> <expr> . <alt>)
    `(if ,<test>
        ,<expr>
        (begin #f ,@<alt>))))
```

Příklad použití makra

;; *new-if*: pomocí kvazikvotování

```
(define-macro new-if  
  (lambda (<test> <expr> . <alt>)  
    `(if ,<test>  
        ,<expr>  
        (begin #f ,@<alt>))))
```

```
(let ((x 10))
```

```
  (new-if (even? x) (+ x 1)))
```

↓ aktivace transformační procedury makra

```
(if (even? x) (+ x 1) (begin #f))
```

↓ vyhodnocení výrazu v prostředí, kde x má vazbu 10

```
11
```

Rozšíření EVAL

$\text{Eval}[E, \mathcal{P}]$:

- (A) Pokud je E **cislo**, ... jako obvykle
- (B) Pokud je E **symbol**, ... jako obvykle
- (C) Pokud je E **seznam** tvaru $(E_1 E_2 \dots E_n)$, pak nejprve provedeme vyhodnocení prvního prvku E_1 v prostředí \mathcal{P} a výslednou hodnotu označíme F_1 , to jest $F_1 := \text{Eval}[E_1, \mathcal{P}]$. Mohou nastat čtyři situace:
 - (C.1) Pokud F_1 je **procedura**, ... jako obvykle
 - (C.2) Pokud F_1 je **speciální forma**, ... jako obvykle
 - (C.3) Pokud F_1 je **makro** jehož transformační procedura je T , pak
 - ① $F' := \text{Apply}[T, E_2, \dots, E_n]$
(F' je výsledkem aplikace transf. procedury na nevyhodnocené arg.)
 - ② Výsledek vyhodnocení F elementu E v prostředí \mathcal{P} je definován
 $F := \text{Eval}[F', \mathcal{P}]$
(F je výsledek vyhodnocení elementu F' v prostředí \mathcal{P}).
 - (C.e) Pokud F_1 není **procedura**, **speciální forma**, ani **makro**, pak vyhodnocení končí chybou „CHYBA: První prvek seznamu ...“.

Ladění maker: základní princip

- potlačíme vyhodnocení transformovaného kódu
- využívá dodatečné KVOTOVÁNÍ

```
(define-macro new-if
  (lambda (test expr . alt)
    (list 'quote
          (list 'if test expr
                (if (null? alt) #f (car alt))))))
```

`(new-if #f blah-blah)` \implies `(if #f blah-blah #f)`

```
(define-macro new-if
  (lambda (test expr . alt)
    `(if ,test ,expr (begin #f ,@alt))))
```

`(new-if #f blah-blah)` \implies `(if #f blah-blah (begin #f))`

chceme vytvořit `and2` dvou argumentů vracející `#t` nebo `#f`
chceme vytvořit pouze s pomocí `if`

;; nedostačující řešení pomocí procedury:

```
(define and2
  (lambda (elem1 elem2)
    (if elem1
        (if elem2
            #t
            #f)
        #f)))
```

předchozí má vážný nedostatek:

`(and2 #f blah-blah)` \Rightarrow error (chceme `#f`)

`and2` se dvěma argumenty vracející konjunkci

potřebujeme: během vyhodnocování každý výraz

```
(and2 expr1 expr2)
```

nahradit výrazem

```
(if expr1 (if expr2 #t #f) #f)
```

bez toho aniž by se vyhodnocovaly `expr1` a `expr2`

Řešení pomocí makra:

```
(define-macro and2
  (lambda (expr1 expr2)
    `(if ,expr1
        (if ,expr2
            #t
            #f)
        #f)))
```

Ukzky pouit `and2`

`(and2 1 (+ 1 2))`

⇓

`(if 1 (if (+ 1 2) #t #f) #f)`

⇓

`#t`

`(and2 #f blah-blah)`

⇓

`(if #f (if blah-blah #t #f) #f)`

⇓

`#f`

`(and2 #t #f)`

⇓

`(if #t (if #f #t #f) #f)`

⇒ `#f`

Anaforický `if`: `if*`

- `if*`, který pracuje stejně jako `if`, ale umožňuje v druhém a třetím výrazu používat symbol `$result`, který bude vždy navázaný na výsledek vyhodnocení prvního výrazu

praktické rozšíření, místo:

```
(if (member 'b '(a b c d))  
    (list 'nalezen (member 'b '(a b c d)))  
    'blah)
```

stačí napsat:

```
(if* (member 'b '(a b c d))  
     (list 'nalezen $result)  
     'blah)            $\implies$  (nalezen (b c d))
```

Řešení: během vyhodnocování každá výraz

```
(if* expr1 expr2 expr3)
```

potřebujeme nahradit výrazem:

```
(let (($result expr1))  
  (if $result expr2 expr3))
```

a to opět bez vyhodnocování `expr1` až `expr3`

`if*` jako makro

```
(define-macro if*  
  (lambda (test expr . alt)  
    `(let (($result ,test))  
      (if $result  
        ,expr  
        ,@alt))))
```

Ukázky použití `if*`

bez `$result` se chová jako normální `if`

```
(if* 1 2 3)
```

⇓

```
(let (($result 1)) (if $result 2 3))
```

⇓

2

příklad použití `$result`

```
(if* 1 $result 3)
```

⇓

```
(let (($result 1)) (if $result $result 3))
```

⇓

1

Složitější ukázka použití `if*`

```
(if* (member 'b '(a b c d))  
      (list 'nalezen $result)  
      'blah)
```



```
(let (($result (member (quote b)  
                        (quote (a b c d)))))  
  (if $result  
      (list (quote nalezen) $result)  
      (quote blah)))
```



```
(nalezen (b c d))
```

- Všimněte si: v expandovaném výrazu nejsou žádné ``',

if pomocí cond

```
(define-macro if
  (lambda (test expr alt)
    `(cond (,test ,expr)
           (else ,alt))))
```

if pomocí cond (bez nutnosti mít alternativní větev)

```
(define-macro if
  (lambda (test expr . alt)
    `(cond (,test ,expr)
           (else (begin #f ,@alt))))))
```

podobné jako předchozí, ale vrátíme nedefinovanou hodnotu

```
(define-macro if
  (lambda (test expr . alt)
    `(cond (,test ,expr)
           (else (begin (cond) ,@alt))))))
```

`cond` pomocí `if`

musíme přepsat jeden `cond`-výraz pomocí několika `if`ů

;; základní `cond` pomocí `if` (pomocí rekurzivního vnoření)

```
(define-macro cond
  (lambda (clist)
    (let dive-ifs ((clist clist))
      (if (null? clist)
          `(if #f #f)
          (if (equal? (caar clist) 'else)
              (cadar clist)
              `(if ,(caar clist)
                    ,(cadar clist)
                    ,(dive-ifs (cdr clist))))))))
```


Příklad použití

```
(cond ((= x 3) 'blah)
      ((> x 10) (+ 1 x))
      ((prop? x y) (list x y))
      (else (f 20)))
```



```
(if (= x 3)
    (quote blah)
    (if (> x 10)
        (+ 1 x)
        (if (prop? x y)
            (list x y)
            (f 20)))))
```



...

`cond` pomocí `if`

musíme přepsat jeden `cond`-výraz pomocí několika `if`ů

;; základní `cond` pomocí `if` (řešeno jako rekurzivní makro)

```
(define-macro cond
  (lambda (clist)
    (if (null? clist)
        `(if #f #f)
        (if (equal? (caar clist) 'else)
            (cadar clist)
            `(if ,(caar clist)
                  ,(cadar clist)
                  (cond ,@(cdr clist)))))))
```

Příklad použití

```
(cond ((= x 3) 'blah)
      ((> x 10) (+ 1 x))
      ((prop? x y) (list x y))
      (else (f 20)))
```



```
(if (= x 3)
    (quote blah)
    (cond ((> x 10) (+ 1 x))
          ((prop? x y) (list x y))
          (else (f 20))))
```



...

Rozšířená verze `define`

`define` jsme zatím používali pouze ve tvaru

