

PARADIGMATA PROGRAMOVÁNÍ 2A

MAKRA II



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

Motivace: Chceme vyřešit problém se „symbol capture“

V následujícím makru dochází k zachycení symbolu `curval`

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

Příklad použití:

```
(let ((curval 10))
  (capture
    (display "Hodnota: ")
    (display curval)
    (newline)
    (+ curval 1))))  $\Rightarrow$  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))  $\Rightarrow$  101
```

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

`'ahoj` \implies element *symbol*, který má jméno *ahoj*

`'blah` \implies element *symbol*, který má jméno *blah*'

`(define s 'ahoj)` na `s` se naváže `ahoj`

`s` \implies `ahoj` (to jest na `s` je jako hodnota navázaný symbol)

Porovnávání pojmenovaných symbolů probíhá vzhledem k jejich jménům.

`(equal? 'ahoj 'blah)` \implies `#f`

`(equal? 'ahoj 'ahoj)` \implies `#t`

`(eq? 'ahoj 'blah)` \implies `#f`

`(eq? 'ahoj 'ahoj)` \implies `#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)` \implies nově vygenerovaný symbol

`(symbol? (gensym))` \implies `#t`

`(equal? (gensym) (gensym))` \implies `#f`

`(define s (gensym))`

`(equal? s s)` \implies `#t`

`s` \implies `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))))
```

⇓

```
(let ((„vygenerovaný symbol“ 100))  
  (display "Hodnota: ")  
  (display curval)  
  (newline)  
  (+ curval 1))  $\Rightarrow$  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)
```



```
(let ((„symbol“ 1))  
  (if „symbol“ „symbol“ (or 2 3)))  $\implies$  ...
```

V pořádku (jednonásobné vyhodnocení):

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

Rovněž v pořádku (nedochází k symbol capture):

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

Speciální forma `case`

Příklad použití:

```
(case (+ 1 2)
  ((0 1 2) 'blah)
  ((3 4) 'ahoj)
  (else 'nic))  $\Rightarrow$  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)))  $\Rightarrow$  10 místo 1000
```

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))))))
```

Speciální forma `cond` podle R5RS.

Některé možnosti `cond` jsme zatím zatajovali.

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

`(cond)` \implies nedefinovaná hodnota

`(cond (else 'blah))` \implies `blah`

`(cond ('blah))` \implies `blah`

`(cond (10 => -))` \implies `-10`

`(cond ((= 1 1)
 (display "X")
 (newline)
 (+ 1 2)))` \implies 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)	\Rightarrow	nevim
(test 1)	\Rightarrow	jedna
(test 2)	\Rightarrow	#t
(test 3)	\Rightarrow	4 rovněž zobrazí 3
(test 10)	\Rightarrow	100

Speciální forma `cond` podle R5RS

```
(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))))))))))
```

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))  $\Rightarrow$  (10 45)
```

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)))  $\Rightarrow$  (10 45)
```


Makro pro cyklus typu for

Příklad zamýšleného cyklu

```
(let ((i 0)
      (result 0))
  (for (set! i 5)
    (> i 0)
    (set! i (- i 1))
    (display (list "Stav: " i result))
    (newline)
    (set! result (+ result i)))
  (display (list "Koncovy: " i result))
  (newline))
```

Řešení

;; cyklus typu for (C, PERL a další)

```
(define-macro for
  (lambda (init condition incr . body)
    (let ((loop-name (gensym)))
      `(begin
         ,init
         (let ,loop-name ()
           (if ,condition
               (begin ,@body
                      ,incr
                      (,loop-name))))))))))
```

Cyklus `do`: Nativní cyklus jazyka Scheme

Příklad použití:

```
(do ((x '(1 3 5 7 9) (cdr x))      ; navázaný symbol
    (sum 0 (+ sum (car x)))        ; navázaný symbol
    ((null? x) sum)                ; limitní podmínka
    (display (list x sum))         ; tělo cyklu
    (newline)))  $\Rightarrow$  25
```

Během iterace se postupně zobrazí:

```
((1 3 5 7 9) 0)
```

```
((3 5 7 9) 1)
```

```
((5 7 9) 4)
```

```
((7 9) 9)
```

```
((9) 16)
```

;; cyklus do pomocí letrec

```
(define-macro do
  (lambda (binding condition . body)
    (let ((loop-name (gensym)))
      `(letrec ((,loop-name
                  (lambda ,(map car binding)
                    (if ,(car condition)
                        (begin ,@(cdr condition))
                        (begin ,@body
                              (,loop-name
                               ,@(map caddr binding)))))))
        (,loop-name ,@(map cadr binding))))))
```

