

KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA  
UNIVERZITA PALACKÉHO, OLMOUC

# PARADIGMATA PROGRAMOVÁNÍ 2A

## INTERPRET S VEDLEJŠÍMI EFEKTY A MAKRY

Slajdy vytvořili Vilém Vychodil a Jan Konečný

## Základní interpret Scheme (ve Scheme)

- máme udělané z předchozího semestru
- interpret Scheme ve Scheme, který je *čistě funkcionální*

### Co interpret umí:

- procedury: primitivní, uživatelské, procedury vyšších řádů
- elementy prvního řádu: čísla, symboly, seznamy, procedury, prostředí

### Co interpret neumí:

- žádný element není mutovatelný (ani páry, ani prostředí, ...)
- neumí (re)definovat/změnit vazbu symbolu (nemá `define` ani `set!`)
- rekurzivní procedury je potřeba zavádět pomocí  $\lambda$ -kombinátoru (to je nepohodlné)
- neumí sekvencovat výrazy (nemá `begin`): nelze pohodlně vytvářet interní definice
- nemá makra (uživatelsky definované formy)

# Systém manifestovaných typů

*;; vytvoř element jazyka s manifestovaným typem*

```
(define curry-make-elem  
  (lambda (type-tag)  
    (lambda (data)  
      (cons type-tag data))))
```

*;; vrať visačku s typem, vrať data*

```
(define get-type-tag car)  
(define get-data cdr)
```

*;; test daného typu*

```
(define curry-scm-type  
  (lambda (type)  
    (lambda (elem)  
      (equal? type (get-type-tag elem)))))
```

# Základní elementy jazyka Scheme

## Čísla

```
(define make-number (curry-make-elem 'number))  
(define scm-number? (curry-scm-type 'number))
```

## Symboly

```
(define make-symbol (curry-make-elem 'symbol))  
(define scm-symbol? (curry-scm-type 'symbol))
```

## Tečkové páry

;; konstruktor páru *cons*

```
(define make-pair
  (let ((make-physical-pair (curry-make-elem 'pair)))
    (lambda (head tail)
      (make-physical-pair (cons head tail)))))
```

;; test datového typu

```
(define scm-pair? (curry-scm-type 'pair))
```

;; selektor páru *car* (*cdr* se udělá analogicky)

```
(define pair-car
  (lambda (pair)
    (if (scm-pair? pair)
        (car (get-data pair))
        (error "; Car: argument must be a pair")))))
```

**Prostředí** – zavedeme jako tabulku vytvořenou pomocí párů

Takto si prostředí „představujeme“:

- **tabulka vazeb**: symbol – element (hodnota navázaná na symbol),
- **ukazatel na předka** (prostředí, které je „výš v hierarchii“).

<i>symbol</i>	<i>element</i>
$E_1$	$F_1$
$E_2$	$F_2$
$\vdots$	$\vdots$
$E_k$	$F_k$
$\vdots$	$\vdots$

kde  $E_1, E_2, \dots$  jsou **symboly** a  
 $F_1, F_2, \dots$  jsou **elementy**;  
+ **ukazatel na předka**

**Implementace pomocí párů**

(predek . (( $E_1$  .  $F_1$ )  
          ( $E_2$  .  $F_2$ )  
          ...  
          ( $E_k$  .  $F_k$ )  
          ...))

## Prostředí

*;; převed' asociční seznam na tabulku prostředí*

```
(define assoc->env
  (lambda (l)
    (if (null? l)
        the-empty-list
        (make-pair (make-pair
                    (make-symbol (caar l))
                    (cdar l))
                  (assoc->env (cdr l))))))
```

*;; konstruktor prostředí*

```
(define make-env
  (let ((make-physical-env (curry-make-elem 'environment)))
    (lambda (pred table)
      (make-physical-env
       (cons pred table)))))
```

*;; test datového typu*

```
(define scm-env? (curry-scm-type 'environment))
```

*;; konstruktor globálního prostředí*

```
(define make-global-env  
  (lambda (alist-table)  
    (make-env scm-false (assoc->env alist-table))))
```

