KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 11: Kvazikvotování a manipulace se symbolickými výrazy

Učební materiál k přednášce 14. prosince 2006
(pracovní verze textu určená pro studenty)

JAN KONEČNÝ, VILÉM VYCHODIL

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 2006

Lekce 11: Kvazikvotování a manipulace se symbolickými výrazy

Obsah lekce: Tato lekce obsahuje několik klasických příkladů zpracování symbolických dat se kterými se lze setkat i v jiné v literatuře. Nejprve se budeme zabývat kvazikvotováním, což je obecnější metoda kvotování umožňující programátorům snadněji definovat některé seznamy. Dále se budeme zabývat následujícími okruhy problémů: zjednodušování aritmetických výrazů, symbolická derivace, konverze mezi symbolickými výrazy zapsanými v prefixové, infixové, postfixové a polské bezzávorkové notaci. Nakonec ukážeme metodu vyhodnocování výrazů v polské bezzávorkové notaci.

Klíčová slova: kvazikvotování, infixová notace, postfixová notace, polská bezzávorková notace.

11.1 Kvazikvotování

V sekci 4.5 jsme ukázali speciální formu `quote`, která vracela svůj argument v nevyhodnocené podobě. Nyní představíme zobecnění této speciální formy – `quasiquote`. Speciální forma `quasiquote`, v porovnání se speciální formou `quote`, umožňuje navíc určit podvýrazy jejího argumentu, které budou vyhodnoceny běžným způsobem. Bez tohoto určení funguje speciální forma `quasiquote` stejným způsobem jako speciální forma `quote`. Viz následující příklady:

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

Zatímco speciální forma `quote` „slepě“ vrací svůj argument, speciální forma `quasiquote` argument projde a vyhledá v něm podvýrazy, které jsou jednoprvkové seznamy začínající symbolem `unquote`. Tyto podvýrazy jsou nahrazeny vyhodnocením jejich druhého prvku. Nejlépe to bude vidět na příkladech:

```
(define x 3)
(quasiquote (unquote x))                ⟹ 3
(quasiquote (unquote (+ 1 2)))          ⟹ 3
(quasiquote (1 x (unquote (+ 1 x 2))))  ⟹ (1 x 6)
```

Příklad 11.1. (a) Někdy se symbol `unquote` nemusí nacházet viditelně na začátku seznamu. V následující ukázce tento symbol není prvním prvkem seznamu `(1 unquote (+ 1 2))`, ale až druhým. Tento seznam je tečkový pár `(1 . (unquote (+ 1 2)))`, jehož druhým prvkem je seznam začínající symbolem `unquote`:

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

je totéž jako

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

(b) Symbol `unquote` musí být prvním prvkem seznamu, nikoli jen páru. Navíc seznam, jehož první prvek je symbol `unquote`, musí obsahovat právě jeden další prvek. Následující výrazy neobsahují správné použití symbolu `unquote` v argumentu speciální formy `quasiquote`:

```
(quasiquote (1 (unquote . 2)))
(quasiquote (1 (unquote)))
(quasiquote (1 (unquote 2 3)))
```

Výsledky vyhodnocení těchto výrazů standard jazyka Scheme R^{5}RS neurčuje. Naopak přímo říká, že v takových případech dochází k nepředpovídatelnému chování. V našem abstraktním interpretu necháme vyhodnocení takových výrazů skončit chybou „CHYBA: Nekorektní argument speciální formy `quasiquote`“.

Další symbol se speciálním významem pro speciální formu `quasiquote` je symbol `unquote-splicing`. Pokud argument speciální formy `quasiquote` obsahuje seznam ve tvaru `(unquote-splicing` ⟨*arg*⟩`)`, je vyhodnocen výraz ⟨*arg*⟩. Předpokládá se, že se tento argument vyhodnotí na seznam. Celý výraz

`(unquote-splicing` ⟨*arg*⟩`)`

je pak nahrazen tímto výsledným seznamem bez otevírající a uzavírající závorky. Viz příklad:

```
(quasiquote (1 (unquote (build-list 3 +)) #t))                ⟹  (1 (0 1 2) #t)
(quasiquote (1 (unquote-splicing (build-list 3 +)) #t))       ⟹  (1 0 1 2 #t)
(quasiquote (1 (unquote (map - (list 1 2 3))) #t))            ⟹  (1 (-1 -2 -3) #t)
(quasiquote (1 (unquote-splicing (map - (list 1 2 3))) #t))   ⟹  (1 -1 -2 -3 #t)
```

Odstraněny jsou přitom jen vnější závorky. Pokud seznam vzniklý vyhodnocením výrazu ⟨*arg*⟩ má jako prvky další seznamy, nebudou k nim náležející výrazy odstraněny. Tedy například:

```
(quasiquote (1 2 (unquote-splicing (map list (list 1 2 3))))) ⟹ (1 2 (1) (2) (3))
```

V sekci 4.5 jsme zavedli použití uvozovky ' jako syntaktický cukr pro speciální formu quote. Také pro speciální formu quasiquote je k dispozici zkrácený zápis. Namísto výrazu (quasiquote ⟨*arg*⟩) můžeme ekvivalentně psát jen `⟨*arg*⟩. Dále namísto výrazů (unquote ⟨*arg*⟩) a (unquote-splicing ⟨*arg*⟩) je možno psát pouze ,⟨*arg*⟩ a ,@⟨*arg*⟩ (v tomto pořadí). Například:

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

Podvýrazy argumentu speciální formy quasiquote mohou přirozeně opětně obsahovat použití této speciální formy:

```
`(1 2 ,(list `(3)))                       ⟹  (1 2 ((3)))
`(1 2 ,(map (lambda (x) `(,x)) `(1 2 3))) ⟹  (1 2 ((1) (2) (3)))
```

Nyní ukážeme několik příkladů s použitím kvotování a kvazikvotování. Všechny jsou uvedeny s použitím syntaktického cukru, i po jeho odstranění. Speciální forma quote svůj argument vrací bez jakékoli změny, bez ohledu na to, jestli se uvnitř něho vyskytují symboly unquote a unquote.

```
''(1 2 3) = (quote (quasiquote (1 2 3)))
  ⟹  (quasiquote (1 2 3))
`'(1 2 3) = (quasiquote (quote (1 2 3)))
  ⟹  (quote (1 2 3))
``(1 2 3) = (quasiquote (quasiquote (1 2 3)))
  ⟹  (quasiquote (1 2 3))
''(1 2 ,(+ 1 2)) = (quote (quasiquote (1 2 (unquote (+ 1 2)))))
  ⟹  (quasiquote (1 2 (unquote (+ 1 2))))
`'(1 2 ,(+ 1 2)) = (quasiquote (quote (1 2 (unquote (+ 1 2)))))
  ⟹  (quote (1 2 3))
```

