Paradigmata programování 1

Kvazikvotování a manipulace se symbolickými výrazy

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 11

Přednáška 11: Přehled

1. Kvazikvotování
   - rozdíl mezi speciálními formami `quote` a `quasiquote`
   - vkládání hodnot pomocí `unquote` a `unquote-splicing`
   - syntaktický cukr pro kvazikvotování
2. Práce se symbolickými výrazy
   - zjednodušování aritmetických výrazů dvou argumentů
   - zjednodušovací tabulky
   - zjednodušování operací více argumentů
   - symbolické derivace
   - převod symbolických výrazů mezi různými notacemi
3. Vyhodnocování symbolických výrazů
   - postfixové vyhodnocování
   - zásobníkový model vyhodnocování bezzávorkových výrazů

Kvazikvotování

Speciální formy `quote` a `quasiquote`

- `quote` – vrací svůj argument bez vyhodnocení (známe),
- `quasiquote` – jako `quote`, ale u některých podvýrazů lze vynutit vyhodnocení

Příklad (identické chování `quote` a `quasiquote`)

```
(quasiquote 15)                ⟹  15
(quasiquote symbol)            ⟹  symbol
(quasiquote (x . 3))           ⟹  (x . 3)
(quasiquote (1 2 (3 4) 5))     ⟹  (1 2 (3 4) 5)
```

Syntaktický cukr:

- `(quote` $\langle expr \rangle$`)` $\equiv$ `'`$\langle expr \rangle$
- `(quasiquote` $\langle expr \rangle$`)` $\equiv$ `` ` ``$\langle expr \rangle$

Použití unquote a unquote-splicing

V kvazikvotovaných výrazech lze použít:

- `(unquote` $\langle expr \rangle$`)` – vyhodnotí $\langle expr \rangle$ a vloží hodnotu na dané místo
  `(unquote` $\langle expr \rangle$`)` $\equiv$ `,`$\langle expr \rangle$
- `(unquote-splicing` $\langle expr \rangle$`)` – vyhodnotí $\langle expr \rangle$ a vloží prvky *výsledného seznamu* na dané místo, jeden po druhém ($\langle expr \rangle$ se musí vyhodnotit na seznam)
  `(unquote-splicing` $\langle expr \rangle$`)` $\equiv$ `,@`$\langle expr \rangle$

Příklad (rozdíl mezi unquote a unquote-splicing)

```
`(1 2 ,(build-list 3 +) #f)    ⟹  (1 2 (0 1 2) #f)
`(1 2 ,@(build-list 3 +) #f)   ⟹  (1 2 0 1 2 #f)
`(1 2 ,(+ 2 3) #f)             ⟹  (1 2 5 #f)
`(1 2 ,@(+ 2 3) #f)            ⟹  chyba
```

Příklad (kvazikvotované seznamy a jejich vyhodnocení)

```
`(1 2 3 4)                ⟼  (1 2 3 4)
`(1 (+ 10 20) 3 4)        ⟼  (1 (+ 10 20) 3 4)
`(1 ,(+ 10 20) 3 4)       ⟼  (1 30 3 4)
`'(1 ,(+ 10 20) 3 4)      ⟼  (quote (1 30 3 4))

`(1 ,(append '(a b) '(c d) '(10 20)) 3 4)
  ⟼  (1 (a b c d 10 20) 3 4)
`(1 ,@(append '(a b) '(c d) '(10 20)) 3 4)
  ⟼  (1 a b c d 10 20 3 4)

`(1 2 3 4 5)              ⟼  (1 2 3 4 5)
`(1 (+ 2 3) 4 5)          ⟼  (1 (+ 2 3) 4 5)
`(1 ,(+ 2 3) 4 5)         ⟼  (1 5 4 5)
`(1 ((,(+ 2 3))) 4 5)     ⟼  (1 ((5)) 4 5)
```