*;; vrať tabulku*

```
(define get-table  
  (lambda (elem)  
    (if (scm-env? elem)  
        (cdr (get-data elem))  
        (error "; Get-table: arg. must be an env."))))
```



*;; vrat předka*

```
(define get-pred
  (lambda (elem)
    (if (scm-env? elem)
        (car (get-data elem))
        (error "; Get-pred: arg. must be an env."))))
```

*;; je globální prostředí?*

```
(define global?
  (lambda (elem)
    (and (scm-env? elem)
         (equal? scm-false (get-pred elem)))))
```

;; hledání vazeb v asociačním poli

```
(define scm-assoc
  (lambda (key alist)
    (cond ((scm-null? alist) scm-false)
          ((equal? key (pair-car (pair-car alist)))
           (pair-car alist))
          (else (scm-assoc key (pair-cdr alist))))))
```

;; vyhledej vazbu v prostředí env, nebo vrať not-found

```
(define lookup-env
  (lambda (env symbol search-nonlocal? not-found)
    (let ((found (scm-assoc symbol (get-table env))))
      (cond ((not (equal? found scm-false)) found)
            ((global? env) not-found)
            ((not search-nonlocal?) not-found)
            (else (lookup-env (get-pred env)
                               symbol #t not-found))))))
```

## Primitivní procedury

*;; konstruktor primitivní procedury a predikát*

```
(define make-primitive (curry-make-elem 'primitive))  
(define scm-primitive? (curry-scm-type 'primitive))
```

*;; vytváření primitivních procedur pomocí wrapperu*

```
(define wrap-primitive  
  (lambda (proc)  
    (make-primitive  
      (lambda arguments  
        (expr->intern  
          (apply proc (map get-data arguments))))))))
```

## Uživatelské procedury

```
(define make-procedure
  (let ((make-physical-procedure
        (curry-make-elm 'procedure)))
    (lambda (env args body)
      (make-physical-procedure (list env args body)))))

(define procedure-environment ...)
(define procedure-arguments ...)
(define procedure-body ...)

(define scm-user-procedure? (curry-scm-type 'procedure))
(define scm-procedure?
  (lambda (elem)
    (or (scm-primitive? elem)
        (scm-user-procedure? elem))))
```

## Primitivní speciální formy

```
(define make-specform (curry-make-elem 'specform))
```

```
(define scm-specform? (curry-scm-type 'specform))
```

```
(define scm-form?
```

```
  (lambda (elem)
```

```
    (or (scm-specform? elem))))
```

## Speciální elementy jazyka

*;; pravdivostní hodnoty*

```
(define scm-false ((curry-make-elem 'boolean) #f))  
(define scm-true  ((curry-make-elem 'boolean) #t))  
(define scm-boolean? (curry-scm-type 'boolean))
```

*;; prázdný seznam*

```
(define the-empty-list  
  ((curry-make-elem 'empty-list) '()))  
(define scm-null?  
  (lambda (elem) (equal? elem the-empty-list)))
```

*;; nedefinovaná hodnota*

```
(define the-undefined-value ((curry-make-elem 'undefined) '()))  
(define scm-undefined?  
  (lambda (elem) (equal? elem the-undefined-value)))
```

## Reader

*;; převedení výrazu do interní formy*

```
(define expr->intern
  (lambda (expr)
    (cond ((symbol? expr) (make-symbol expr))
          ((number? expr) (make-number expr))
          ((and (boolean? expr) expr) scm-true)
          ((boolean? expr) scm-false)
          ((null? expr) the-empty-list)
          ((pair? expr)
           (make-pair (expr->intern (car expr))
                      (expr->intern (cdr expr))))
          ((eof-object? expr) #f)
          (else (error "; Syntactic error."))))
```

*;; načti vstupní výraz do interní formy*

```
(define scm-read  
  (lambda ()  
    (expr->intern (read))))
```

## Printer

*;; pouze použije display a vypíše syrovou reprezentaci*

```
(define scm-print  
  (lambda (elem)  
    (display elem)))
```



## Pomocné procedury

```
;; map přes elementy tvořící scm-seznam  
;; výsledkem je klasický seznam  
(define map-scm-list->list ...
```

```
;; převed' scm-seznam na klasický seznam  
(define scm-list->list ...
```

```
;; převed' klasický seznam na scm-seznam  
(define list->scm-list ...
```