Upozorněme, že na symboly unquote a unquote-splicing nejsou navázány ani procedury, ani speciální formy. Tím, že na tyto symboly navážeme nějakou hodnotu, nezměníme vyhodnocování speciální formy quasiquote. Viz následující příklady:

```
unquote                                          ⟹  „CHYBA: Symbol unquote nemá vazbu.“
(define unquote (lambda (x) (+ 1 x)))
`(,10) = (quasiquote (unquote 10))               ⟹  (10)
`(,,10) = (quasiquote (unquote (unquote 10)))    ⟹  (11)
```

Standard jazyka Scheme R^{5}RS neříká jakým způsobem má interpret zacházet se symboly unquote a unquote-splicing. Některé interpretry, jako je například DrScheme, se odlišují od našeho abstraktního interpretru v aplikaci speciální formy quasiquote. Výše popsaným způsobem se chovají například interpretry Bigloo nebo Guile. Například v DrScheme po navázání hodnoty na unquote přestane správně fungovat vyhodnocování kvazikvotovaných výrazů.

11.2 Zjednodušování aritmetických výrazů

V této a následujících sekcích uvedeme procedury pro zpracování symbolických výrazů. V této sekci půjde o vytvoření procedury `simplify`, která zjednodušuje aritmetické výrazy.

Aritmetické výrazy budeme v tomto případě reprezentovat S-výrazy které jsou čísla, symboly nebo dvouprvkové nebo tříprvkové seznamy, jejichž první prvek je jeden ze symbolů `+`, `-`, `*` a `/` označujících operaci a zbylé prvky jsou aritmetické výrazy. Čísla přitom reprezentují číselné hodnoty, které označují, symboly reprezentují „proměnné“, a výrazy ve tvaru seznamu reprezentují provedení operace sčítání, odčítání, násobení a dělení nad danými operandy. Například seznam `(+ (* 3 x) 2)` reprezentuje aritmetický výraz ve tvaru $3x + 2$.

Kód procedury `simplify` uvedený v programu 11.1 nejdříve popíšeme a poté se budeme věnovat úpravám tohoto řešení.

V těle procedury `simplify` definujeme několik pomocných procedur. První z nich je predikát `==`, který zjistí, zdali jsou oba jeho argumenty čísla a zdali jsou si rovna (při porovnání predikátem `=`). Dále definujeme proceduru `div-by-zero`, pomocí niž bude vrácen příznak dělení nulou a to v případě, že by měl být zjednodušován výraz, ve kterém se dělí nulou. V těle jsou dále uvedeny pomocné procedury zjednodušující určitý typ výrazu. V programu 11.1 máme uvedenu jen jednu z nich – proceduru `simplify-add` zajišťující zjednodušování výrazů, které jsou ve tvaru součtu. Další procedury, které by se v kódu nacházely na místě výpustky, jsou uvedeny v programu 11.2. Jedná se o procedury `simplify-min`, `simplify-mul` a konečně `simplify-div` zajišťující zjednodušování výrazů ve tvaru rozdílu, součinu a podílu.

Procedura `simplify` zjednodušuje výraz podle jeho charakteru. Jedná-li se o číslo nebo o symbol, jedná se o atomický výraz a zjednodušit už nejde. Pokud se jedná o seznam začínající symbolem operace, zjednoduší se jeho operandy rekurzivním voláním procedury `simplify`. V případě, že se po zjednodušení jedná o dvouprvkový seznam obsahující (mimo symbolu operace) číslo, je zjednodušením přímo výsledek vyhodnocení tohoto výrazu. Totéž platí pro tříprvkový seznam obsahující symbol operace a dvě čísla. V ostatních případech aplikujeme jednu ze zjednodušujících pomocných procedur (to jest procedur `simplify-add`, ..., `simplify-div`) v závislosti na symbolu operace, kterým zjednodušovaný výraz začíná. Viz příklady aplikace:

```
(simplify '(+ 10))                    ⟹  10
(simplify '(+ 1 2))                   ⟹  3
(simplify '(+ (* 2 3) (+ y (+ 2 x)))) ⟹  (+ 6 (+ y (+ 2 x)))
(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)
```

Elegantnějšího řešení můžeme dosáhnout tak, že vytvoříme tabulku, ve které budeme mít uložené pomocné procedury na zjednodušování a k nim příslušné symboly operací. Implementovat bychom ji mohli například tak, jak je uvedeno v programu 11.3.

V proceduře `simplify` pak můžeme větvení `cond` zkrátit třeba takto:

```
(if (and (number? x) (number? y)) (eval expr)
    (apply (cdr (assoc op simplification-table)) expr))
```

Zde využíváme procedury `assoc`, kterou jsme doposud neuvedli. Tato procedura přijímá dva argumenty $\langle e \rangle$ a $\langle l \rangle$. První argument $\langle e \rangle$ je libovolný element a druhý argument $\langle l \rangle$ je seznam tečkových párů. Procedura vrací ten nejlevější tečkový pár ze seznamu $\langle l \rangle$, jehož první prvek je shodný s elementem $\langle e \rangle$. Viz ilustrativní příklady použití `assoc`:

```
(define s '((a . 10) (b . (a b c)) (20 30) (b . 666)))
(assoc 'a s)  ⟹  (a . 10)
```

Program 11.1. Procedura pro zjednodušování aritmetických výrazů.

```
(define simplify
  (lambda (expr)
    ;; zjisti, zda-li jsou oba argumenty stejna cisla
    (define ==
      (lambda (x y)
        (and (number? x) (number? y) (= x y))))

    ;; osetreni deleni nulou
    (define div-by-zero
      (lambda () 'division-by-zero))

    ;; zjednoduseni pro pricitani
    (define simplify-add
      (lambda (op x y)
        (cond ((== x 0) y)                  zjednodušení využívající 0 + y = y
              ((== y 0) x)                  zjednodušení využívající x + 0 = 0
              ((equal? x y) `(* 2 ,x))      zjednodušení využívající x + x = 2 · x
              (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))
             (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)))))))))
```