Příklad (kvazikvotování – na co dát pozor)

```
(define s '(a b c))
`(1 ,s 2)    ⟹  (1 (a b c) 2)
`(1 ,@s 2)   ⟹  (1 a b c 2)

`(1 '2 3)    ⟹  (1 (quote 2) 3)
`'(1 2 3)    ⟹  (quote (1 2 3))
'`(1 2 3)    ⟹  (quasiquote (1 2 3))

`(1 `(2 3))          ⟹  (1 (quasiquote (2 3)))
`(1 `(,(+ 1 2) 3))   ⟹  (1 (quasiquote ((unquote (+ 1 2)) 3)))
`(1 ,`(,(+ 1 2) 3))  ⟹  (1 (3 3))
```

- kvazikvotování lze vnořovat
- `unquote` a `unquote-splicing` má význam pouze na „stejné úrovni vnoření“

Intermezzo: Asociační seznamy

Seznam je **asociační seznam** pokud je ve tvaru:

$$((\langle klíč_1 \rangle \,.\, \langle hodnota_1 \rangle)\ (\langle klíč_2 \rangle \,.\, \langle hodnota_2 \rangle)\ \cdots\ (\langle klíč_n \rangle \,.\, \langle hodnota_n \rangle))$$

Základní selektor `assoc` pro práci s asociačním seznamem:

```
(define s (list (cons 'a 10) (cons 'b 20) (cons 'c 30)))
(assoc 'b s)  ⟹  (b . 20)
(assoc 'a s)  ⟹  (a . 10)
(assoc 'c s)  ⟹  (c . 30)
(assoc 'd s)  ⟹  #f
```

```
(define assoc
  (lambda (elem l)
    (cond ((null? l) #f)
          ((equal? (caar l) elem) (car l))
          (else (assoc elem (cdr l))))))
```

Práce se symbolickými výrazy

Motivační příklad – co dělá každý pořádný překladač

Uvažujeme symbolické výrazy kódující aritmetické výrazy s proměnnými.
Úkolem je napsat proceduru na zjednodušování výrazů, např.:
`(+ y (/ (+ 2 x) (+ 1 x (* 2 0.5))))` se zjednoduší na `(+ 1 y)`

Jak naučit počítač symbolicky upravovat výraz?

- procedura `simplify` použitelná ve tvaru `(simplify` $\langle výraz \rangle$`)`
- rekurzivní charakter problému:
  - pokud je výraz atom, nezjednodušujeme
  - pokud je výraz seznam `(`*op* `a b c` $\cdots$ `x)`, pak:
    - pak nejprve zjednodušíme argumenty (vede na rekurzivní aplikaci `simplify`)
    - potom využijeme známé zákony a pokusíme se zkrátit výsledek
      $(x \cdot 0 = 0, x \cdot 1 = x, x + x = 2x, \ldots)$

Jednoduchá verze `simplify`

Zjednodušuje výrazy s omezením:

- aritmetické výrazy mohou mít nejvýš dva argumenty
- zjednodušuje pevně zabudovanou množinu operací: $+$, $\times$, $-$, $\div$

```
(define simplify
  (lambda (expr)

    ;; zjisti, jestli jsou oba argumenty stejná čísla
    (define ==
      (lambda (x y)
        (and (number? x) (number? y) (= x y))))

    ;; ošetření dělení nulou
    (define div-by-zero
      (lambda () 'division-by-zero))
        ⋮
```

```
;; zjednodušení pro sčítání
(define simplify-add
  (lambda (op x y)
    (cond ((== x 0) y)                    ; 0 + y = y
          ((== y 0) x)                    ; x + 0 = x
          ((equal? x y) `(* 2 ,x))        ; x + x = 2 × x
          (else (list op x y)))))

;; zjednodušení pro násobení
(define simplify-mul
  (lambda (op x y)
    (cond ((== x 0) 0)                    ; 0 × y = 0
          ((== y 0) 0)                    ; x × 0 = 0
          ((== x 1) y)                    ; 1 × y = y
          ((== y 1) x)                    ; x × 1 = x
          (else (list op x y)))))
```

```
;; zjednodušení pro odčítání
(define simplify-min
  (lambda (op x y)
    (cond ((== x 0) (list op y))          ; 0 − x = −x
          ((== y 0) x)                    ; x − 0 = x
          ((equal? x y) 0)                ; x − x = 0
          (else (list op x y)))))
```

```
;; zjednodušení pro dělení
(define simplify-div
  (lambda (op x y)
    (cond ((== x 0) 0)                    ; 0 ÷ x = 0
          ((== y 0) (div-by-zero))        ; x ÷ 0  nelze
          ((== x 1) (list op y))          ; 1 ÷ x = 1/x
          ((== y 1) x)                    ; x ÷ 1 = x
          ((equal? x y) 1)                ; x ÷ x = 1
          (else (list op x y)))))
```

```
  ⋮
(cond
 ((number? expr) expr)
 ((symbol? expr) expr)
 ((and (list? expr) (member (car expr) '(+ * - /)))
  (let ((op (car expr))
        (expr (map simplify expr)))
    (if (null? (cddr expr))                ⟵ případ jednoho argumentu
        (if (number? (cadr expr)) (eval expr) expr)
        (let ((x (cadr expr))
              (y (caddr expr)))
          (cond ((and (number? x) (number? y)) (eval expr))
                ((equal? op '+) (simplify-add op x y))
                ((equal? op '*) (simplify-mul op x y))
                ((equal? op '-) (simplify-min op x y))
                (else (simplify-div op x y)))))))))
```

