KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO, OLOMOUC

PARADIGMATA PROGRAMOVÁNÍ 2A
MAKRA II

Slajdy vytvořili Vilém Vychodil a Jan Konečný

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))  ⟹  ···
```

Pojmenovaný `let`

```
;; pojmenovaný let umožňující vázat hodnoty
(define-macro let
  (lambda (sym assgn . body)
    `((lambda ()
        (define ,syn
           (lambda ,(map car assgn)
             (begin ,@body)))
        (,sym ,@(map cadr assgn))))


(let func ((x 10)
           (y (+ x 1)))
  (list func x y))
  ⇓
((lambda ()
   (define func (lambda (x y)
                  (begin (list func x y))))
   (func 10 (+ x 1))))  ⟹  ...
```

Oba `lety` v jednom

```
;; 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))  ⟹
((lambda (x)
   ((lambda (y)
      ((lambda () (list x y))))
    (+ x 1)))
 10)  ⟹  ...
```

`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:

```
(procedure (n)
   (if (= n 1)
       1
       (* n (self (- n 1))))))
```

⇓

```
(lambda (n)
  ((lambda (y) (y y n))
     (lambda (self* n)
       (if (= n 1) 1
           (* n (self (- n 1)))))))
```

⇓ (ještě po expanzi `self`)

```
(lambda (n)
  ((lambda (y) (y y n))
     (lambda (self* n)
       (if (= n 1) 1
           (* n (self* self* (- n 1)))))))
```

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))))
```

$\longmapsto$ 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))))  ⟹  6
```

Řešení předchozího problému

Příklad:

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

⇓

```
(let ((f (lambda (x)
           ((lambda (y) (y y x))
              (lambda (self x)
                (let ((self (lambda (x) (self self x))))
                  (if (list? x) (apply + (map self x))
                      1)))))))
  (f '(a ((b c) ((d))) ((e) f))))
```

Speciální forma `let-values`: naše "vylepšené `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)
```

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)))
```

⇓

```
(let ((seznam `("Vilem" 100 blah ,(+ 1 2))))
  (apply
   (lambda (blah name value next comment v n c x)
     (begin
       (list blah name value next comment v n c x)))
   (append
     (list 10)
     seznam
     (cdr seznam)
     (list (+ 10 20)))))
```

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)  ⟹  #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)))))
```

```
(letref ((x 10)
         (f (lambda (n)
              (if (= n 0) 1 (* n (f (- n 1))))))
         (y 100))
  (display x) (newline) (set! x (f 20))
  (display x) (newline) (refresh 'x)
  (display x) (newline) #f)  ⟹  #f
```

⇓

```
((lambda ()
   (define x 10)
   (define f (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))))
   (define y 100)
   (define refresh
     (lambda (symbol)
       (cond
        ((equal? symbol 'x) (set! x 10))
        ((equal? symbol 'f) (set! f (lambda (n) ...)))
        ((equal? symbol 'y) (set! y 100)))))
   ...
```

```
((lambda ()
   (define x 10)
   (define f (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))))
   (define y 100)
   (define refresh
     (lambda (symbol)
       (cond
        ((equal? symbol 'x) (set! x 10))
        ((equal? symbol 'f) (set! f (lambda (n) ...)))
        ((equal? symbol 'y) (set! y 100)))))
   (begin (display x) (newline)
          (set! x (f 20))
          (display x) (newline)
          (refresh 'x)
          (display x) (newline) #f)))
```

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)  ⟹  #f
```

Protože `refresh` vypada takto:

```
(define refresh
     (lambda (symbol)
       (cond ((equal? symbol 'i) (set! i 0))
             ((equal? symbol 'x)
               (set! x (begin (display "VOLANA")
                              (set! i (+ i 1)) i)))))))
```

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)))))
```

Expandovaný kód předchozí ukázky bude vypadat takto:

```
((lambda ()
   (define i 0)
   (define x (begin (display "VOLANA") (set! i (+ i 1)) i))
   (define refresh
     (let ((mem (list (cons 'i i) (cons 'x x))))
       (lambda (symbol)
         (cond ((equal? symbol 'i)
                   (set! i (cdr (assoc 'i mem))))
               ((equal? symbol 'x)
                   (set! x (cdr (assoc 'x mem)))))))))
   (begin (display (list i x)) (set! x 'blah)
          (display (list i x)) (refresh 'x)
          (display (list i x)) #f)))
```

Motivace: Chceme vyřešit problém se „symbol capture"

V následujícím makru dochází k zachyceni symbolu `curval`

```
(define-macro capture
  (lambda body
    `(let ((curval 100))
       ,@body)))
```

Příklad použití:

```
(let ((curval 10))
  (capture
   (display "Hodnota: ")
   (display curval)
   (newline)
   (+ curval 1)))  ⟹  101
```

Motivace: Chceme vyřešit problém se „symbol capture"

Důvod zachycení symbolu

```
(capture
 (display "Hodnota: ")
 (display curval)
 (newline)
 (+ curval 1))
```

⇓

```
(let ((curval 100))
  (display "Hodnota: ")
  (display curval)
  (newline)
  (+ curval 1))  ⟹  101
```

Předchozí problém lze čistě vyřešit *zavedením nového typu symbolů.*
Všechny symboly, které jsme doposud uvažovali byly tzv. **pojmenované**.