Program 11.2. Interní definice v proceduře pro zjednodušování výrazů.

```
;; zjednoduseni pro nasobeni
(define simplify-mul
  (lambda (op x y)
    (cond ((== x 0) 0)                 zjednodušení využívající 0 · y = 0
          ((== y 0) 0)                 zjednodušení využívající x · 0 = 0
          ((== x 1) y)                 zjednodušení využívající 1 · y = y
          ((== y 1) x)                 zjednodušení využívající x · 1 = y
          (else (list op x y)))))

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

;; zjednoduseni pro deleni
(define simplify-div
  (lambda (op x y)
    (cond ((== x 0) 0)                 zjednodušení využívající 0/y = 0
          ((== x 1) (list op y))       zjednodušení využívající 1/y = y^-1
          ((== y 0) (div-by-zero))     dělení nulou je nedefinované
          ((== y 1) x)                 zjednodušení využívající x/1 = x
          ((equal? x y) 1)             zjednodušení využívající x/x = 1
          (else (list op x y)))))
```

Program 11.3. Tabulka procedur pro zjednodušování výrazů.

```
(define simplification-table
  ;; pomocne procedury == a div-by-zero
  (let ((== (lambda (x y)
              (and (number? x) (number? y) (= x y))))
        (div-by-zero (lambda () 'division-by-zero)))

   ;; vlastni tabulka
   `((+ . ,(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)
                   ((== y 0) 0)
                    ⋮
```

Program 11.4. Procedura pro vyhledávání v asociačním seznamu.

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

```
(assoc 'b s)  ⟼  (b a b c)
(assoc 20 s)  ⟼  (20 30)
(assoc 'c s)  ⟼  #f
```

Proceduru `assoc` by bylo jednoduché implementovat, jak ukazuje program 11.4.

Nyní se zaměříme na zobecnění procedury `simplify` tak, aby bylo možné zjednodušovat i aritmetické výrazy obsahující použití `+` a `*` pro n operandů. Při sčítání (násobení) vyfiltrujeme všechna čísla a sečteme (vynásobíme) je. Zbylých, to jest nečíselných prvků, vytvoříme seznam. To provedeme například následovně s využitím procedur `filter` a `remove` (viz například programy 6.2 a 6.3 na straně 146.)

```
(let ((value (apply (eval op) (filter number? (cdr expr))))
      (compound (remove number? (cdr expr))))
      ...)
```

Tyto hodnoty pak zpracujeme následující procedurou `simplify-addmul`:

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

Procedura `simplify-addmul` vrací nulu, pokud se jedná o operaci `*` a číselná hodnota navázaná na formální argument `value` je nulová. A vrací přímo číselnou hodnotu, pokud je seznam složených výrazů (navázaný na formální argument `compound`), prázdný. Pokud je číselná hodnota rovna neutrálnímu prvku příslušné operace, což zjistíme vyhodnocením výrazu `(= value (eval `(,op))`, je vrácen původní výraz bez číselné hodnoty, nebo složený výraz ze seznamu `compound`, pokud je tento seznam jednoprvkový. V ostatních případech už nelze výraz zjednodušit.

Výsledný kód s uvedenými dvěma změnami by pak vypadal tak jak je to uvedeno v programu 11.5.

Následují příklady aplikace:

```
(simplify '(+))                    ⟼  0
(simplify '(+ 10))                 ⟼  10
(simplify '(+ 1 2))                ⟼  3
(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 '(- 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)
```

Program 11.5. Vylepšená procedura pro zjednodušování aritmetických výrazů.

```
;; tabulka pro zjednodusovani
(define simplification-table
  (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)
                    ((equal? x y) 0)
                    (else (list op x y)))))

      (/ . ,(lambda (op x y)
              (cond ((== x 0) 0)
                    ((== x 1) (list op y))
                    ((== y 0) (div-by-zero))
                    ((== y 1) x)
                    ((equal? x y) 1)
                    (else (list op x y)))))))

;; vlastni procedura provadejici zjednodusovani
(define simplify
  (lambda (expr)
    (cond
      ((number? expr) expr)
      ((symbol? expr) expr)
      ((list? expr)
       (let ((op (car expr))
             (expr (map simplify expr)))
         (if (forall number? (cdr expr))
             (eval expr)
             (let ((simplifier (assoc op simplification-table)))
               (if simplifier
                   (apply (cdr simplifier) expr)
                   (error "No record for such operation"))))))
      (else (error "Incorrect input expression")))))
```

```
(simplify '(/ (+ 2 x) (+ 1 x (* 2 1/2))))  ⟹ 1
```

Procedura `simplify` samozřejmě ještě není (zdaleka) dokonalá. Na závěr sekce ukážeme několik příkladů aplikace procedury `simplify`, které ukazují její nedostatky:

```
(simplify '(+ x (- x)))           ⟹ (+ x (- x))          ideální zjednodušení: 0
(simplify '(* x (/ 1 x)))         ⟹ (* x (/ x))          ideální zjednodušení: 1
(simplify '(+ 1 (+ 1 (+ 1 x))))   ⟹ (+ 1 (+ 1 (+ 1 x)) ideální zjednodušení: (+ x 3)
(simplify '(+ x x x x))           ⟹ (+ x x x x)          ideální zjednodušení: (* x 4)
```

11.3 Symbolická derivace

V této sekci se budeme zabývat realizací procedury `diff` na nalezení symbolické derivace aritmetického výrazu (podle dané proměnné). Pro jednoduchost se omezíme jen na aritmetické výrazy s binárními operacemi.

Jak na to tedy půjdeme: Proměnné budeme derivovat na číslo 1, konstanty na číslo 0, složitější výrazy pak podle jejího derivačního předpisu. Tento předpis budeme vybírat na základě operace, kterou výraz začíná. Předpis přitom budeme reprezentovat procedurou dvou argumentů, kterými budou operandy derivovaného výrazu. Procedura předpisu bude vracet výraz odpovídající derivaci výrazu. Například pro operaci $+$, to bude procedura, která vznikne vyhodnocením λ-výrazu

```
(lambda (x y) `(+ ,(derive x) ,(derive y))),
```

která formalizuje známý derivační vztah $(f + g)' = f' + g'$. Operace a k nim příslušné předpisy budeme uchovávat v tabulce obdobně jako v sekci 11.2. Tato tabulka bude opět reprezentovaná seznamem řádků. Těmito řádky budou tečkové páry, tabulka tedy bude organizována jako asociační seznam. Tento přístup umožňuje rozšiřovat proceduru pro derivování pouhým přidáváním řádků do této tabulky. Například kdybychom chtěli derivovat i výrazy obsahující podvýrazy ve tvaru `(sin x)`, stačilo by přidat do kódu řádek

```
(sin . ,(lambda (x) `(* (cos ,x) ,(derive x))),
```

což formalizuje vztah $(\sin f)' = (\cos f') \cdot f'$ (nezapomeňte, že f nemusí být pouze proměnná). Definice procedury `diff` je uvedena v programu 11.6.

