

KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO, OLMOUC

PARADIGMATA PROGRAMOVÁNÍ 2

KORUTINY, NEDETERMINISMUS

Slajdy vytvořili Vilém Vychodil a Jan Konečný

Cykly vybavené „break“, „continue“ a „redo“

- `break` ... přerušení cyklu a návrat s danou hodnotou
- `continue` ... provedení další iterace (přeskočení zbytku těla)
- `redo` ... skok na začátek těla cyklu (bez testu podmínky)

;; *while* vybavený *break*

```
(define-macro while
  (lambda (condition . body)
    (let ((loop-name (gensym)))
      '(call/cc
        (lambda (break)
          (let ,loop-name ()
            (if ,condition
                (begin ,@body
                       (,loop-name))))))))))
```

```
(let ((i 0)
      (j 0))
  (while (< i 10)
    (set! j (+ j i))
    (set! i (+ i 1))
    (if (> i 5)
        (break))))
(list i j))  $\implies$  (6 15)
```

```
(let ((n 0) (i 10))
  (while (>= i 0)
    (set! i (- i 1))
    (let ((j i))
      (while (>= j 0)
        (set! n (+ n j))
        (break))))))
```

n) \implies 45

;; while vybavený break a continue

```
(define-macro while
  (lambda (condition . body)
    (let ((loop-name (gensym)))
      '(call/cc
        (lambda (break)
          (let ,loop-name ()
            (if ,condition
                (begin (call/cc
                        (lambda (continue)
                          ,@body))
                       (,loop-name))))))))))
```

Příklad použití:

```
(let ((i 0)
      (j 0))
  (while (< i 10)
    (if (<= i 2)
        (begin
          (set! i (+ i 1))
          (continue)))
        (set! j (+ j i))
        (set! i (+ i 1))
        (if (> i 5)
            (break))))
(list i j))  $\Rightarrow$  (6 12)
```

;; *while* vybavený *break*, *continue* a *redo* (PERL)

```
(define-macro while
  (lambda (condition . body)
    (let ((loop-name (gensym)))
      '(call/cc
        (lambda (break)
          (let ,loop-name ()
            (if ,condition
                (begin (call/cc
                        (lambda (continue)
                          (let ((redo #f))
                            (call/cc
                              (lambda (f)
                                (set! redo f))))
                              ,@body))))
              (,loop-name))))))))))
```

Příklad použití:

```
(let ((i 0) (j 0))
  (while (< i 10)
    (set! j (+ j i))
    (set! i (+ i 1))
    (if (and (>= i 10) (< i 20))
        (redo))))
(list i j) ⇒ (20 190)
```

Pro `continue` místo `redo` bychom dostali:

```
(let ((i 0) (j 0))
  (while (< i 10)
    (set! j (+ j i))
    (set! i (+ i 1))
    (if (and (>= i 10) (< i 20))
        (continue))))
(list i j) ⇒ (10 45)
```

```

(define-macro do
  (lambda (binding condition . body)
    (let ((loop-name (gensym)))
      '(call/cc
        (lambda (break)
          (letrec
            ((,loop-name
              (lambda ,(map car binding)
                (if ,(car condition)
                    (begin ,@(cdr condition))
                    (begin (call/cc
                          (lambda (continue)
                            (let ((redo #f))
                              (call/cc (lambda (f)
                                      (set! redo f)))
                                ,@body))))
              (,loop-name ,@(map caddr binding)))))))
      (,loop-name ,@(map cadr binding))))

```


Makro realizující cyklus typu `repeat ~ until`
(již jsme implementovali v předchozích lekcích)

```
(define-macro repeat
  (lambda (args)
    (letrec ((but-last ... ; viz přednášku 4
              (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)))))))
```

Makro obohacené o `break`, `continue` a `redo`

```
(define-macro repeat
  (lambda (args)
    (letrec (... ; vazby jako na předchozím slajdu
      '(call/cc
        (lambda (break)
          (let (,loop-name ())
            (call/cc
              (lambda (continue)
                (let ((redo #f))
                  (call/cc (lambda (f) (set! redo f)))
                  ,@body))))
            (cond (,@(map (lambda (conds)
                          '(,(car conds)
                            (begin ,@(cdr conds))))
                        (cdr limits))
                  (else (,loop-name))))))))))
```

Iterátory (opakování)

- iterátor – pro danou datovou strukturu postupně vrací její prvky

Pomocné definice:

;; identifikátor ukončení iterace

```
(define *end-of-iteration* (lambda () #f))
```

;; implementace chybového hlášení

;; predikát indikující konec iterace

```
(define finished?  
  (lambda (elem)  
    (eq? elem *end-of-iteration*)))
```

```

(define generate-iterator
  (lambda (l)
    (letrec ((return #f)
              (start
               (lambda ()
                 (let loop ((l l))
                   (if (null? l)
                       (return *end-of-iteration*)
                       (begin
                        (call/cc
                         (lambda (new-start)
                           (set! start new-start)
                           (return (car l))))
                        (loop (cdr l))))))))))
    (lambda () (call/cc
                 (lambda (f)
                   (set! return f)
                   (start)))))))