Příklad (použití jednoduché verze `simplify`)

Kde funguje uspokojivě:

```
(simplify '(+ 10))                  ⟹  10
(simplify '(+ 1 2))                 ⟹  3
(simplify '(- x 0))                 ⟹  x
(simplify '(- 0 x))                 ⟹  (- x)
(simplify '(/ 0 x))                 ⟹  0
(simplify '(/ x 0))                 ⟹  division-by-zero
(simplify '(/ x 1))                 ⟹  x
(simplify '(/ 1 x))                 ⟹  (/ x)
(simplify '(/ (+ 2 x) (* 2 0.5)))   ⟹  (+ 2 x)
```

Kde lze ještě vylepšit:

```
(simplify '(+ (* 2 3) (+ y (+ 2 x))))  ⟹  (+ 6 (+ y (+ 2 x)))
```

Tabulka zjednodušování – elegantní řešení

```
(define simplification-tbl
  ;; pomocné procedury == a div-by-zero
  (let ((== (lambda (x y)
              (and (number? x) (number? y) (= x y))))
        (div-by-zero (lambda () 'division-by-zero)))
    ;; asociační tabulka symbol ~ zjednodušovací procedura
    `((+ . ,(lambda (op x y)
              (cond ((== x 0) y)
                    ((== y 0) x)
                    ((equal? x y) `(* 2 ,x))
                    (else (list op x y)))))
      (* . ,(lambda (op x y)
              (cond ((== x 0) 0)
                    ⋮
```

Zjednodušovací procedura s tabulkou jako parametrem

```
(define simplify
  (lambda (expr table)
    (let simplify ((expr expr))
      (cond
       ((number? expr) expr)
       ((symbol? expr) expr)
       ((and (list? expr) (assoc (car expr) table))
        (let ((expr (map simplify expr)))
          (if (null? (cddr expr))
              (if (number? (cadr expr)) (eval expr) expr)
              (if (forall number? (cdr expr))
                  (eval expr)
                  (apply (cdr (assoc (car expr) table))
                         expr)))))))))
```

Verze pro + a × s libovolně mnoha argumenty

Myšlenka dalšího zjednodušování:

- všechna čísla ze seznamu posčítáme/vynásobíme (argument `value`)
- ostatní (složené) hodnoty ze seznamu dáme do seznamu (argument `compound`)

```
(define simplify-addmul
  (lambda (op compound value)
    (cond ((and (equal? op '*) (= value 0)) 0)
          ((null? compound) value)
          ((= value (eval `(,op)))          ⟵ případ neutrálního prvku
           (if (null? (cdr compound))
               (car compound)
               (cons op compound)))
          (else `(,op ,value ,@compound)))))
```

```
(define simplification-tbl
  (let ((== (lambda (x y)
              (and (number? x) (number? y) (= x y))))
        (div-by-zero (lambda () 'division-by-zero))
        (simplify+*
         (lambda (op . rest)
           (let ((value (apply (eval op)
                               (filter number? rest)))
                 (compound (remove number? rest)))
             (simplify-addmul op compound value)))))
    `((+ . ,simplify+*)
      (* . ,simplify+*)
      (- . ,(lambda (op x y)
              (cond ((== x 0) (list op y))
                    ((== y 0) x)
                       ⋮
```