Procedura `diff` tedy pro aritmetický výraz a proměnnou vrací jeho derivaci podle této proměnné. Viz příklady aplikace této procedury:

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

Výsledný výraz můžeme zjednodušit pomocí procedury `simplify`, kterou jsme naspali v předchozí sekci.

```
(simplify (diff 'x 'x))                   ⟹ 1
(simplify (diff 'x 'y))                   ⟹ 0
(simplify (diff '(+ x x) 'x))             ⟹ 2
(simplify (diff '(+ x x) 'y))             ⟹ 0
(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))
```

Poznámka 11.2. Všimněte si, že v těle procedury `diff` v programu 11.6 jsme definovali pomocnou proceduru `derive` v jejímž těle je použita tabulka pravidel. Na druhou stranu ale taky platí, že procedury, které se nacházejí v tabulce pravidel, používají pomocnou proceduru `derive`. To je ale zcela v pořádku

Program 11.6. Procedura pro symbolickou derivaci.

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

    (define variable?
      (lambda (expr)
        (equal? expr var)))

    (define constant?
      (lambda (expr)
        (or (number? expr)
            (and (symbol? expr)
                 (not (variable? expr))))))

    (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))))))

    (define derive
      (lambda (expr)
        (cond ((variable? expr) 1)
              ((constant? expr) 0)
              (else (apply (cdr (assoc (car expr) table))
                           (cdr expr))))))

    (derive expr)))
```

a kód je naprosto funkční. Někoho by možná mohlo zmást, že v těle procedury máme několik procedur, které se na sebe vzájemně odkazují. Mohla by se tedy nabídnout otázka, zdali je to vůbec přípustné a jestli k vůli tomu při aplikaci `diff` nemůže dojít k chybě. K chybě nedojde proto, že veškeré aplikace procedur `derive` a procedur v tabulce pravidel probíhají až v okamžiku, kdy jsou v lokálním prostředí procedury `diff` zavedeny všechny lokální vazby symbolů `variable?`, `constant?`, `table` a `derive`. To je důsledkem toho, že při vzniku procedury se nevyhodnocuje její tělo. Jinými slovy, během vzniku procedury vůbec nevadí, že v jejím těle je uveden výraz obsahující symbol, který je (zatím) bez vazby. Podstatné je, že vazby budou existovat v okamžiku její aplikace, což je v případě programu 11.6 splněno.

Pomocí procedury `diff` můžeme napsat proceduru vyššího řádu, která vrací proceduru jako proceduru jednoho argumentu reprezentující matematickou funkci. Provedeme to tak, že zkonstruujeme λ-výraz, jehož tělem je výsledek aplikace procedur `simplify` a `diff` na zadaný výraz. Ten vyhodnotíme procedurou `eval`.

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

Procedura `diff-procedure` na rozdíl od procedury `derivace`, kterou jsme implementovali v programu 2.7 na straně 61 vrací přesnou reprezentaci derivované funkce. Viz následující příklad použití:

```
(define f (diff-procedure '(* x x) 'x))
(f 1)   ⟹  2
(f 10)  ⟹  20
(f 15)  ⟹  30
```

Na druhou stranu procedura `derivace` pro přibližnou derivaci z programu 2.7 je univerzálnější v tom, že jí můžeme předat jako argument rovnou proceduru. Procedura `diff` pracuje nad symbolickými výrazy a pokud by byla derivované funkce zastoupena procedurou obsahující operace, které nejsou v tabulce pravidel pro derivaci, museli bychom tabulku rozšířit.

Programovací jazyky FORTRAN a LISP vznikly prakticky současně (FORTRAN je o něco málo starší). Přesto je filosofie obou jazyků naprosto odlišná a dá se říct, že oba dva jazyky předurčily vývoj mnoha dalších rodin programovacích jazyků. Zatímco FORTRAN byl programovací jazyk sloužící pro numerické výpočty, tedy jediná data, která bylo možné ve FORTRANu zpracovávat, byla čísla, LISP byl již od počátku jazykem zpracovávajícím *symbolická data*, viz [MC60]. Za symbolická data považujeme data reprezentující symbolické výrazy, tedy čísla, symboly a seznamy. To je výrazný posun proti pouhému „numerickému zpracování čísel“. Vznik LISPu byl motivován potřebou mít k dispozici programovací jazyk, ve kterém bude možné pracovat s procedurami reprezentujícími rekurzivní funkce nad symbolickými daty. Za zmínku stojí, že motivační příklad se symbolickými derivacemi byl představen již v prvním publikovaném dokumentu o jazyku LISP, kterým je článek [MC60].

11.4 Infixová, postfixová a bezzávorková notace

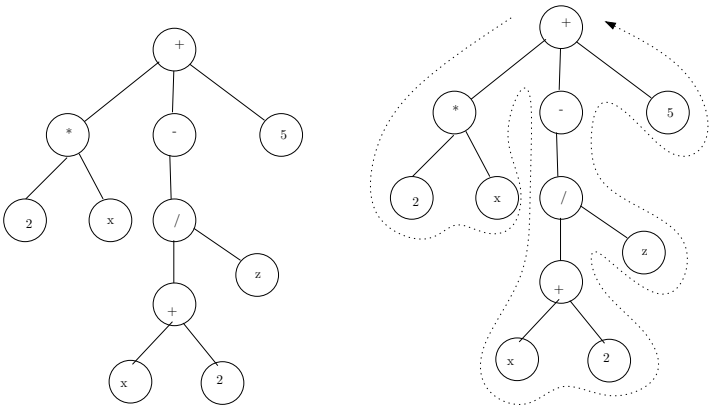
V této sekci se budeme zabývat problémem převodu výrazů zapsaných v různých notacích. Již v první lekci jsme konstatovali, že jazyk Scheme používá prefixovou notaci, ale běžně se také používají jiné notace. Nejčastější je infixová, méně časté jsou postfixová notace a postfixová bezzávorková notace (tak zvaná polská reverzní notace), viz sekci 1.2. Při psaní praktických aplikací se můžeme setkat s problémem převodu výrazů mezi těmito notacemi. Tím se tedy budeme zabývat nyní.

První a nejdůležitější je si vždy uvědomit strukturu výrazu. Máme-li například symbolický výraz

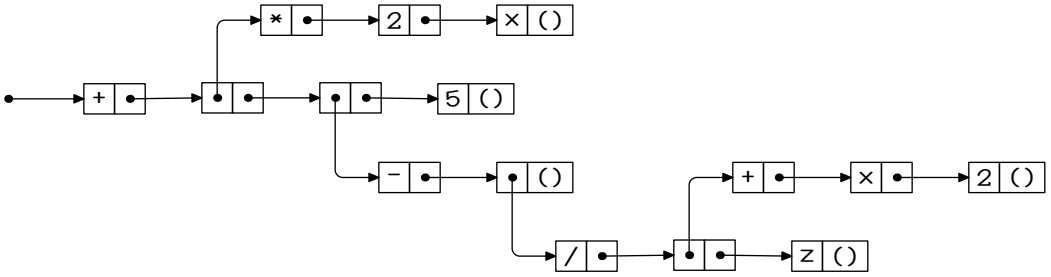
`(+ (* 2 x) (- (/ (+ x 2) z)) 5)`,

pak jej z hlediska operací a operandů můžeme zachytit uzlově ohodnoceným stromem tak, jak je tomu v obrázku 11.1 vlevo. Z hlediska datové reprezentace je seznam `(+ (* 2 x) (- (/ (+ x 2) z)) 5)`

Obrázek 11.1. Struktura výrazu `(+ (* 2 x) (- (/ (+ x 2) z)) 5)` (NACRT OBRAZKU)



Obrázek 11.2. Fyzická struktura seznamu `(+ (* 2 x) (- (/ (+ x 2) z)) 5)`



reprezentován strukturou párů, která je nakreslena v obrázku 11.2. Problém převodu prefixového výrazu do ostatních notací je vlastně problémem průchodu stromem naznačeným v obrázku 11.1, který je fyzicky reprezentovaný strukturou z obrázku 11.2. Pokud budeme tento strom procházet do hloubky, to jest ve směru tak, jak to naznačuje šipka v obrázku 11.1 vpravo, projdeme postupně všechny uzly reprezentující „podvýrazy“ přitom pro daný výraz je vždy jako první zpracován jeho nejlevější podvýraz. Jelikož se prefixová, infixová a postfixová notace liší jen pozicí (symbolu) operace, převod můžeme provést jednoduše tak, že během průchodu strukturou budeme konstruovat její „kopii“ až na to, že v každém nelistovém uzlu budeme vždy dávat symbol pro operaci na požadované místo: tedy buď před, mezi, nebo za všechny operandy. To v podstatě v grafové terminologii odpovídá *pre-order*, *in-order* a *post-order* zpracování nelistových uzlů stromu.

V programu 11.7 je uvedena procedura `prefix->postfix` pro převod výrazu v prefixové notaci do postfixové notace. Při splnění prvních tří podmínek v `cond`-výrazu je převod triviální. V případě, že vstupní výraz je seznam, je nejprve rekurzivně aplikována procedura `prefix->postfix`, což způsobí převod všech podvýrazů do postfixové notace. Z výsledku jejich převodu je vytvořen výsledný výraz připojením symbolu pro operaci za poslední z něj. Viz příklady použití procedury:

```
(prefix->postfix '(* 2 x))            ⟹  (2 x *)
(prefix->postfix '(- (/ (+ x 2) z)))  ⟹  (((x 2 +) z /) -)
```

Upravením procedury pro převod do postfixové notace můžeme vytvřit proceduru `prefix->polish` pro převod prefixových výrazů do polské reverzní bezzávorkové notace tak, jak je to ukázáno v programu 11.8. Jedinou změnou oproti myšlence použité v programu `prefix->postfix` je to, že nyní kromě připojení symbolu operace až za poselní operand navíc odstraňujeme z výrazu veškeré závorky (kromě vnějších). To v podstatě odpovídá operaci linearlizace, kterou jsme probrali v lekci 9, viz sekci 9.5. Vskutku, bezzávorkovou notaci bychom z prefixové mohli získat pouhou linearizací příslušného seznamu. Na druhou stranu, procedura z programu 11.8 pracuje efektivněji (jednoprůchodově). Viz příklady použití procedury:

```
(prefix->polish 'x)                   ⟹  (x)
```

Program 11.7. Procedura pro převod výrazů do postfixové notace.

```
(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)
           (append (map prefix->postfix (cdr S-expr))
                   (list (car S-expr))))
          (else "incorrect expression"))))
```

Program 11.8. Procedura pro převod do postfixové (polské) bezzávorkové notace.

```
(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)
           (append (apply append (map prefix->polish (cdr S-expr)))
                   (list (car S-expr))))
          (else "incorrect expression"))))
```

```
(prefix->polish 20)                  ⟼  (20)
(prefix->polish '(* 2 x))            ⟼  (2 x *)
(prefix->polish '(- (/ (+ x 2) z)))  ⟼  (x 2 + z / -)
(prefix->polish '(* 2 (+ 3 5)))      ⟼  (2 3 5 + *)
(prefix->polish '(* (+ 2 3) 5))      ⟼  (2 3 + 5 *)
```

Na předchozím příkladu si povšimněte dvou věcí. Výsledný výraz je vždy ve tvaru jediného lineárního seznamu a to i v případě, že vstupní výraz byl atom, to je rozdíl oproti závorkované postfixové notaci. Na posledních dvou řádcích předchozího příkladu je vidět, jak se do bezzávorkové notace promítá jiné uzávorkování dvou aritmetických výrazů.

Analogicky, jako jsme naprogramovali procedury pro převod z prefixové do postfixových notací, bychom mohli naprogramovat i procedury pro opačné převody. Jejich naprogramování je více méně rutinní a necháme jej na laskavém čtenáři. Mnohem obtížnější je však manipulace s infixovými výrazy. Nejprve uveďme proceduru pro převod prefixových výrazů do infixu. Procedura `prefix->infix` vykonávající tento převod je uvedena v programu 11.9. Převod čísel a symbolů do infixu je opět triviální. V případě seznamů musíme ošetřit několik situací. Ve vnitřním `cond`-výrazu nejprve ošetřujeme situaci, kdy je převáděný seznam jednoprvkový, to jest obsahuje pouze symbol pro operaci a žádné operandy. V tomto případě je převod opět triviální. Dále ošetřujeme situaci, kdy máme vstupní operaci s jedním operandem. Ten převedeme do infixu, ale symbol pro operaci píšeme pořád před něj, to odpovídá například hodnotám -2, $\frac{1}{2}$ reprezentovaným seznamy `(- 2)` a `(/ 2)`, a podobně. V ostatních případech musíme za každý operand vložit symbol pro operaci. Navíc do sebe vnoříme závorky tak, aby každá operace měla pouze dva operandy. K tomuto účelu můžeme s výhodnou použít proceduru `foldl`, viz lekci 7 věnující se akumulaci. Následující příklady ukazují použití procedury.

```
(prefix->infix '(-))          ⟼  (-)
(prefix->infix '(- 2))        ⟼  (- 2)
(prefix->infix '(- 2 3))      ⟼  (2 - 3)
```

Program 11.9. Procedura pro převod výrazů do infixové notace.