```

Příklad použití:

```
(define p (generate-iterator '(a b c d e)))
```

(p) \Rightarrow a

(p) \Rightarrow b

(p) \Rightarrow c

(p) \Rightarrow d

(p) \Rightarrow e

(p) \Rightarrow #<procedura> (indikátor konce)

(eq? (p) *end-of-iteration*) \Rightarrow #t

Příklad: Hlubkový iterátor

```
(define generate-depth-iterator
  (lambda (l)
    (letrec ((return #f)
              (start
               (lambda ()
                 (let loop ((l l))
                   (cond ((null? l) 'nejaka-hodnota)
                         ((pair? l)
                          (begin
                           (loop (car l))
                           (loop (cdr l))))
                         (else
                          (call/cc
                           (lambda (new-start)
                             (set! start new-start)
                             (return l)))))))
               (return '())))) ...
```

```
⋮  
(lambda ()  
  (call/cc  
    (lambda (f)  
      (set! return f)  
      (start))))))
```

Poznámka:

- už není potřeba `*end-of-iteration*`
- prohledávání je ukončeno `()`

Příklad použití:

```
(define p (generate-depth-iterator '(a (b (c (d)) e))))  
(define q (generate-depth-iterator '(((a b) ((c d))) e)))
```

(p) \Rightarrow a

(q) \Rightarrow a

(p) \Rightarrow b

(q) \Rightarrow b

(p) \Rightarrow c

(q) \Rightarrow c

(p) \Rightarrow d

(q) \Rightarrow d

(p) \Rightarrow e

(q) \Rightarrow e

(p) \Rightarrow ()

(q) \Rightarrow ()

Korutiny

- korutiny ... reprezentují podprogramy, které se vzájemně přepínají
- C. T. Haynes, D. P. Friedman, M. Wand. Continuations and Coroutines. In: *Conf. ACM Symp. LISP and Functional Programming*, 293–298, 1984.

;; makro na vytváření korutin

```
(define-macro coroutine
  (lambda (arg . body)
    '(letrec ((state (lambda ,arg ,@body))
              (resume
               (lambda (c . elems)
                 (call/cc
                  (lambda (f)
                    (set! state f)
                    (apply c elems)))))))
      (lambda elems
        (apply state elems))))))
```

Příklad: bez použití `resume` se chová jako *normální procedura*

```
(define c (coroutine (x) (+ x 1)))  
(c 10)  $\implies$  11
```

Příklad: dvě korutiny, jedna přepne na druhou

```
(define d (coroutine ()  
            (display "VOLANA D")  
            (newline)  
            (resume c 10)))
```

(d) \implies 11 a vypíše VOLANA d)

(d) \implies nic

(d 10) \implies 10

(d 'blah) \implies blah

Iterátor pomocí korutin

;; *identifikátor ukončení iterace a predikát (již máme implementované)*

```
(define *end-of-iteration* (lambda () #f))  
(define finished?  
  (lambda (elem)  
    (eq? elem *end-of-iteration*)))
```

;; *iterátor: korutina volající další korutinu (caller)*

```
(define generate-iterator  
  (lambda (l)  
    (coroutine (caller)  
      (let loop ((l l))  
        (if (null? l)  
            (resume caller *end-of-iteration*)  
            (begin  
              (resume caller (car l))  
              (loop (cdr l))))))))))
```

- K tomu abychom mohli používat iterátor potřebujeme vytvořit další korutinu (zde se jmenuje „`user`“, v kódu iterátoru to je „`caller`“).

```
(letrec ((iterator (generate-iterator '(a b c d e)))
         (user
          (coroutine ()
            (let iter ()
              (let ((v (resume iterator user)))
                (if (not (finished? v))
                    (begin
                     (display v)
                     (newline)
                     (iter))))))))))
(user))
```

- Předchozí bychom mohli zjednodušit makrem `with-iterator`.