## Evaluátor

*;; vyhodnot' výraz v daném prostředí*

```
(define scm-eval  
  (lambda (elem env)
```

*;; vyhodnocování elementů podle jejich typu*

```
(cond
```

*;; symboly se vyhodnocují na svou aktuální vazbu*

```
((scm-symbol? elem)  
 (let ((binding (lookup-env env elem #t #f)))  
   (if binding  
       (pair-cdr binding)  
       (error "; EVAL: Symbol not bound")))))
```

*;; vyhodnocení seznamu*

```
((scm-pair? elem)
```

*;; nejprve vyhodnotíme první prvek seznamu*

```
(let* ((first (pair-car elem))  
      (args (pair-cdr elem))  
      (f (scm-eval first env)))
```

*;; podle prvního prvku rozhodni o co se jedná*

```
(cond
```

*;; pokud se jedná o proceduru:*

*;; vyhodnoť argumenty a aplikuj ji*

```
((scm-procedure? f)  
 (scm-apply f (map-scm-list->list  
              (lambda (elem)  
                (scm-eval elem env))  
              args)))
```

```
;; pokud se jedná o formu  
;; aplikuj s nevyhodnocenými argumenty:  
((scm-form? f)  
  (scm-form-apply env f (scm-list->list args))))
```

```
;; na prvním místě stojí nepřipustný prvek  
(error "; EVAL: First element ...")))
```

```
;; vše ostatní se vyhodnocuje na sebe sama  
(else elem)))
```

;; vytvoř tabulku vazeb: formální argument – argument

```
(define make-bindings
  (lambda (formal-args args)
    (cond ((scm-null? formal-args) the-empty-list)
          ((scm-symbol? formal-args)
           (make-pair (make-pair formal-args
                                 (list->scm-list args))
                      the-empty-list))
          (else (make-pair
                  (make-pair (pair-car formal-args)
                             (car args))
                  (make-bindings (pair-cdr formal-args)
                                (cdr args)))))))
```

*;; aplikuj proceduru, předka nastav na env*

```
(define scm-env-apply
  (lambda (proc env args)
    (cond ((scm-primitive? proc)
           (apply (get-data proc) args))
          ((scm-user-procedure? proc)
           (scm-eval (procedure-body proc)
                     (make-env
                      env
                      (make-bindings
                       (procedure-arguments proc)
                       args))))
          (else (error "APPLY: Expected procedure")))))
```

*;; aplikuj proceduru s lexikálním předkem*

```
(define scm-apply
  (lambda (proc args)
    (cond ((scm-primitive? proc)
           (scm-env-apply proc #f args))
          ((scm-user-procedure? proc)
           (scm-env-apply
            proc
            (procedure-environment proc)
            args))
          (else (error "APPLY: Expected procedure")))))
```

*;; aplikuj speciální formu*

```
(define scm-form-apply
  (lambda (env form args)
    (cond ((scm-specform? form)
           (apply (get-data form) env args))
          (else (error "APPLY: Expected sp. form")))))
```



## Toplevel Environment (počáteční prostředí)

*;; vytvoř prostředí, které je nejvyšší v hierarchii*

```
(define scheme-toplevel-env
  (make-global-env
    '(
      ;; speciální forma if
      (if . ,(make-specform
              (lambda (env condition expr . alt-expr)
                (let ((result (scm-eval condition env)))
                  (if (equal? result scm-false)
                      (if (null? alt-expr)
                          the-undefined-value
                          (scm-eval (car alt-expr) env))
                      (scm-eval expr env)))))))
```

*;; speciální formy and a or*

```
(and . , (make-specform ...  
(or . , (make-specform ...
```

*;; speciální forma lambda (v těle jen jeden výraz)*

```
(lambda . , (make-specform  
              (lambda (env args body)  
                (make-procedure env args body))))
```

*;; speciální forma the-environment*

```
(the-environment . , (make-specform  
                    (lambda (env) env)))
```