```
(define prefix->infix
  (lambda (S-expr)
    (cond ((null? S-expr) S-expr)
          ((number? S-expr) S-expr)
          ((symbol? S-expr) S-expr)
          ((pair? S-expr)
           (let* ((op (car S-expr))
                  (tail (cdr S-expr))
                  (len (length tail)))
             (cond ((= len 0) S-expr)
                   ((= len 1) (list op (prefix->infix (car tail))))
                   (else (foldl (lambda (expr collected)
                                  (list collected op expr))
                                (prefix->infix (car tail))
                                (map prefix->infix (cdr tail))))))))
          (else "incorrect expression"))))
```

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

Převod výrazu z obrázku 11.1 by dopadl takto:

```
(prefix->infix '(+ (* 2 x) (- (/ (+ x 2) z)) 5))
  ⟼  (((2 * x) + (- ((x + 2) / z))) + 5)
```

Poznámka 11.3. Podotkněme, že procedura `prefix->infix` není zdaleka dokonalá, nerespektuje například přirozenou asociativitu operací. Všechny výrazy ve tvaru $x_1 \circledcirc \cdots \circledcirc x_n$ chápe jako výrazy tvaru

$$(\cdots((x_1 \circledcirc x_2) \circledcirc x_3) \cdots \circledcirc x_n),$$

což nemusí být vždy žádoucí. V některých případech vyžadujeme závorkování opačně, někdy jej lze úplně vynechat. Úpravy procedury ponecháváme na čtenáři.

Na závěr sekce poznamenejme, že pokud jsme řekli, že převod do infixové notace je obtížnější než převod na postfixovou notaci, který je více méně rutinní, pak musíme říct, že *převod z infixové notace* (třeba do prefixové) je ještě výrazně složitější. Zvlášť je tomu tak právě v případě, kdy se vynechávají závorky z důvodu *asociativity operací*, nebo když chceme do našich úvah započíst pravidla pro *priority operací* (pravidla typu „násobení“ má přednost před „sčítáním“, tedy infixový výraz `(2 * 3 + 5)` znamená v prefixové notaci `(+ (* 2 3) 5)` a nikoliv `(* 2 (+ 3 5))`). Problém rozpoznávání struktury výrazů je obecně obtížný. V informatice se tímto obecným problémem zabývá samotná disciplína – teorie *formálních jazyků a automatů*, která je přednášena jako jeden ze základních kursů na všech informatických oborech vysokých škol. Proto ponecháme převody z infixové notace zatím stranou. Co bychom si ale z této sekce měli odnést je poznatek, že jednou z hlavních výhod prefixové notace je její jednoduchost a tím pádem snadná strojová zpracovatelnost. Zvolení prefixové notace symbolických výrazů jako základní notace pro dialekty LISPu lze tedy bez nadsázky označit jako geniální tah.

11.5 Vyhodnocování výrazů v postfixové a v bezzávorkové notaci

V této sekci se budeme snažit pro výrazy používané v předchozí sekci sestavit vhodné evaluátory. Pro výrazy v prefixovém tvaru to činit nemusíme, protože evaluátor již je nám k dispozici v podobě procedury `eval` a jedná se samotný evaluátor jazyka Scheme. Zaměříme se tedy na konstrukci procedur provádějící

Program 11.10. Procedura vyhodnocující postfixové výrazy.

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

vyhodnocování výrazů v postfixové a bezzávorkové reverzní notaci. Infixovou notací se opět kvůli složitosti nebudeme zabývat (studenti se s problematikou bíže setkají také v kursu *překladačů*).

Vyhodnocování závorkovaných postfixových výrazů bychom mohli provést analogicky jako vyhodnocování výrazů v prefixové notaci, pouze musíme počítat s tím, že výraz, jenž se má vyhodnotit na proceduru (nebo na speciální formu), stojí v seznamu jako poslední. Neformálně můžeme popsat vyhodnocování výrazů v postfixové notaci následovně:

- číslo se vyhodnotí na svou hodnotu,
- symbol se vyhodnotí na svou vazbu,
- je-li daný výraz seznam, tak se začnou vyhodnocovat jeho prvky jeden po druhém, až se vyhodnotí poslední z nich, pak se ověří, jestli se (poslední) vyhodnotil na proceduru. Pokud ano, je procedura aplikována s argumenty jimiž jsou elementy vzniklé vyhodnocením předchozích prvků seznamu.

Postup můžeme formalizovat procedurou `postfix-eval` uvedenou v programu 11.10. Tato procedura obsahuje pomocnou koncově rekurzivní proceduru `iter`, která je v jejím těle jednorázově aplikována (pomocí pojmenovaného `let`). Tato pomocná procedura se stará o postupné procházení prvků v seznamu, které jsou během průchodu postupně vyhodnocovány. Výsledky jejich vyhodnocení se postupně akumulují v seznamu navázaném na `args`. Při dosažení posledního prvku je aplikována procedura získaná vyhodnocením posledního elementu. Argumenty předané při aplikaci procedury jsou právě elementy, které byly akumulovány v seznamu navázaném na `args`.

```
(postfix-eval '(60 (10 20 +) /))  ⟹  2
(postfix-eval '((10 20 +) 60 /))  ⟹  1/2
(postfix-eval '(10 20 +))         ⟹  30
```

Je samozřejmě možné vyhodnocovat i výrazy obsahující procedury pro manipulaci s páry:

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

Upozorněme na fakt, že vyhodnocovací proces implementovaný v programu 11.10 je oproti vyhodnocovacímu procesu jazyka Scheme silně zjednodušený. Vůbec se například nepočítá s použitím speciálních forem. Viz příklad:

```
(postfix-eval '(20 ((x) x lambda)))  ⟹  „CHYBA: Symbol x nemá vazbu“
```

Důvodem chyby je právě pořadí, v jakém vyhodnocujeme argumenty. Pokud postupujeme v seznamu od jeho počátku až ke konci, právě až na konci zjistíme, zda-li se daný výraz vyhodnotil na proceduru nebo na speciální formu. To jest zavedení speciálních forem v našem modelu vyhodnocování by nemělo smysl, protože ještě před tím, než zjistíme typ aplikovaného elementu, jsou již vyhodnoceny všechny argumenty.

Nyní obrátíme naší pozornost na vyhodnocování bezzávorkových výrazů. To by mohlo být na první pohled složitější, protože ve výrazech chybějí závorky explicitně určující „strukturu výrazů“. I v tomto případě lze ale navrhnout jednoznačný vyhodnocovací proces a implementovat jej.

Před tím, než začneme konstruovat evaluátor výrazů, si musíme objasnit jeden nepříjemný rys, který souvisí s odstraněním závorek z postfixových výrazů. V bezzávorkové notaci totiž již obecně neexistuje jednoznačný přepis prefixových výrazů, pokud bereme v potaz symboly pro operace s libovolným počtem operandů, viz příklad:

```
(+ 2 3 (* 4 5))                    ⟹  25
(prefix->polish '(+ 2 3 (* 4 5)))  ⟹  (2 3 4 5 * +)
(+ 2 (* 3 4 5))                    ⟹  62
(prefix->polish '(+ 2 (* 3 4 5)))  ⟹  (2 3 4 5 * +)
```

Zde vidíme, že dva různé výrazy mají stejnou reverzní bezzávorkovou reprezentaci. Než přistoupíme ke konstrukci vyhodnocovacího procesu, musíme tuto situaci vyřešit. Jinak bychom nevěděli, zda-li vyhodnotit výraz `(2 3 4 5 * +)` na hodnotu 25 nebo na hodnotu 62 (nebo ještě nějak úplně jinak). Jelikož problém nejednoznačnosti spočívá v tom, že neznáme počet operandů pro operace, nabízí se dvě metody řešení:

(i) Každou operaci uvažovat vždy pouze *s pevnou aritou*, to jest s pevným počtem operandů. Toto řešení je jednoduché a my se jej přidržíme. Zavedeme tabulku symbolů, ve které budeme mít pro každou operaci zaznamenán počet jejích operandů.

(ii) Do výrazů budeme před každý symbol operace vždy zapisovat počet operandů, na které se vztahuje. Například ve výrazu `(+ 2 3 (* 4 5))` má „`+`“ tři operandy a „`*`“ pouze dva. V reverzní bezzávorkové notaci bychom tedy výraz zapsali ve tvaru `(2 3 4 5 2 * 3 +)`, přitom červeně jsou zdůrazněny hodnoty reprezentující počty operandů. Na druhou stranu výraz `(+ 2 (* 3 4 5))` by byl reprezentován seznamem `(2 3 4 5 3 * 2 +)`. Oba výrazy jsou tedy různé a k nejednoznačnostem nedochází. Výrazy kódované tímto způsobem jsou pochopitelně delší, ale zase v nich můžeme používat operace s libovolnými operandy.

Nyní vezmeme v úvahu úmluvu (i) a budeme se soustředit na implementaci. Jako první budeme definovat tabulku symbolů, viz definici v programu 11.11. V této tabulce je pro každý symbol uvažované operace

Program 11.11. Jednoduchá tabulka s prostředím vazeb pro postfixový evaluátor.

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

tříprvkový záznam ve formě seznamu `(`⟨*symbol*⟩ ⟨*arita*⟩ ⟨*procedura*⟩`)`, kde ⟨*symbol*⟩ je jméno operace, ⟨*arita*⟩ je číslo určující počet operandů dané operace a ⟨*procedura*⟩ je procedura používající se při „aplikaci operace“. Všimněte si, že v tabulce 11.11 jsme definovali unární operaci označenou `n`, což je unární minus. Pro unární minus již nemůžeme zvolit symbol „`-`“, který je vyhrazen pro odčítání dvou čísel (uvědomte si, že arita „`-`“ je dána pevně na jedinou hodnotu).

Vyhodnocování postfixových výrazů budeme provádět pomocí dodatečné datové struktury – zásobníku. Samotná reprezentace zásobníku je z našeho pohledu primitivní, protože jej můžeme reprezentovat přímo seznamem (`cons` „přidává“ prvek na vrchol zásobníku, `car` vrací prvek na vrcholu, a `cdr` „odebírá“ prvek z vrcholu zásobníku). Během vyhodnocování tedy budeme mít k dispozici *vstupní výraz* reprezentovaný lineárním seznamem čísel a symbolů a *zásobník*. V každém kroku se ze vstupního výrazu odebere nejlevější „slovo“ (číslo nebo symbol) a zpracuje se. Zásobník je na počátku prázdný. Podle typu slova na vstupu budeme rozlišovat dvě situace:

(i) Na vstupu je číslo. V tomto případě přesuneme číslo na vrchol zásobníku a zpracujeme následující vstupní slovo.

(ii) Je-li na vstupu symbol f označující operaci, odstraníme jej ze vstupu a vyzvedneme ze zásobníku tolik prvků, jaká je arita symbolu f (aritu nejdeme v tabulce symbolů, kterou jsme již zavedli) a aplikujeme příslušnou proceduru s těmito argumenty (procedura je opět k nalezení v tabulce). Výsledek vyhodnocení dáme na vrchol zásobníku.

Vyhodnocování končí vypotřebováním vstupu. V tom případě je výsledek uložen na vrcholu zásobníku (nebo za výsledek vyhodnocování můžeme považovat celý zásobník).

Příklad 11.4. (a) Uvažujme výraz `(10 20 +)`. Průběh jeho vyhodnocování daný předchozím postupem, můžeme zaznamenat v tabulce se dvěma sloupci. První sloupec ukazuje stav vstupního výrazu, ze kterého jsou postupně zepředu odebírány prvky, a druhý sloupec představuje aktuální stav zásobníku. Každý řádek tabulky pak koresponduje s jedním elementárním krokem výpočetu. V případě našeho výrazu bude výpočet vypadat takto:

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

Výsledek vyhodnocení je tedy `30`.

(b) V případě výrazu `'(10 20 30 + *)` probíhá vyhodnocování takto:

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

Výsledek vyhodnocení je `500`.

(c) Konečně, v případě `'(10 20 + 30 *)` probíhá vyhodnocování takto:

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

Výsledek vyhodnocení je `900`.

Proceduru provádějící vyhodnocování pomocí zásobníku můžeme naprogramovat tak, jak je to uvedeno v programu 11.12. Zde je uvedena procedura `polish-eval`, která jako argumenty akceptuje daný výraz, který bude vyhodnocen, a tabulku symbolů. V našem případě budeme vždy používat tabulku z programu 11.11. Nic však nebrání tomu, abychom zavedli jinou tabulku. Procedura ve svém těle používá pomocnou interní iterativní proceduru `iter`, která má dva argumenty: seznam reprezentující vstupní výraz a seznam reprezentující zásobník (na počátku prázdný). Ve svém těle procedura dělá přesně to, co jsme řekli v předchozích paragrafech. Pomocná procedura `list-pref` uvedená v programu 11.11 slouží k získání prvních n prvků ze seznamu: procedura slouží k vyzvednutí více hodnot ze zásobníku před aplikací procedury odpovídající symbolu operace.