Nedeterminismus

- nedeterministický operátor `amb`, vrací jeden z vyhodnoc. argumentů, přitom mu jde o to *najít aspoň jedno řešení* (operátor nedeterministicky konverguje k řešení, pokud existuje)
- J. McCarthy. A Basis for a Mathematical Theory of Computation. In: P. Braffort, D. Hirschberg (Eds.): *Computer Programming and Formal Systems*. North-Holland, 1967.

```
(amb)            $\implies$  Error: Tree Exhausted
```

```
(amb 1 2 3 4)   $\implies$  1
```

```
(if (amb #f #t)  
    'blah  
    (amb))  $\implies$  blah
```

```
(let ((x (amb 1 2 3 4)))  
    (if (odd? x)  
        (amb)  
        x))  $\implies$  2
```

Nedeterminismus

- `amb-fail`, pomocná procedura, její účel bude jasný dále
- bez argumentu: slouží k vyvolání návratu (*backtracking*)
- 1 argument: nastaví hodnotu aktuálního návratu na danou hodnotu

```
(define amb-fail
  (let ((failure #f))
    (lambda args
      (if (null? args)
          failure
          (set! failure
                 (if (procedure? (car args))
                     (car args)
                     (lambda ()
                       (error "AMB: Tree Exhausted"))))))))

;; inicializace amb-fail
(amb-fail #f)
```

O co tady jde?

```
(amb-fail #f)
```

```
(amb-fail)            $\Rightarrow$   $\langle (), (\text{error } \dots), \mathcal{P} \rangle$ 
```

```
(amb-fail odd?)
```

```
(amb-fail)            $\Rightarrow$  primitivní procedura odd?
```

```
(amb-fail (lambda (x)
            (amb-fail odd?)
            x))
```

```
((amb-fail) 10)       $\Rightarrow$  10
```

```
((amb-fail) 10)       $\Rightarrow$  #f
```

```
((amb-fail) 11)       $\Rightarrow$  #t
```

```
(amb-fail #f)
```

```
(let ((prev-fail (amb-fail)))  
  (call/cc (lambda (exit)  
            (call/cc (lambda (next)  
                      (amb-fail (lambda ()  
                                  (amb-fail prev-fail)  
                                  (next))))  
                    (exit 1))))  
  (prev-fail))))  $\implies$  1
```

```
((amb-fail))  $\implies$  „CHYBA: AMB: Tree Exhausted“
```



```
(amb-fail #f)
```

```
(let ((prev-fail (amb-fail)))  
  (call/cc (lambda (exit)  
    (call/cc (lambda (next)  
      (amb-fail (lambda ()  
                  (amb-fail prev-fail)  
                  (next))))  
      (exit 1)))  
    (call/cc (lambda (next)  
      (amb-fail (lambda ()  
                  (amb-fail prev-fail)  
                  (next))))  
      (exit 2)))  
    (prev-fail))))))  $\Longrightarrow$  1
```

```
((amb-fail))  $\Longrightarrow$  2
```

```
((amb-fail))  $\Longrightarrow$  „CHYBA: AMB: Tree Exhausted“
```

```

next →
(excaper
  (lambda (□)
    (let ((prev-fail (amb-fail)))
      (call/cc (lambda (exit)
                 □
                 (call/cc (lambda (next)
                           (amb-fail (lambda ()
                                       (amb-fail prev-fail)
                                       (next))))
                           (exit 2))))
      (prev-fail))))))

```

```

exit →
(excaper
  (lambda (□)
    (let ((prev-fail (amb-fail)))
      □)))

```

`failure` ($v \mathcal{P}$) \rightarrow $\langle (), (\text{amb-fail prev-fail}) (\text{next}), \mathcal{P}_x \rangle$

Tedy procedura při aplikaci

- uloží „předchozí fail“ ve vazbě na `failure` v \mathcal{P} ,
- aktivuje `next` – pokračování dalším prvkem.

;; *amb* (ambiguous) operátor

```
(define-macro amb
  (lambda (elems
    (let ((previous-fail (gensym)))
      `(let ((,previous-fail (amb-fail)))
        (call/cc
          (lambda (exit)
            ,@(map (lambda (elem)
                    `(call/cc
                     (lambda (next)
                       (amb-fail
                        (lambda ()
                          (amb-fail ,previous-fail)
                          (next))))
                     (exit ,elem))))))
          elems)
        (,previous-fail))))))
```

```
(let ((elem (amb 1 2 3))) elem)  $\Rightarrow$  1
```

```
(let ((elem (amb 1 2 3 15 4)))  
  (if (< elem 10)  
      'xxx)  
  elem)  $\Rightarrow$  1
```

```
(let ((elem (amb 1 2 3 15 4)))  
  (if (< elem 10)  
      (amb))  
  elem)  $\Rightarrow$  15
```

PROBOHA PROČ?

```

next →
(excaper
  (lambda (□)
    (let ((elem
          (let ((prev-fail (amb-fail)))
            (call/cc (lambda (exit)
                      □
                      (call/cc (lambda (next)
                                (amb-fail (lambda ()
                                           (amb-fail prev-fail)
                                           (next))))
                                (exit 2))))
          (prev-fail))))))
    (if (< elem 10)
        (amb))
    elem)))

```

;; odvozený konstrukt: *assert*

```
(define assert  
  (lambda (elem)  
    (if (not elem)  
        (amb))))
```

Příklad použití *assert* (spolu s *amb*)

```
(let ((x (amb 1 2 3 4 5)))  
  (assert (odd? x))  
  x)
```