`'ahoj` $\longmapsto$ element *symbol*, který má jméno *ahoj*
`'blah` $\longmapsto$ element *symbol*, který má jméno *blah'*
`(define s 'ahoj)` na `s` se naváže `ahoj`
`s` $\longmapsto$ `ahoj` (to jest na `s` je jako hodnota navázaný symbol)

Porovnávání pojmenovaných symbolů probíhá vzhledem k jejich jménům.

```
(equal? 'ahoj 'blah)  ⟼  #f
(equal? 'ahoj 'ahoj)  ⟼  #t
(eq? 'ahoj 'blah)     ⟼  #f
(eq? 'ahoj 'ahoj)     ⟼  #t
```

Důvod: v prostředích se hledají vazby **podle jmen symbolů**,
nikoliv podle jejich fyzického uložení v paměti.

Nový typ symbolu: **bezejmenný (generovaný) symbol**:

- vzniká voláním procedury bez argumentu `gensym`,
- každý generovaný symbol je roven pouze sám sobě,
- nemá žádnou „čitelnou externí reprezentaci“.

```
(gensym)                          ⟹  nově vygenerovaný symbol
(symbol? (gensym))                ⟹  #t
(equal? (gensym) (gensym))        ⟹  #f

(define s (gensym))
(equal? s s)  ⟹  #t
s             ⟹  g3  (vypíše Dr. Scheme)
```

Poznámka: i kdyby interpret dva nově vygenerované symboly „vypisoval stejně“, nejedná se o týž symbol. (!)

Řešení motivačního problému

místo:

```
(define-macro capture
  (lambda body
    `(let ((curval 100))
       ,@body)))
```

napíšeme:

```
(define-macro no-capture
  (lambda body
    (let ((new-unnamed-symbol (gensym)))
      `(let ((,new-unnamed-symbol 100))
         ,@body))))
```

Na `new-unnamed-symbol` bude vázán nově vygenerovaný symbol.
Jelikož je tento symbol beze jména, nelze se na něj z `body` nijak dostat.

Příklad:

```
(let ((curval 10))
  (no-capture
   (display "Hodnota: ")
   (display curval)
   (newline)
   (+ curval 1)))
```

$\Downarrow$

```
(let ((„vygenerovaný symbol“ 100))
  (display "Hodnota: ")
  (display curval)
  (newline)
  (+ curval 1))  ⟼  11
```

Řešení problému s makrem realizujícím spec. formu „or“.

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

Nyní vypadá přepis takto:

```
(or 1 2 3)
```

$\Downarrow$

```
(let ((„symbol“ 1))
  (if „symbol“ „symbol“ (or 2 3)))  ⟼  ···
```

V pořádku (jednonásobné vyhodnocení):

```
(let ((x 0))
  (or (begin (set! x (+ x 1))
             x)
      blah))  ⟼  1
```

Rovněž v pořádku (nedochází k symbol capture):

```
(let ((result 10))
  (or #f result))  ⟼  10
```

Speciální forma case

Příklad použití:

```
(case (+ 1 2)
  ((0 1 2) 'blah)
  ((3 4) 'ahoj)
  (else 'nic))  ⟹  ahoj
```

```
;; naivní makro (má capture na result)
(define-macro case
  (lambda (value . clist)
    `(let ((result ,value))
       (cond ,@(map (lambda (x)
                      (if (list? (car x))
                          `((member result ',(car x))
                            ,(cadr x))
                          `(else ,(cadr x))))
                    clist)))))
```

Opět nefunguje

```
(let ((result 1000))
  (case 10
    ((10 20) result)
    (else #f)))
```

$\Longmapsto$ 10 místo 1000

protože `case` se expanduje takto:

```
(let ((result 10))
  (cond ((member result '(10 20)) result)
        (else #f)))
```

Řešení je opět jednoduché:

```
(define-macro case
  (lambda (value . clist)
    (let ((result (gensym)))
      `(let ((,result ,value))
         (cond ,@(map (lambda (x)
                        (if (list? (car x))
                            `((member ,result ',(car x))
                              ,(cadr x))
                            `(else ,(cadr x))))
                      clist))))))
```

Pak to bude vypadat takto:

```
(let ((„symbol“ 10))
  (cond ((member „symbol“ '(10 20)) result)
        (else #f)))
```

Speciální forma `cond` podle R6RS.
Některé možnosti `cond` jsme zatím zatajovali.