```
(define symbol <vyraz>)
```

v R5RS Scheme je `define` zaveden taky ve tvaru

```
(define (symbol <argumenty>...) <vyrazy>...)
```

Příklad: faktoriál

```
(define (f n)
  (if (= n 1)
      1
      (* n (f (- n 1)))))
```

Příklad: nepovinné argumenty

```
(define (f x y . args)
  (list x y args))
```

Rozšířená verze `define`

Pokud by náš interpret nedisponoval rozšířeným `define`,
pak bychom jej mohli vyrobit jako makro:

```
(define-macro def
  (lambda (first . args)
    (if (symbol? first)
        `(define ,first ,@args)
        `(define ,(car first)
              (lambda ,(cdr first)
                ,@args)))))
```

Příklad použití:

```
(def (f n)
  (if (= n 1)
      1
      (* n (f (- n 1)))))

(f 6)  $\implies$  720
```

Konjunkce libovolně mnoha argumentů pomocí `if`

`;; základní verze`

```
(define-macro and
  (lambda (args)
    (if (null? args)
        #t
        `(if ,(car args)
              (and ,@(cdr args))
              #f)))))
```

`(and 1 2 3)` \implies `#t`

protože:

`(and 1 2 3)` \implies `(if 1 (and 2 3) #f)` \implies ...

`(and 2 3)` \implies `(if 2 (and 3) #f)` \implies ...

`(and 3)` \implies `(if 3 (and) #f)` \implies ...

`(and)` \implies `#t` \implies `#t`

Konjunkce libovolně mnoha argumentů pomocí `if`

`;;` zlepšená verze (zobecněné pravdivostní hodnoty)

```
(define-macro and
  (lambda (args)
    (if (null? args)
        #t
        (if (null? (cdr args))
            (car args)
            `(if ,(car args)
                  (and ,@(cdr args))
                  #f))))))
```

yní se již chová jako klasický `and`:

`(and)` \implies `#t`

`(and 1 2 3)` \implies `3`

`(and 1 #f 3)` \implies `#f`

Disjunkce libovolně mnoha argumentů pomocí `if`

`;; základní verze`

```
(define-macro or
  (lambda (args)
    (if (null? args)
        #f
        `(if ,(car args)
              #t
              (or ,@(cdr args))))))
```

Příklad použití:

<code>(or)</code>	\Rightarrow	<code>#f</code>
<code>(or 1 2 3)</code>	\Rightarrow	<code>#t</code> chtěli bychom 1
<code>(or #f 2 3)</code>	\Rightarrow	<code>#t</code> chtěli bychom 2

Disjunkce libovolně mnoha argumentů pomocí `if`

;; rozšířená verze

```
(define-macro or
  (lambda (args)
    (if (null? args)
        #f
        (if (null? (cdr args))
            (car args)
            `(if ,(car args)
                  ,(car args)
                  (or ,@(cdr args)))))))
```

Chová se (zdánlivě) v pořádku:

<code>(or)</code>	\implies	<code>#f</code>
<code>(or 1 2 3)</code>	\implies	<code>1</code>
<code>(or #f 2 3)</code>	\implies	<code>2</code>

Problémy s implementací disjunkce pomocí `if`

- naše implementace `or`: dvakrát vyhodnocuje pravdivé argumenty (kromě posledního)
- důsledek: nechová se jako klasický `or` pokud použijeme vedlejší efekt

Chování klasického `or`

```
(let ((x 0))  
  (or (begin (set! x (+ x 1))  
            x)  
      blah))  $\implies$  1
```

Chování našeho `or`:

```
(let ((x 0))  
  (or (begin (set! x (+ x 1))  
            x)  
      blah))  $\implies$  2
```

Problémy s implementací disjunkce pomocí `if`

;; pokus o řešení předchozího problému

```
(define-macro or
  (lambda (args)
    (if (null? args)
        #f
        (if (null? (cdr args))
            (car args)
            `(let ((result ,(car args)))
              (if result
                  result
                  (or ,@(cdr args))))))))
```

Nyní už jsme předchozí problém odstranili, ...