;; cyklus *do* pomocí pojmenovaného *letu* (úprava předchozího)

```
(define-macro do
  (lambda (binding condition . body)
    (let ((loop-name (gensym)))
      `(let ,loop-name
         ,(map (lambda (x)
                  (list (car x) (cadr x)))
               binding)
          (if ,(car condition)
              (begin ,@(cdr condition))
              (begin ,@body
                     (,loop-name
                      ,@(map caddr binding))))))))))
```

Cyklus typu `repeat ~ until`

Příklad zamýšleného použití

```
(let ((x 20)
      (y 15))
  (repeat
    (set! y (+ y 4))
    (set! x (- x 1))
    (until ((<= x 10) (list 'foo x y))
            ((>= y 30) (list 'bar y (+ x 20))))))
⇒ (bar 31 36)
```

- příkazy v těle vždy proběhnou alespoň jednou
- cyklus se opakuje, dokud není splněna (některá) limitní podmínka
- test limitních podmínek probíhá vždy po dokončení těla

```

(define but-last
  (lambda (l)
    (cond ((null? l) #f)
          ((null? (cdr l)) (cons '() (car l)))
          (else (let ((result (but-last (cdr l))))
                    (cons (cons (car l) (car result))
                          (cdr result)))))))

```

`(but-last '(a b c d))` \implies `((a b c) . d)`

```

(define but-last
  (lambda (l)
    (foldr (lambda (x y)
              (if y
                  (cons (cons x (car y)) (cdr y))
                  (cons '() x)))
            #f
            l)))

```

;; makro realizující cyklus typu repeat~until

(define-macro repeat

(lambda (args

(define but-last ...) ; interně definovaný but-last

(let* ((split-args (but-last args))

(body (car split-args))

(limits (cdr split-args))

(loop-name (gensym)))

`(let ,loop-name ()

,@body

(cond ,@(map (lambda (conds)

`(,(car conds)

(begin ,@(cdr conds))))

(cdr limits))

(else (,loop-name))))))

Poznámka o makrech v Dr. Scheme

- transformační procedura makra se neaplikuje v prostředí svého vzniku, ale v *prostředí počátečních vazeb*,
- prostředí počátečních vazeb \neq globální prostředí,
- prostředí počátečních vazeb: nelze v něm definovat vazby,
- omezení Dr. Scheme kvůli oddělení makroexpance a vyhodnocování.

;; pomocná procedura

```
(define proc  
  (lambda (x) (list '- x)))
```

;; makro

```
(define-macro m  
  (lambda (elem) (proc elem)))
```

(m 10) \Rightarrow Error: Symbol `proc` je nenavázaný

KVAZIKVOTOVÁNÍ – úkolem je vyrobit makro realizující kvazikvotování

`(kvaziquote blah)`



`(quote blah) \implies blah`

`(kvaziquote (a b))`



`(apply append (list (quote a)) (list (quote b))
 (quote ())) \implies (a b)`

`(kvaziquote (a (unquote (+ 1 2))))`



`(apply append (list (quote a)) (list (+ 1 2)) (quote ()))`

`(kvaziquote (a (unquote-splicing 1)))`



`(apply append (list (quote a)) 1 (quote ())) \implies ...`

;; pomocná transformační procedura

```
(define trans-expr
  (lambda (expr)
    (cond
      ((or (not (list? expr)) (null? expr))
       (list 'list (list 'quote expr)))
      ((eq? (car expr) 'unquote) (list 'list (cadr expr)))
      ((eq? (car expr) 'unquote-splicing) (cadr expr))
      ((eq? (car expr) 'kvaziquote)
       (list 'list (list 'quote expr)))
      (else (list 'list (list 'kvaziquote expr))))))
```

(te 1)	\Rightarrow	(list (quote 1))
(te '())	\Rightarrow	(list (quote ()))
(te '(1 2 3))	\Rightarrow	(list (kvaziquote (1 2 3)))
(te '(unquote (1 2 3)))	\Rightarrow	(list (1 2 3))
(te '(unquote-splicing (1 2 3)))	\Rightarrow	(1 2 3)
(te '(kvaziquote (1 2)))	\Rightarrow	(list (quote (kvaz. (1 2))))

;; makro pro kvazikvotování bez použití kvazikvotování

```
(define-macro kvaziquote  
  (lambda (expr)
```

;; pomocná transformační procedura (předchozí slide)

```
(define trans-expr (lambda (expr) ...))
```

;; vlastní transformace

```
(if (not (list? expr))  
    (list 'quote expr)  
    (apply list 'apply 'append  
            (append (map trans-expr expr)  
                    '((quote ()))))))
```

HYGIENICKÁ MAKRA

Proč „hygienická“?

