KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO, OLOMOUC

PARADIGMATA PROGRAMOVÁNÍ 2A
MAKRA III

Slajdy vytvořili Vilém Vychodil a Jan Konečný

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)
```

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)
```

Makro pro cyklus typu `for` (C, PERL a další)

Co chceme napodobit:

```
{
  int i = 0;
  int result = 0;

  for (i = 5; i > 0; i--)
  {
    printf("Stav: %i %i\n", i, result);
    result++;
  }
  printf("Koncovy: %i %i\n", i, result);
}
```

Pozn.: zatím nebudeme řešit `break` a `continue`.

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))  ⟼  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))))))
```

$\Longmapsto$ `(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))  ⟹  ((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 makroexpanze a vyhodnocování.

```
;; pomocná procedura
(define proc
  (lambda (x) (list '- x)))

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

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

KVAZIKVOTOVÁNÍ – úkolem je vyrobit makro realizující kvazikvotování

```
(kvaziquote blah)
  ⇓
(quote blah)  ⟹  blah

(kvaziquote (a b))
  ⇓
(apply append (list (quote a)) (list (quote b))
              (quote ()))  ⟹  (a b)

(kvaziquote (a (unquote (+ 1 2))))
  ⇓
(apply append (list (quote a)) (list (+ 1 2)) (quote ()))

(kvaziquote (a (unquote-splicing l)))
  ⇓
(apply append (list (quote a)) l (quote ()))  ⟹ ···
```

```
;; 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)                      ⟼  (list (quote 1))
(te '())                    ⟼  (list (quote ()))
(te '(1 2 3))               ⟼  (list (kvaziquote (1 2 3)))
(te '(unquote (1 2 3)))     ⟼  (list (1 2 3))
(te '(unquote-splicing (1 2 3)))  ⟼  (1 2 3)
(te '(kvaziquote (1 2)))  ⟼  (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 složitě 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*▷ ◁*pravidlo*$_1$▷ ◁*pravidlo*$_2$▷ ...)
- ◁*klicova-slova*▷ ... seznam symbolů, které jsou dále chápány jako klíčová slova (seznam může být prázdný)
- ◁*pravidlo*$_n$▷ ... 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.

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

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` $\Longmapsto$ `(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**
    $stmt_1$
    $stmt_2$
    ⋮
    $stmt_n$
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)` $\longmapsto$ nedefinovaná hodnota

`(f 4)` $\longmapsto$ 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)` $\longmapsto$ nedefinovaná hodnota

`(f 4)` $\longmapsto$ 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))  ⟹  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))  ⟹  3
```