```
(let ((x 0))
  (or (begin (set! x (+ x 1)) x)
      blah))  $\Rightarrow$  1
```

... ale nový problém jsme vyrobili

Klasický `or`:

```
(let ((result 10))  
  (or #f result))  $\implies$  10
```

Náš `or`:

```
(let ((result 10))  
  (or #f result))  
  ↓  
(let ((result #f))  
  (if result result (or result)))  $\implies$  #f
```

- došlo k překrytí symbolu `result` symbolem stejného jména, který je používán uvnitř našeho makra
- tomuto efektu se říká **symbol capture** (**variable capture**)
- v další lekci ukážeme čisté řešení tohoto problému

Makro realizující `let` pomocí λ -výrazů

;; základní `let` umožňující vázat hodnoty

```
(define-macro let
  (lambda (assgn . body)
    `((lambda ,(map car assgn)
        (begin ,@body))
      ,@(map cadr assgn))))
```

```
(let ((x 10)
      (y (+ x 1)))
  (list x y))
```

⇓

```
((lambda (x y)
  (begin (list x y)))
  10 (+ x 1))  $\Rightarrow$  ...
```

Pojmenovaný `let`

;; makro rozlišuje pojmenovaný/nepojmenovaný `let`

```
(define-macro let
```

```
  (lambda args
```

```
    (if (symbol? (car args))
```

```
      ;; pojmenovaný let
```

```
      `((lambda ()
```

```
        (define ,(car args)
```

```
          (lambda ,(map car (cadr args))
```

```
            (begin ,@(cddr args))))
```

```
        ,(car args) ,@(map cadr (cadr args))))))
```

```
      ;; nepojmenovaný let
```

```
      `((lambda ,(map car (car args))
```

```
        (begin ,@(cdr args)))
```

```
        ,@(map cadr (car args))))))
```

let* pomocí rekurzivního vnoření

```
(define-macro let*  
  (lambda (assgn . body)  
    (define iter  
      (lambda (assgn)  
        (if (null? assgn)  
            `((lambda () ,@body))  
            `((lambda (,(caar assgn))  
                ,(iter (cdr assgn)))  
              ,(cadar assgn))))))  
    (iter assgn)))
```

```
(let* ((x 10) (y (+ x 1))) (list x y))  $\Rightarrow$   
((lambda (x)  
  ((lambda (y)  
    ((lambda () (list x y)))  
    (+ x 1)))  
  10))  $\Rightarrow$  ...
```

let* jako rekurzivní makro

```
(define-macro let*  
  (lambda (assgn . body)  
    (if (null? assgn)  
        `((lambda () ,@body))  
        `((lambda (,(caar assgn))  
            (let* ,(cdr assgn)  
              ,@body))  
          ,(cadar assgn))))))
```

let* jako rekurzivní makro využívající vytvořený let

```
(define-macro let*  
  (lambda (assgn . body)  
    (if (null? assgn)  
        `((lambda () ,@body))  
        `(let (,(car assgn))  
            (let* ,(cdr assgn)  
              ,@body))))))
```


Pokud použijeme místo `(lambda () ,@body)` formu `begin` takto:

```
(define-macro let*  
  (lambda (assgn . body)  
    (if (null? assgn)  
        `(begin ,@body)  
        `(let (,(car assgn))  
            (let* ,(cdr assgn)  
                ,@body))))))
```

tak bude mít náš `let*` jiný význam

například:

```
(let* () (define x 10))
```

by provedlo definici v globálním prostředí (!!)

letrec* **pomocí** set!

```
(define-macro letrec
  (lambda (assgn . body)
    `((lambda ,(map car assgn)
        ,@(map (lambda (i)
                  `(set! ,(car i) ,(cadr i)))
                assgn)
        ,@body)
      ,@(map (lambda (i) #f) assgn)))))
```

letrec* **pomocí** define

```
(define-macro letrec
  (lambda (assgn . body)
    `((lambda ()
        ,@(map (lambda (i)
                  `(define ,(car i) ,(cadr i)))
                assgn)
        ,@body))))
```

Makra pro vytváření rekurzivních procedur

Makro pro vytváření rekurzivních procedur bez `define` používáme princip y-kombinátoru

```
(define-macro procedure
  (lambda (args . body)
    `(lambda ,args
       ((lambda (y)
          (y y ,@args))
        (lambda (self* ,@args)
          ,@body))))))
```

makro realizující volání sebe sama