Protože umožňují vytvářet bezpečná makra.

Základní rysy

- definována v R5RS (kromě Scheme, pokud vím, nikdo nemá)
- kompletně jiný přístup k makrům než `define-macro`
- makra jsou definována pomocí (několika) přepisovacích pravidel

Výhody

- prakticky odpadají složité kvazikvotované výrazy
- nemůže nastat *symbol capture*
- makra jsou v souladu s lexikálním rozsahem platnosti
- makra lze definovat lokálně

Nevýhody

- některá makra se tímto způsobem nedělají pohodlně

Soulad s lexikálním rozsahem platnosti spočívá ve:

- 1 Jestliže je v těle makra definována vazba na dosud nepoužitý symbol, tento symbol je v těle makra *automaticky přejmenován* tak, aby nemohlo dojít ke kolizi se jménem již existujícího symbolu.
 - o přejmenování symbolu se „programátor nestará“
 - přejmenování probíhá zcela transparentně
- 2 Při vyhodnocování těla makra se vazby všech volných výskytů symbolů (to jest vazby symbolů, které nebyly vytvořené lokálně v rámci makra) hledají v prostředí definice makra
 - prostředí definice makra = lexikální předek
 - při použití makra nezáleží na vazbách v prostředí použití makra

Vytvoření hygienického makra

- `(define-syntax <nazev> <transformacni-procedura>)`
- `<transformacni-procedura>` vzniká pomocí spec. formy `syntax-rules`

Vytvoření transformační procedury hygienického makra

- `(syntax-rules <klicova-slova> <pravidlo1> <pravidlo2> ...)`
- `<klicova-slova>` ... seznam symbolů, které jsou dále chápány jako klíčová slova (seznam může být prázdný)
- `<pravidlon>` ... přepisovací pravidla, viz dále

Přepisovací pravidla jsou pravidla tvaru `(<vzor> <nahrazní>)`, kde

- `<vzor>` je výraz specifikující konkrétní případ použití makra, viz dále
- `<nahrazní>` je libovolný výraz, kterým bude „volání makra“ nahrazeno v případě shody s daným vzorem

Vzory (pro detaily viz R5RS) se skládají ze:

- **symboly** ... označují *klíčová slova* nebo *vstupní elementy*
- **seznamy** skládající se ze vzorů
- speciální vzor výpustka „...“ (tři tečky)
význam: vzor před kterým je výpustka se může několikrát opakovat
nebo nemusí být přítomen

Vzory se porovnávají (na úplnou shodu) se vstupem jeden po druhém.

Symbolsy vyskytující se ve vzoru (kromě prvního) mohou být:

- 1 symboly vyskytující se mezi klíčovými slovy
 - shoda se vzorem nastává pouze v případě, kdy má vstupní výraz na dané pozici stejný symbol
- 2 symboly nevyskytující se mezi klíčovými slovy
 - během porovnávání vstupního výrazu se vzorem jsou takové symboly navázány na vstupní výraz

První symbol ve vzoru se shoduje s názvem makra.

Makro `and` realizované jako hygienické makro

```
(define-syntax and
  (syntax-rules ()
    ((and) #t) ; žádná klíčová slova
    ((and test) test) ; and bez argumentu
    ((and test1 test2 ...) ; and s jedním argumentem
     (if test1 (and test2 ...) #f)))) ; dva a více argumentů
```

Makro `setf!` (v tomto případě slouží `car`, `cdr` a `ref` jako klíčová slova

```
(define-syntax setf!
  (syntax-rules (car cdr ref)
    ((setf! (car pair) value) (set-car! pair value))
    ((setf! (cdr pair) value) (set-cdr! pair value))
    ((setf! (ref vector index) value)
     (vector-set! vector index value))
    ((setf! symbol value) (set! symbol value))))
```