Příklad (použití rozšířené verze `simplify`)

Nové možnosti zjednodušování:

```
(simplify '(+ 1 2 x 3 y 5 6))                ⟹ (+ 17 x y)
(simplify '(+ (* 2 3) y (+ 2 x)))            ⟹ (+ 6 y (+ 2 x))
(simplify '(- (* 1 x) (+ 2 -3 x 1)))         ⟹ 0
(simplify '(/ (+ 2 x) (+ 1 x (* 2 1/2))))    ⟹ 1
  ⋮
```

Pořád není ideální:

```
(simplify '(+ x (- x)))          ⟹  (+ x (- x))
(simplify '(* x (/ 1 x)))        ⟹  (* x (/ x))
(simplify '(+ 1 (+ 1 (+ 1 x))))  ⟹  (+ 1 (+ 1 (+ 1 x)
(simplify '(+ x x x x))          ⟹  (+ x x x x)
  ⋮
```

Práce se symbolickými výrazy

Motivační příklad – symbolické derivace

Uvažujeme symbolické výrazy kódující aritmetické výrazy s proměnnými.
Úkolem je napsat proceduru, která pro daný výraz vrací jeho derivaci, např.:
na základě výrazu `(* x x)` a proměnné `x` vracíme `(* 2 x)`

Jak naučit počítač symbolicky derivovat?

- naimplementujeme tabulku pravidel pro derivování

$$c' = 0,$$
$$x' = 1,$$
$$(f \pm g)' = f' \pm g',$$
$$(f \cdot g)' = f' \cdot g + g' \cdot f, \ldots$$

- rekurzivně aplikujeme pravidla
- na výsledek použijeme zjednodušování – čitelnější výstup

Implementace symbolické derivace

Vytvoříme proceduru `diff`:

- dva formální argumenty: `expr` (vstupní výraz), `var` (proměnná)
- procedura `diff` používá lokálně definované vazby:
  - `variable?` – predikát (je daný výraz proměnná?)
  - `constant?` – predikát (je daný výraz konstanta?)
  - `table` – tabulka pravidel pro derivování
  - `derive` – rekurzivní procedura pro vlastní výpočet

Schématicky:

```
(define diff
  (lambda (expr var)
      ⋮
    (derive expr))
```

Interní predikáty `variable?` a `constant?`

```
;; testuj, jestli je daný výraz proměnná
(define variable?
  (lambda (expr)
    (equal? expr var)))

;; testuj, jestli je daný výraz konstanta
(define constant?
  (lambda (expr)
    (or (number? expr)
        (and (symbol? expr)
             (not (variable? expr))))))
```

Poznámka:

- vazba symbolu `var` se bere z prostředí vzniku procedur (argument `diff`)

Interní tabulka pravidel pro derivování

;; asociační seznam symbol $\sim$ pravidlo

```
(define table
  `((+ . ,(lambda (x y)
            `(+ ,(derive x) ,(derive y))))
    (- . ,(lambda (x y)
            `(- ,(derive x) ,(derive y))))
    (* . ,(lambda (x y)
            `(+ (* ,x ,(derive y))
                (* ,(derive x) ,y))))
    (/ . ,(lambda (x y)
            `(/ (- (* ,(derive x) ,y)
                   (* ,x ,(derive y)))
                (* ,y ,y))))))
```

Poznámka:

- procedura navázaná na `derive` zatím není definována (**!!**)