*;; speciální forma quote*

```
(quote . , (make-specform  
           (lambda (env elem) elem)))
```

```
;; aritmetika
(* . ,(wrap-primitive *))
(+ . ,(wrap-primitive +))
  ⋮
;; práce s páry
(cons . ,(make-primitive make-pair))
(car . ,(make-primitive pair-car))
(cdr . ,(make-primitive pair-cdr))

;; negace
(not . ,(make-primitive
         (lambda (elem)
           (if (equal? elem scm-false)
               scm-true
               scm-false))))))
```

*;; další selektory*

```
(environment-parent . ,(make-primitive get-pred))  
(procedure-environment . ...  
(procedure-arguments . ...  
(procedure-body . ...
```

*;; konverze prostředí na seznam*

```
(environment->list . ,(make-primitive  
    (lambda (elem)  
      (if (equal? elem scm-false)  
          scm-false  
          (get-table elem))))))
```

```
;; procedura eval (dvou argumentů)
(eval . ,(make-primitive
          (lambda (elem env)
            (scm-eval elem env))))
```

```
;; procedura apply
(apply . ,(make-primitive
           (lambda (proc . rest)
             (scm-apply proc
                        (apply-collect-arguments rest))))))
```

;; procedura *apply* s explicitním prostředím předka

```
(env-apply . ,(make-primitive
               (lambda (proc env . rest)
                 (scm-env-apply proc
                                env
                                (apply-collect-arguments rest))))))
```

))) ; konec toplevel environment

## Globální prostředí: Pomocné procedury

*;; pro obecný typ volání `apply` sestaví seznam argumentů*

```
(define apply-collect-arguments
  (lambda (args)
    (cond ((null? args)
           (error "APPLY: argument missing"))
          ((and (not (null? args)) (null? (cdr args)))
           (scm-list->list (car args)))
          (else (cons (car args)
                      (apply-collect-arguments
                       (cdr args)))))))
```

## Důsledky neexistence define

- rekurze pomocí  $y$ -kombinátorů
- do globálního prostředí nelze během činnosti interpretu zavést nové definice (uživatelských procedur)

## Hierarchie tří počátečních prostředí

### 1 `toplevel-environment`

- v hierarchii úplně nejvyšší (nemá předka)
- obsahuje základní definice (primitivní procedury a spec. formy)

### 2 `midlevel-environment`

- jeho předkem je `toplevel-environment`
- obsahuje definice uživatelských procedur, které jsou k dispozici na počátku běhu interpretu (`map`, `length`, ...)

### 3 `global-environment`

- jeho předkem je `midlevel-environment`
- neobsahuje žádné definice

```

(define scheme-midlevel-env
  (make-env scheme-toplevel-env
    (assoc->env '(
      :
      (map .
        ,(make-procedure
          scheme-toplevel-env
          (expr->intern '(f l))
          (expr->intern
            '((lambda (y)
              (y y l))
              (lambda (map l)
                (if (null? l)
                  ()
                  (cons (f (car l))
                      (map map (cdr l))))))))))
      ))))

```



## Cyklus REPL

```
(define scm-repl
  (lambda ()
    (let ((glob-env (make-env
                     scheme-midlevel-env
                     the-empty-list)))
      (let loop ()
        (display "]=> ")
        (let ((elem (scm-read)))
          (if (not elem)
              'bye-bye
              (let ((result (scm-eval elem glob-env)))
                (scm-print result)
                (loop))))))))))
```

;; spuštění REPLu

```
(scm-repl)
```

## Příklady použití interpretu

`lambda`  $\mapsto$  spec. forma

`(lambda (x) (+ x 1))`  $\mapsto$  procedura

`((lambda (x) (+ x 1)) 10)`  $\mapsto$  11

`((lambda (proc) (proc (x) (+ x 1))) lambda) 10`  $\mapsto$  11

`((lambda list (map - list)) 1 2 3 4)`  $\mapsto$  (-1 -2 -3 -4)

`(eval '(* x x)`

`(procedure-environment`

`((lambda (x)`

`(lambda (y) (+ x y)))`

`10)))`  $\mapsto$  100

`(apply ((lambda (pi) (lambda (x) (+ x pi))) 10) 20 '())`  
 $\mapsto$  30

`(env-apply (lambda (x) (+ x y))`

`((lambda (y) (the-environment)) 100)`

`20 '())`  $\mapsto$  120

## Interpret obohacený o imperativní rysy

- budeme rozšiřovat předchozí interpret
- budeme postupovat metodou „nejmenšího odporu“