- víc argumentů v těle, prázdné tělo, klíčové slovo „`=>`“

```
(cond)                          ⟹   nedefinovaná hodnota

(cond (else 'blah))             ⟹   blah

(cond ('blah))                  ⟹   blah

(cond (10 => -))                ⟹   -10

(cond ((= 1 1)
       (display "X")
       (newline)
       (+ 1 2)))                ⟹   3 rovněž zobrazí X
```

Další příklad:

```
(define test
  (lambda (n)
    (cond ((= n 1) 'jedna)
          ((= n 2))
          ((= n 3) (display n)
                   (newline)
                   (+ n 1))
          ((and (> n 4) n) =>
           (lambda (x) (* x x)))
          (else 'nevim))))
```

```
(test 0)   ⟼  nevim
(test 1)   ⟼  jedna
(test 2)   ⟼  #t
(test 3)   ⟼  4 rovněž zobrazí 3
(test 10)  ⟼  100
```

Speciální forma `cond` podle R6RS

```
(define-macro cond
  (lambda clist
    (let ((symbol (gensym)))
      (if (null? clist)
          `(if #f #f)
          (if (equal? (caar clist) 'else)
              `(begin ,@(cdar clist))
              `(let ((,symbol ,(caar clist)))
                 (if ,symbol
                     ,(if (null? (cdar clist))
                          symbol
                          (if (equal? (cadar clist) '=>)
                              `(,(caddar clist) ,symbol)
                              `(begin ,@(cdar clist))))
                     (cond ,@(cdr clist)))))))))
```

```
(cond ((= n 1) 'jedna)
      ((= n 2))
      ((= n 3) (display n)
               (newline)
               (+ n 1))
      ((and (> n 4) n) =>
       (lambda (x) (* x x)))
      (else 'nevim))))
```

⇓

```
(let ((„symbol1“ (= n 1)))
 (if „symbol1“ (begin (quote jedna))
     (let ((„symbol2“ (= n 2)))
       (if „symbol2“ „symbol2“
         (let ((„symbol3“ (= n 3)))
           (if „symbol3“ (begin (display n)
                                (newline) (+ n 1))
               (let ((„symbol4“ (and (> n 4) n)))
                 (if „symbol4“ ((lambda (x) (* x x)) „symbol4“)
                     (begin (quote nevim)))))))))))
```

Implementace maker realizujících cykly

```
;; cyklus typu while
(define-macro while
  (lambda (condition . body)
    (let ((loop-name (gensym)))
      `(let ,loop-name ()
            (if ,condition
                (begin ,@body
                       (,loop-name)))))))
```

Příklad použití:

```
(let ((i 0) (j 0))
  (while (< i 10)
         (set! j (+ j i))
         (set! i (+ i 1)))
  (list i j))  ⟼  (10 45)
```

Příklad použití:

```
(let ((i 0) (j 0))
  (while (< i 10)
         (set! j (+ j i))
         (set! i (+ i 1)))
  (list i j))  ⟹  (10 45)
```

⇓

```
(let „symbol“ ()
  (if (< i 10)
      (begin (set! j (+ j i))
             (set! i (+ i 1))
             (list i j)
             („symbol“))))
```

Tohle ale bude vracet nedefinovanou hodnotu.

Úprava: vrací hodnotu vyhodnocení posledního výrazu v těle

```
(define-macro while
  (lambda (condition . body)
    (let ((loop-name (gensym))
          (last-value (gensym)))
      `(let ,loop-name ((,last-value (if #f #f)))
            (if ,condition
                (,loop-name (begin ,@body))
                ,last-value)))))
```

Příklad použití:

```
(let ((i 0)
      (j 0))
  (while (< i 10)
         (set! j (+ j i))
         (set! i (+ i 1))
         (list i j)))  ⟼  (10 45)
```

Příklad použití:

```
(let ((i 0)
      (j 0))
  (while (< i 10)
         (set! j (+ j i))
         (set! i (+ i 1))
         (list i j)))  ⟹  (10 45)
```

```
(let „symbol1“ ((„symbol2“ (if #f #f)))
  (if (< i 10)
      („symbol1“ (begin (set! j (+ j i))
                        (set! i (+ i 1))
                        (list i j)))
      „symbol2“))
```