Nefunkční verze `setf!` (`car`, `cdr` a `ref` nejsou uvedena jako klíčová slova
(`define-syntax setf!`

```
(syntax-rules ()  
  ((setf! (car pair) value) (set-car! pair value))  
  ((setf! (cdr pair) value) (set-cdr! pair value))  
  ((setf! (ref vector index) value)  
   (vector-set! vector index value))  
  ((setf! symbol value) (set! symbol value))))
```

Příklad, proč výše uvedené nefunguje:

```
(define p (cons 10 20))  
(setf! (cdr p) 'svete) ; použito bude první pravidlo  
p  $\Rightarrow$  (svete . 20)
```

Důvod nefunkčnosti:

- symbol `cdr` ve vstupním výrazu se naváže na symbol `car`
- vstupní výraz tím pádem odpovídá prvnímu pravidlu

Makro `or` realizované jako hygienické makro

```
(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((result test1))
       (if result result (or test2 ...))))))
```

Makro `def` jako hygienické makro (zde záleží na pořadí pravidel)

```
(define-syntax def
  (syntax-rules ()
    ((def (name arg ...) stmt ...)
     (define name (lambda (arg ...) stmt ...)))
    ((def symbol stmt)
     (define symbol stmt))))
```

Složitější příklad použití hygienických maker: „for“ à la Pascal

```
for  $i := start$  to/downto  $end$  [step  $k$ ] do  
     $stmt_1$   
     $stmt_2$   
     $\vdots$   
     $stmt_n$   
endfor
```

Cyklus bychom chtěli používat takto:

```
(for i := 1 to 10 do (display i) (newline))  
(for i := 10 downto 1 do (display i) (newline))  
(for i := 1 to 10 step 2 do (display i) (newline))  
(for i := 10 downto 1 step 2 do (display i) (newline))
```

Pomocí různých vzorů rozlišíme jednotlivé případy použití.

Příkaz `for` ve stylu jazyka Pascal

(define-syntax for

(syntax-rules (:= to downto do step)

((for var := start to end do stmt ...)

(let loop ((var start))

(if (<= var end)

(begin

stmt ...

(loop (+ var 1))))))

((for var := start downto end do stmt ...)

(let loop ((var start))

(if (>= var end)

(begin

stmt ...

(loop (- var 1))))))

: pokračujeme na dalším slajdu

∴ pokračování z předchozího slajdu

```
((for var := start to end step inc do stmt ...)  
  (let loop ((var start))  
    (if (<= var end)  
      (begin  
        stmt ...  
        (loop (+ var inc))))))  
((for var := start downto end step dec do stmt ...)  
  (let loop ((var start))  
    (if (>= var end)  
      (begin  
        stmt ...  
        (loop (- var dec))))))
```

Hygienická makra je možné **definovat lokálně** pomocí speciálních forem:

- `let-syntax` ... jednotlivá pravidla se „vzájemně nevidí“
- `letrec-syntax` ... pravidla se všechna „vzájemně vidí“, pravidla mohou používat ostatní pravidla (hrozí zacyklení)

Příklad lokální definice makra `when` v proceduře

```
(define f
  (lambda (n)
    (let-syntax
      ((when (syntax-rules ()
              ((when test stmt1 ...)
                (if test
                    (begin stmt1 ...))))))
      (when (> n 3) (display "BLAH") (newline) (+ n 1)))))
```

(f 1) \implies nedefinovaná hodnota

(f 4) \implies 5 rovněž vytiskne BLAH

V následujícím příkladu nedojde u symbolu `test` k jeho zachycení

```
(define f
  (lambda (n)
    (let-syntax
      ((when (syntax-rules ()
              ((when test stmt1 ...)
                (if test
                    (begin stmt1 ...))))))
      (let ((test #f))
        (when (> n 3)
          (display (list test "BLAH"))
          (newline)
          (+ n 1)))))))
```

(f 1) \Rightarrow nedefinovaná hodnota

(f 4) \Rightarrow 5 rovněž se vytiskne (#f BLAH)

Následující nebude fungovat: & je definované pomocí &

(let-syntax

```
((& (syntax-rules ()  
    ((&) #t)  
    ((& test) test)  
    ((& test1 test2 ...)   
      (if test1 (& test2 ...) #f)))))
```

(& 1 2 3)) \Rightarrow Error: & not bound

Následující už bude fungovat (díky letrec-syntax)

(letrec-syntax

```
((& (syntax-rules ()  
    ((&) #t)  
    ((& test) test)  
    ((& test1 test2 ...)   
      (if test1 (& test2 ...) #f)))))
```

(& 1 2 3)) \Rightarrow 3