### Postup implementace

- 1 mutovatelné tečkové páry: `set-car!`, `set-cdr!`  
(umožní destruktivní práci se seznamy)
- 2 mutovatelné prostředí  
(umožní destruktivní změny prostředí, například změny vazeb)
- 3 sekvencování: `begin`, `lambda`  
(umožní rozumné interní definice)
- 4 zavedení mutátorů prostředí: `define` a `set!`  
(umožní zavádění nových definic + imperativní změnu vazeb)

## Mutovatelné tečkové páry

*;; mutátor páru set-car!*

```
(define pair-set-car!  
  (lambda (pair value)  
    (if (scm-pair? pair)  
        (begin  
          (set-car! (get-data pair) value)  
          the-undefined-value)  
        (error "SET-CAR!: argument must be a pair"))))
```

*;; mutátor páru set-cdr!*

```
(define pair-set-cdr!  
  (lambda (pair value)  
    (if (scm-pair? pair)  
        (begin  
          (set-cdr! (get-data pair) value)  
          the-undefined-value)  
        (error "SET-CDR!: argument must be a pair"))))
```

## Mutovatelné tečkové páry

*;; do globálního prostředí přidáme:*

```
(set-car! . ,(make-primitive pair-set-car!))  
(set-cdr! . ,(make-primitive pair-set-cdr!))
```

V tuto chvíli můžeme používat `set-car!` a `set-cdr!` v interpretu:

*;; příklad imperativní změny argumentů procedury*

```
(set-car! (procedure-arguments map) 'blah)  
(procedure-arguments map)  $\implies$  (blah 1)  
(map - '(1 2 3 4))  $\implies$  error: symbol f not bound
```

## Mutovatelné prostředí: předehra pro define a set!

- do každé tabulky vazeb přidáme napevno nový první prvek „#f“
- tím zajistíme že každá tabulka vazeb bude mutovatelná pomocí mutátorů `set-car!` a `set-cdr!`

```
(predek . (#f (E1 . F1)  
            (E2 . F2)  
            ...  
            (En . Fn)))
```

*;; přidáme do globálního prostředí:*

```
(lookup-env .  
  ,(make-primitive  
    (lambda (env symbol . nonlocal)  
      (lookup-env env symbol  
                  (or (null? nonlocal)  
                      (equal? (car nonlocal) scm-true))  
                  scm-false))))
```

## Define jako procedura

- vytvoříme `define`, který nejprve „zhmotní sebe sama“, a pak
- použije „sebe sama“ k zavedení „sebe sama“ do prostředí

```
((lambda (y)
  (y (environment-parent (the-environment))
     'define
     y))
(lambda (env symbol value)
  (if (lookup-env env symbol #f)
      (set-cdr! (lookup-env env symbol) value)
      ((lambda (table)
         (set-cdr! table (cons (cons symbol value)
                               (cdr table))))
       (environment->list env))))))
```

## Define jako procedura

Příklady použití:

```
(define (the-environment)
  'faktorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (faktorial (- n 1))))))
```

```
(define (the-environment)
  'map
  (lambda (f l)
    (if (null? l)
        '()
        (cons (f (car l))
                (map f (cdr l))))))
```



## Set! jako procedura

;; zavedení *set!* užitím funkce *define*

```
(define (the-environment)
  'set!
  (lambda (env symbol value)
    (set-cdr! (lookup-env env symbol) value)))
```

Příklad použití:

```
(set! (the-environment) '* 10)
```

\*  $\Rightarrow$  10

```
(lookup-env (the-environment) '* )  $\Rightarrow$  (* . 10)
```

## Zabudované speciální formy define, set! a begin

;; speciální forma *begin*

```
(begin . , (make-specform
           (lambda (env . body)
             (let iter ((body body))
               (cond ((null? body) the-undefined-value)
                     ((null? (cdr body))
                      (scm-eval (car body) env))
                     (else (begin
                              (scm-eval (car body) env)
                              (iter (cdr body))))))))))
```