Interní tabulka pravidel pro derivování

```
;; derivuj vstupní výraz
(define derive
  (lambda (expr)
    (cond ((variable? expr) 1)
          ((constant? expr) 0)
          (else (apply (cdr (assoc (car expr) table))
                       (cdr expr))))))
```

Poznámky:

- vazba symbolu `var` se bere z prostředí vzniku procedur (argument `diff`)
- spuštění `(derive expr)` s výchozím vstupním výrazem
- výstup lze zjednodušovat pomocí `simplify`

Příklad (použití `diff` a `simplify`)

```
(diff '(+ x x) 'x)  ⟹  (+ 1 1)
(diff '(* x y) 'x)  ⟹  (+ (* x 0) (* 1 y))

(diff '(+ (* x 2) (* x x)) 'x)
  ⟹  (+ (+ (* x 0) (* 1 2)) (+ (* x 1) (* 1 x)))
(diff '(* x (* x x)) 'x)
  ⟹  (+ (* x (+ (* x 1) (* 1 x))) (* 1 (* x x)))

(simplify (diff '(+ x x) 'x))  ⟹  2
(simplify (diff '(* x y) 'x))  ⟹  y

(simplify (diff '(+ (* x 2) (* x x)) 'x))
  ⟹  (+ (+ x x) 2)
(simplify (diff '(* x (* x x)) 'x))
  ⟹  (+ (* x (+ x x)) (* x x))
```

Příklad (vytvoření funkce derivace pomocí symbolického předpisu)

```
(define diff-proc
  (lambda (expr var)
    (eval
     `(lambda (,var)
        ,(simplify (diff expr var))))))

(define diff-proc
  (lambda (expr var)
    (eval
     `(lambda (,var)
        ,(simplify (diff expr var)))
     (environment-parent (the-environment)))))
```

Příklad (Notace zápisu aritmetických výrazů)

- **infixová** – symbol pro operace je mezi operandy („běžná notace“)
  - plusy: snadno se čte
  - minusy: asociativita, priorita, operace pouze dva argumenty

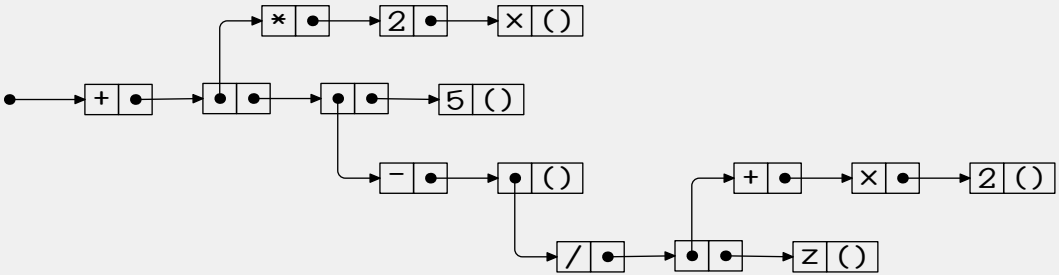
  Příklad: `2 * (3 + 5)`
- **prefixová** – symbol pro operace před operandy
  - plusy: libovolný počet operandů, odpadají problémy s asociativitou / prioritou
  - minusy: nezvyk, velké množství závorek

  Příklad: `(* 2 (+ 3 5))` (vynásob dvojku se součtem trojky a pětky)
- **postfixová** – symbol pro operace za operandy

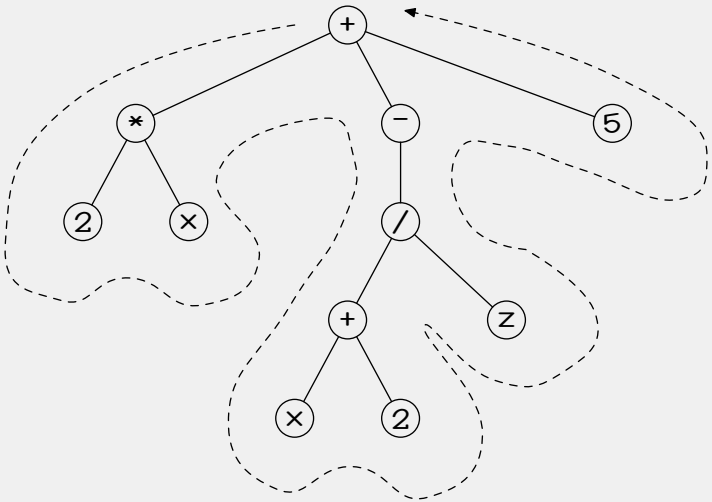
  Příklad: `(2 (3 5 +) *)`
- **postfixová bezzávorková (polská)**
  - plusy: strojově snadno analyzovatelná
  - minusy: nejméně čitelná, operace mají fixní počet operandů