```
(polish-eval '(10 20 +) env)        ⟹  (30)
(polish-eval '(10 20 30 +) env)     ⟹  (50 10)
```

Program 11.12. Procedura vyhodnocující výrazy v reverzní bezzávorkové notaci.

```
(define list-pref
  (lambda (n l)
    (if (<= n 0)
        '()
        (cons (car l)
              (list-pref (- n 1) (cdr l))))))

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

```
(polish-eval '(10 20 30 + *) env)    ⟹  (500)
(polish-eval '(10 20 30 * +) env)    ⟹  (610)
(polish-eval '(10 20 + 30 *) env)    ⟹  (900)
(polish-eval '(10 20 * 30 +) env)    ⟹  (230)
(polish-eval '(10 30 n +) env)       ⟹  (-20)
(polish-eval '(10 n 30 +) env)       ⟹  (20)
(polish-eval '(10 n 30 n +) env)     ⟹  (-40)
(polish-eval '(10 n 30 n + n) env)   ⟹  (40)
(polish-eval '(10 8 2 / ^) env)      ⟹  (10000)
```

Zásobníkové vyhodnocování postfixových bezzávorkových výrazů se používá daleko častěji, než jak bychom možná intuitivně čekali. Na tomto stylu vyhodnocování výrazů je založeno celé jedno minoritní paradigma – *zásobníkové paradigma* (dost často se však za samostatné paradigma nepovažuje). Typickým zástupcem zásobníkového jazyka je FORTH. Jazyk PostScript, který umí interpretovat každá trochu lepší tiskárna, a který je v současnosti de facto standardem, pokud jde přes přenositelné formáty popisu tiskové strany, je rovněž dialektem jazyka FORTH upraveným právě pro použití popisu obsahu tiskové strany. Virtuální stroje zpracovávající programy v bajtkódu (viz první lekci) jsou vesměs naprogramovány jako zásobníkové vyhodnocovací programy zpracovávající programy v zásobníkových jazycích. Existují samozřejmě i hardwarové zásobníkové procesory, které jsou součástí malých počítačů a tiskáren.

Shrnutí

V této lekci jsme se zabývali zpracováním symbolických výrazů reprezentovaných seznamy skládajícími se z čísel, symbolů a dalších seznamů reprezentující symbolické výrazy. Nejprve jsme pro zjednodušení práce zavedli rozšíření kvotování – tak zvané kvazikvotování, což je obecnější metoda kvotování umožňující programátorům snadněji definovat některé seznamy. V lekci jsme se potom věnovali zjednodušování aritmetických výrazů, tyto aritmetické výrazy byly reprezentovány seznamy. Ukázali jsme několik procedur pro zjednodušování s různou vnitřní strukturou a s různými schopnostmi. Dalším příkladem bylo počítání symbolických derivací. Potom jsme naši pozornost přesunuli na reprezentaci výrazů v infixové, postfixové, a reverzní bezzávorkové notaci. Naprogramovali jsme řadu procedur pro konverzi výrazů v prefixové notaci na ostatní notace a zpět. Poukázali jsme na fakt, že infixová notace je z hlediska strojového zpracování dost komplikovaná. V poslední sekci jsme se věnovali konstrukcí procedur vyhodnocující výrazy v postfixové a reverzní bezzávorkové notaci. Navrhli jsme několik modelů jejich vyhodnocování a ukázali jejích silné a slabé stránky. Vyhodnocování reverzních bezzávorkových výrazů jsme naprogramovali pomocí manipulace s dodatečným zásobníkem, který byl fyzicky reprezentován seznamem.

Pojmy k zapamatování

- kvazikvotování
-
-
-

Nově představené prvky jazyka Scheme

- speciální forma `quasiquote`
- procedura `assoc`

Kontrolní otázky

1. *Co je kvazikvotování? jak se liší od kvotování?*
2.

3.

Cvičení

1. Bez použití interpretru určete výsledky vyhodnocení následujících výrazů:

```
(quasiquote symbol)
`(symbol)
(car ``symbol)
`(+ 1)
`(,+ 1 2)

`(1 ,1)
`(1 ,,1)
`(,+ . ,-)
`(1 + 2 ,@3)
(quote (,@()))

(quasiquote quasiquote)
(quasiquote (+ 1 (unquote +)))
(quasiquote (1 2 ,(+ 1 2)))
`,(+ 1 2)
`,`,(+ 1 2)
`(1 2 ,@(build-list 5 (lambda (x) (* x x))) 3)

(quasiquote (1 2 (unquote-splicing (map list '(1 2 3))) 5))
unquote-splicing
```` ```() ````
``'+
(quote unquote)
```

2. Upravte kód procedury `diff` ze sekce **??** tak, aby bylo možné derivovat výrazy, ve kterých je součet a součin použit na libovolné množství argumentů.
3. prevod postfixu na prefix
4. prevod bezzavorkove notace s pevnou aritou na prefix
5. prevod prefixu na bezzav. notaci s aritami ve vyrazu
6. prevod bez/z. vyrazu s aritami ve vyrazu na prefix
7. zasobnikovy vyhodnocovac bez.z. pracujici s aritami ve vyrazech

**Úkoly k textu**

1. Popište, jak nahradit použití speciální formy `quasiquote`.
2. Rozšiřte proceduru `simplify` tak, aby neměla nedostatky uvedené na konci sekce 11.2.
3. Evaluator pro infixove vyrazy. Pevne dany pocet argumentu (nejvys dva).

### Řešení ke cvičením

1. symbol, (symbol), quasiquote, (+ 1), („procedura sčítání“ 1 2)
(1 1) chyba („procedura sčítání“ . „procedura odčítání“) chyba ((unquote-splicing ()))
quasiquote (+ 1 „procedura sčítání“) (1 2 3) 3 3 (1 2 0 1 4 9 16 3)
(1 2 (1) (2) (3) 5) chyba (quasiquote (quasiquote ())) '+ unquote
```

2. Oproti programu 11.6 sta zmnit lokln definici tabulky takto:

```
(define table
  (let ((2*->* (lambda (x) (if (and (pair? x) (equal? '*2 (car x)))
                               `(* ,@(cdr x))
                               x))))

    `((+ . ,(lambda l `(+ ,@(map (lambda(x) (derive x)) l))))
      (- . ,(lambda (x y) `(- ,(derive x) ,(derive y))))
      (* . ,(lambda l (derive (foldr (lambda (x a) `(*2 ,x ,a)) 1 l))))
      (*2 . ,(lambda (x y) `(+ (* ,(2*->* x) ,(derive y)) (* ,(derive x) ,(2*->* y)))))
      (/ . ,(lambda (x y) `(/ (- (* ,(derive x) ,y) (* ,x ,(derive y))) (* ,y ,y)))))))
```