## Zabudované speciální formy define, set! a begin

;; speciální forma define

```
(define . ,(make-specform
            (lambda (env symbol value)
              (let ((value (scm-eval value env))
                    (result (lookup-env env symbol #f #f)))
                (if result
                    (pair-set-cdr! result value)
                    (pair-set-cdr!
                     (get-table env)
                     (make-pair (make-pair symbol value)
                                (pair-cdr (get-table env))))))))))
```

;; speciální forma set!

```
(set! . ,(make-specform
          (lambda (env symbol value)
            (pair-set-cdr! (lookup-env env symbol #t #f)
                           (scm-eval value env))))))
```

## Upravená lambda, tak aby uměla implicitní begin

*;; speciální forma lambda (umožňuje interní definice)*

```
(lambda .  
  ,(make-specform  
    (lambda (env args . body)  
      (if (not (null? (cdr body)))  
          (make-procedure  
            env args  
            (make-pair  
              (make-symbol 'begin)  
              (let iter ((ar body))  
                (if (null? (cdr ar))  
                    (make-pair (car ar)  
                                the-empty-list)  
                    (make-pair (car ar)  
                                (iter (cdr ar))))))))  
          (make-procedure env args (car body))))))
```

## Predikát eq?

;; naprogramování pomocí interního eq?

```
(define scm-eq?  
  (lambda (elem-a elem-b)  
    (eq? (get-data elem-a)  
         (get-data elem-b))))
```

- ostatní predikáty (eqv? a equal?) lze nadefinovat pomocí eq?
- eqv? a equal? tedy nemusí být zabudované

## Natahování definic z externího souboru

*;; natahuj výrazy a vyhodnocuj*

```
(define scm-load
  (lambda (file-name env)
    (let ((port (open-input-file file-name)))
      (let next ()
        (let ((expr (read port)))
          (if (not (eof-object? expr))
              (begin
                (scm-eval (expr->intern expr) env)
                (next))))))
      (close-input-port port))))
```

- předchozí procedura manipuluje se soubory pomocí portů viz R5RS
- načítá výrazy jeden po druhém a vyhodnocuje je v daném prostředí

## Upravený REPL

```
(define scm-repl
  (lambda ()
    (let ((glob-env (make-env scheme-toplevel-env
                              the-empty-list)))
      (scm-load "includes.scm" glob-env)
      (let loop ()
        (display "]=> ")
        (let ((elem (scm-read)))
          (if (not elem)
              'bye-bye
              (let ((result (scm-eval elem glob-env)))
                (newline)
                (scm-print result)
                (loop))))))))))
```

## Příklady použití interpretu

`(lookup-env (the-environment) '+)`  $\implies$  `(+ . #<...>)`

`(lookup-env (the-environment) '+ #f)`  $\implies$  `#f`

`(define f (lambda (n) (+ n x)))`

`((lambda (x)  
 (env-apply f  
 (the-environment)  
 20 '()))`

`10)`  $\implies$  `30`

`(procedure-body f)`  $\implies$  `(+ n x)`

`(set-car! (cddr (procedure-body f))) 'n)`

`(procedure-body f)`  $\implies$  `(+ n n)`

`(f 10)`  $\implies$  `20`



## Interpret obohacený o makra

- zavedeme nový element – **makro**
- makro v sobě obsahuje ukazatel na (transformační) proceduru

*;; konstruktor makra a detekce typu*

```
(define make-macro (curry-make-elem 'macro))  
(define scm-macro? (curry-scm-type 'macro))
```

*;; test zdali se jedná o primitivní/uživatelskou formu*

```
(define scm-form?  
  (lambda (elem)  
    (or (scm-specform? elem)  
        (scm-macro? elem))))
```

*;; do globálního prostředí přidáme:*

```
(macro . ,(make-primitive make-macro))  
(macro-transformer . ,(make-primitive get-data))
```

## Interpret obohacený o makra

- je třeba přizpůsobit vyhodnocovací proces pro případ, že prvním prvek seznamu se vyhodnotí na makro
- není třeba upravovat samotný `scm-eval`
- upravíme `scm-form-apply` (ošetříme nový případ pro makra)