  Příklad: `2 3 5 + *`

Příklad (struktura výrazu `(+ (* 2 x) (- (/ (+ x 2) z)) 5)`)



Příklad (struktura výrazu `(+ (* 2 x) (- (/ (+ x 2) z)) 5)`)



Příklad (prefixová notace převedená na postfixovou)

```
(define prefix->postfix
  (lambda (S-expr)
    (cond ((number? S-expr) S-expr)
          ((symbol? S-expr) S-expr)
          ((null? S-expr) S-expr)
          ((list? S-expr)
           `(,@(map prefix->postfix (cdr S-expr))
             ,(car S-expr)))
          (else "Incorrect expression."))))
```

```
(prefix->postfix '(* 2 3))           ⟹  (2 3 *)
(prefix->postfix '(* (+ 2 3) 4))     ⟹  ((2 3 +) 4 *)
(prefix->postfix '(cons (+ 2 3) 4))  ⟹  ((2 3 +) 4 cons)
```

Příklad (prefixová notace převedená na bezzávorkovou)

```
(define prefix->polish
  (lambda (S-expr)
    (cond ((number? S-expr) (list S-expr))
          ((symbol? S-expr) (list S-expr))
          ((null? S-expr) (list S-expr))
          ((list? S-expr)
           `(,@(apply append
                      (map prefix->polish (cdr S-expr)))
             ,(car S-expr)))
          (else "Incorrect expression."))))
```

```
(prefix->polish '(* 3 5))          ⟹  (3 5 *)
(prefix->polish '(+ 2 (* 3 5)))    ⟹  (2 3 5 * +)
(prefix->polish '(+ (* 2 3) 5))    ⟹  (2 3 * 5 +)
```

Příklad (prefixová notace převedená na infixovou)

```
(define prefix->infix
  (lambda (S-expr)
    (if (pair? S-expr)
        (let ((op (car S-expr)) (tail (cdr S-expr)))
          (cond ((null? tail) (eval S-expr))
                ((null? (cdr tail))
                 `(,op ,(prefix->infix (car tail))))
                (else
                 (foldl (lambda (expr collected)
                          (list collected op expr))
                        (prefix->infix (car tail))
                        (map prefix->infix (cdr tail))))))
        S-expr)))
```

Příklad (příklady převodu do infixové notace)

```
(prefix->infix '(*))                  ⟼  1
(prefix->infix '(- 2))                ⟼  (- 2)
(prefix->infix '(- 2 3))              ⟼  (2 - 3)
(prefix->infix '(- 2 3 4))            ⟼  ((2 - 3) - 4)
(prefix->infix '(- (/ (+ x 2) z)))    ⟼  (- ((x + 2) / z))
```

Poznámky:

- převod `infix->prefix` je komplikovanější, musíme řešit:
  - priority operací: $2 \times 3 + 4 = (2 \times 3) + 4$ a nikoliv $2 \times (3 + 4)$,
  - asociativitu operací: $2 - 3 - 4 = (2 - 3) - 4$ a nikoliv $2 - (3 - 4)$.
- více: kursy *Formální jazyky a automaty, Překladače I, II*

Vyhodnocování postfixových výrazů

Princip

- jako u vyhodnocování ve Scheme
- operace je na konci seznamu
- nelze (bez dalších koncepčních úprav) zavést speciální formy

Algoritmus vyhodnocení:

- číslo se vyhodnotí na číslo
- symbol se vyhodnotí na svou vazbu
- je-li výraz seznam, potom:
  1. vyhodnotí se jeho prvky jeden po druhém,
  2. až se vyhodnotí poslední z nich, pak se ověří, jestli se vyhodnotil na proceduru,
  3. pokud ano, dojde k aplikaci procedury, ...