```
(define-macro self
  (lambda args
    `(self* self* ,@args)))
```

Příklad použití:

```
(let ((f (procedure (n)
  (if (= n 1)
      1
      (* n (self (- n 1)))))))
  (map f '(1 2 3 4 5 6 7)))
```

má jednu drobnou vadu na kráse:

`self` se uvnitř procedury nechová jako procedura:

```
(let ((f (procedure (x)
  (if (list? x)
      (apply + (map self x))
      1))))
  (f '(a ((b c) ((d))) ((e) f))))
⇒ Error (self není procedura)
```

Řešení předchozího problému

```
(define-macro procedure
  (lambda (args . body)
    `(lambda ,args
      ((lambda (y)
        (y y ,@args))
       (lambda (self ,@args)
        (let ((self (lambda ,args (self self ,@args))))
          ,@body))))))
```

Příklad:

```
(let ((f (procedure (x)
  (if (list? x)
      (apply + (map self x))
      1))))
  (f '(a ((b c) ((d))) ((e) f))))  $\implies$  6
```

Speciální forma `let-values`: naše “vylepšení `let`”

Příklad použití `let-values`:

```
(let ((seznam `("Vilem" 100 blah ,(+ 1 2))))  
  (let-values ((blah 10)  
               ((name value next comment) seznam)  
               ((v n c) (cdr seznam))  
               (x (+ 10 20)))  
    (list blah name value next comment v n c x)))  
⇒ (10 "Vilem" 100 blah 3 100 blah 3 30)
```

Implementace makra:

```
(define-macro let-values
  (lambda (assgn . body)
    `(apply
      (lambda
        ,(apply append
                 (map (lambda (x)
                        (if (list? (car x))
                            (car x)
                            (list (car x))))
                     assgn)))
      (begin ,@body)))
  (append ,@(map (lambda (x)
                    (if (list? (car x))
                        (cadr x)
                        `(list ,(cadr x))))
                assgn))))))
```

Speciální forma `letref`: naše “vylepšení `letrec`”

; ; forma umí obnovit hodnoty vazeb, pokud byly změněny, například:

```
(letref ((x 10)
        (f (lambda (n)
              (if (= n 0)
                  1
                  (* n (f (- n 1)))))))
  (y 100))
```

(display x) zobrazí: 10

(newline)

(set! x (f 20))

(display x) zobrazí: 2432902008176640000

(newline)

(refresh 'x) provede návrat k původní hodnotě

(display x) zobrazí: 10

(newline)

#f) \implies #f

Téměř správné řešení:

```
(define-macro letref
  (lambda (bindings . body)
    `((lambda ()
        ,@(map (lambda (b)
                  `(define ,(car b) ,(cadr b)))
                bindings)
      (define refresh
        (lambda (symbol)
          (cond ,@(map (lambda (x)
                        `((equal? symbol ',(car x))
                          (set! ,(car x) ,(cadr x)))))
                  bindings))))
      (begin ,@body))))))
```

V případě vedlejšího efektu se dostaneme do problému:

```
(letref ((i 0)
        (x (begin (display "VOLANA")
                    (set! i (+ i 1))
                    i))))
(display (list i x))  zobrazí: (1 1)
(set! x 'blah)
(display (list i x))  zobrazí: (1 blah)
(refresh 'x)          dojde k druhému vyhodnocení
(display (list i x))  zobrazí: (2 2)
#f)  $\implies$  #f
```

Předchozí vadu odstraníme přeprogramováním `refresh`

```
(define-macro letref
  (lambda (bindings . body)
    `((lambda ()
        ,@(map (lambda (b) `(define ,(car b) ,(cadr b)))
              bindings)
      (define refresh
        (let ((mem (list
                     ,@(map (lambda (x)
                           `(cons ',(car x) ,(car x)))
                         bindings))))
          (lambda (symbol)
            (cond ,@(map (lambda (x)
                          `(equal? symbol ',(car x))
                            (set! ,(car x)
                                (cdr (assoc ',(car x) mem))))
                        bindings))))
        (begin ,@body))))))
```