*;; aplikuj speciální formu*

```
(define scm-form-apply
  (lambda (env form args)
    (cond ((scm-specform? form)
           (apply (get-data form) env args))
          ((scm-macro? form)
           (scm-eval (scm-apply (get-data form)
                                args)
                     env))
          (else (error "APPLY: Expected spec. form")))))
```

## Příklady použití našich maker

```
(macro (lambda (x y) x))  $\implies$  makro
(macro-transformer (macro (lambda (x y) x)))  $\implies$  tr. pr.
(define m (macro (lambda (x y) x)))
(m 1 2)  $\implies$  1
(m 1 nevyhodi-chybu)  $\implies$  1
(define first-of-2 (lambda (x y) x))
(define m (macro first-of-2))

(define let
  (macro
    (lambda (bindings . body)
      (append (list (list 'lambda
                        (map car bindings)
                        (cons 'begin body))))
              (map cadr bindings)))))

(let ((x 10) (y 20)) (+ x y))  $\implies$  30
```

## Příklady použití našich maker

```
((macro (lambda (x y) x)) 10 nevyhodi-chybu)  $\implies$  10
```

```
(define make-sender  
  (lambda (object)  
    (macro  
      (lambda (signal . args)  
        (cons object  
              (cons (list 'quote signal)  
                    args))))))
```

```
(define obj  
  (lambda s (if (eq? (car s) 'blah) 1 (list 2 s))))
```

```
(define obj:send (make-sender obj))  
(obj:send blah)  $\implies$  1  
(obj:send halb (+ 1 2) 'neco)  $\implies$  (2 (halb 3 neco))
```

## Interpret obohacený o generované symboly

- zavedeme nový typ elementu – *generovaný symbol*

*;; konstruktor symbolu a test zdali se jedná o symbol*

```
(define make-symbol (curry-make-elem 'symbol))  
(define scm-named-symbol? (curry-scm-type 'symbol))
```

*;; konstruktor generovaného symbolu*

```
(define make-generated-symbol  
  (let ((make-physical-element  
        (curry-make-elem 'generated-symbol)))  
    (lambda ()  
      (make-physical-element (cons #f #f)))))
```

- význam „(cons #f #f)“ z předchozího kódu:
  - nově vygener. symbol v sobě obsahuje ukazatel na nový pár (#f . #f)
  - každý generovaný symbol je proto `eq?`-roven pouze sám sobě

## Interpret obohacený o generované symboly

- v proceduře `scm-assoc`, která se stará o hledání vazeb v prostředí přidáme novou větev, která bude ošetřovat případ, kdy hledáme vazbu *vygenerovaného symbolu* (zde je potřeba použít `eq?`)

*;; hledání vazeb v asociačním poli*

```
(define scm-assoc
  (lambda (key alist)
    (cond ((scm-null? alist) scm-false)
          ((eq? key (pair-car (pair-car alist)))
           (pair-car alist))
          ((equal? key (pair-car (pair-car alist)))
           (pair-car alist))
          (else (scm-assoc key (pair-cdr alist))))))
```

## Interpret obohacený o generované symboly

*;; predikát testující zdali je daný element generovaný symbol*

```
(define scm-generated-symbol?  
  (curry-scm-type 'generated-symbol))
```

*;; test zdali se jedná o symbol pojmenovaný/generovaný*

```
(define scm-symbol?  
  (lambda (elem)  
    (or (scm-named-symbol? elem)  
        (scm-generated-symbol? elem))))
```

*;; generované symboly*

```
(gensym . ,(make-primitive make-generated-symbol))
```

## Příklady použití generovaných symbolů

```
(define while
  (macro
    (lambda (test . body)
      (define loop-name (gensym))
      '((lambda ()
          (define ,loop-name
            (lambda ()
              (if ,test
                  (begin
                     ,@body
                     (,loop-name))))))
          (,loop-name))))))
```

```
(define i 10)
(define x 0)
(while (> i 0) (set! i (- i 1)) (set! x (+ x i)))
(i x)  $\Rightarrow$  (0 45)
```