Příklad (průběh vyhodnocení výrazu v postfixové notaci)

```
(postfix-eval '((10 20 cons) (30 40 cons) list))
```

aktuální výraz	*seznam argumentů*
`((10 20 cons) (30 40 cons) list)`	`()`
`(10 20 cons)`	`()`
`(20 cons)`	`(10)`
`(cons)`	`(10 20)`
`((30 40 cons) list)`	`((10 . 20))`
`(30 40 cons)`	`()`
`(40 cons)`	`(30)`
`(cons)`	`(30 40)`
`(list)`	`((10 . 20) (30 . 40))`

$\longmapsto$ `((10 . 20) (30 . 40))`

Příklad (evaluátor postfixových výrazů)

```
(define postfix-eval
  (lambda (expr)
    (cond
     ((number? expr) expr)
     ((symbol? expr) (eval expr))
     (else
      (let proc ((expr expr) (args '()))
        (cond ((null? expr) '())
              ((null? (cdr expr))
               (apply (postfix-eval (car expr))
                      (reverse args)))
              (else (proc (cdr expr)
                          (cons (postfix-eval (car expr))
                                args)))))))))
```

Vyhodnocování výrazů v bezzávorkové notaci

Princip vyhodnocování:

- vyhodnocuje se pomocí dodatečného zásobníku (odložiště výsledků operací)
- nutná podmínka: operace mají pevně danou aritu (počet operandů)

Prostředí (vazby symbolů a arita procedur):

```
(define environment
  `((+ 2 ,+)
    (- 2 ,-)
    (/ 2 ,/)
    (* 2 ,*)
    (n 1 ,-)
    (^ 2 ,expt)))
```

Kde má uplatnění:

- embedded zařízení, malé systémy, tiskárny (PostScript), FORTH, ...

Algoritmus vyhodnocování pomocí zásobníku

Tři základní pravidla:

1. **atom** na začátku (dosud nezpracovaná část výrazu) přesuň na zásobník
2. je-li na začátku (jméno) **operace** s aritou n, pak:
   - vyzvedni n argumentů ze zásobníku,
   - proveď operaci s vyzvednutými argumenty,
   - ulož výsledek operace na vrchol zásobníku.
3. pokud je **vstup prázdný**, stav zásobníku je výsledek výpočtu

```
(polish-eval '(10 20 +) environment)
```

vstup	*zásobník*
(10 20 +)	()
(20 +)	(10)
(+)	(20 10)
()	(30)

Příklad (rozdíly ve vyhodnocování)

vstup	*zásobník*
`(10 20 30 + *)`	`()`
`(20 30 + *)`	`(10)`
`(30 + *)`	`(20 10)`
`(+ *)`	`(30 20 10)`
`(*)`	`(50 10)`
`()`	`(500)`

- `(10 20 30 + *)`
- $(20 + 30) \times 10$

vstup	*zásobník*
`(10 20 + 30 *)`	`()`
`(20 + 30 *)`	`(10)`
`(+ 30 *)`	`(20 10)`
`(30 *)`	`(30)`
`(*)`	`(30 30)`
`()`	`(900)`

- `(10 20 + 30 *)`
- $(10 + 20) \times 30$

```
(define polish-eval
  (lambda (expr env)
    (let iter ((input expr) (stack '()))
      (if (null? input)
        stack
        (let ((word (car input)) (tail (cdr input)))
          (if (not (symbol? word))
            (iter tail (cons word stack))
            (let ((func (assoc word env)))
              (if (not func)
                (error "Symbol not bound")
                (let ((arity (cadr func)) (proc (caddr func)))
                  (iter tail
                    (cons (apply proc
                      (reverse (list-pref arity stack))) ⟵
                      (list-tail stack arity)))))))))))
```