KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 1 ◇ poznámky k přednášce

12. Interpret

verze z 11. prosince 2019

1 Funkce `eval` v Lispu

V prvních částech našeho kurzu jsme se seznamovali s vyhodnocovacím procesem jazyka Lisp. To je přesně popsaný proces, jehož účelem je k daným výrazům zjišťovat jejich hodnoty. Kód (program) napsaný v Lispu pracuje tak, že postupně vyhodnocuje výrazy, takže popsat vyhodnocovací proces znamená popsat, jak programy v Lispu fungují.

Vyhodnocovací proces pracuje s **výrazy**, což jsou **jednoduché výrazy** neboli **atomy** (čísla, symboly, prázdný seznam, textové řetězce) a **složené výrazy** neboli (trochu nepřesně) **seznamy**. V dalších přednáškách jsme se učili s výrazy (zejména seznamy) pracovat. Výrazy našeho programovacího jazyka můžeme tedy chápat jako **data** (jednoduchá nebo datové struktury) a jako s daty s nimi umíme pracovat.

V principu by pro nás mělo být možné napodobit programátory LispWorks a implementovat samotný vyhodnocovací proces. Je to ze dvou shora uvedených důvodů:

1. zdrojový kód máme k dispozici jako data, se kterými umíme pracovat,
2. vyhodnocovací proces je přesně popsaný.

Samotný Lisp umí dělat totéž. Je v něm totiž k dispozici funkce `eval` na vyhodnocování výrazů:

```
CL-USER 1 > (eval '(setf p 10))
10

CL-USER 2 > p
10

CL-USER 3 > (setf q '(+ p 1))
(+ P 1)

CL-USER 4 > (eval q)
11

CL-USER 5 > (eval (list 'let '((o 5)) (list '* 'o q)))
55
```

V této přednášce se pokusíme napsat vlastní funkci `eval` pro (podstatně) zjednodušenou variantu jazyka Lisp. Tím vlastně naprogramujeme jeho **interpret**, tedy program, který je schopen vykonávat program napsaný v Lispu.

Interpret bude

- umět vyhodnocovat symboly a ostatní atomy
- umět vyhodnocovat složené výrazy
- poskytovat uživateli možnost definovat nové globální proměnné a uživatelské funkce
- obsahovat některé vestavěné funkce (*primitiva*) jako aritmetické operace a funkce na práci se seznamy
- znát čtyři základní speciální operátory

2 Zjednodušený Lisp

Naše zjednodušená varianta Lispu bude vycházet z jazyka **Scheme**, který sám je jednoduchou verzí Lispu.

Jazyk Scheme používá jednodušší vyhodnocovací proces než Lisp, což nám zjednoduší implementaci interpretu. Rozdíl spočívá v tom, že ve Scheme **symboly nemají funkční vazby**. Pokud chceme symbol použít jako název funkce, uložíme do něj funkci jako hodnotu, stejně jako kdybychom ho použili jako proměnnou.

Ve Scheme se ovšem funkcím říká **procedury**, takže tento termín budeme pro funkce jazyka Scheme používat i my.

K definici nové vazby v globálním prostředí slouží ve Scheme operátor `define`. Ten se tedy používá jak k definici proměnných, tak procedur:

```
(define x 5)                                                    Scheme
(define power2 (lambda (x) (* x x)))
```

První výraz vytvoří v globálním prostředí vazbu na symbol `x` a nastaví jeho hodnotu na číslo `5` (protože hodnotou čísla je totéž číslo). Druhý výraz vytvoří v globálním prostředí vazbu na symbol `power2` a nastaví jeho hodnotu na proceduru vzniklou vyhodnocením výrazu `(lambda (x) (* x x))`.

Když dáme symbol vyhodnotit, dostaneme jako výsledek proceduru:

```
> power2                                                        Scheme
#< ... procedura ... >
```

Při vyhodnocení tohoto výrazu:

```
(power2 x)                                                    Scheme
```

dostaneme jako výsledek `25`, ale jednodušším způsobem než v Lispu: první prvek seznamu, tedy symbol `power2` se prostě **vyhodnotí** (tak to v Lispu nebylo) a pak se aplikuje procedura, která je jeho hodnotou. (Symbol `x` má stále hodnotu `5`, kterou jsme do něj před chvílí uložili.)

Takto funguje ve Scheme vyhodnocování složených výrazů: při vyhodnocování seznamu

```
(op arg1 arg2 ... argn)                                       Scheme
```

se vykonají tyto kroky:

1. Jestliže *op* je speciální operátor, vyhodnotí se výraz podle jeho pravidel.
2. Jinak se *op* **vyhodnotí**. Výsledkem **musí být procedura**.
3. Potom se vyhodnotí argumenty a procedura se na ně aplikuje jako v Lispu.

Ve Scheme tedy operátor nemusí být symbol. Může to být libovolný výraz, který se vyhodnotí na proceduru:

```
> ((if (> 1 0) + -) 1 1)                                      Scheme
2
```

Ve Scheme tedy nepotřebujeme operátory `function` a `funcall`. Z tohoto pohledu je jednodušší než Lisp.

3 Pojmy a zásady pro interpret

Jak bylo řečeno, budeme programovat **interpret** jazyka. Čtenář se asi setkal i s pojmem **kompilátor**. Proto bude dobré vyjasnit, jaký je mezi nimi rozdíl.

Interpret (*interpretr*) programovacího jazyka je program, který vykonává program zadaný zdrojovým kódem napsaným v programovacím jazyce krok po kroku (výrazu, příkazu). Vykonává přímo úlohu, která je ve zdrojovém kódu zapsána, aniž by program překládal do jiného jazyka.

Kompilátor (*překladač*) programovacího jazyka je něco zcela jiného. Je to program, který převádí (překládá) program zapsaný v jednom programovacím jazyce do jiného (obvykle jednoduššího, nižšího) programovacího jazyka. Sám přitom obvykle žádné instrukce programu nevykonává. Přeložený program slouží pak jako vstup jinému interpretu případně kompilátoru. V Lispworks je k dispozici jak interpret, tak kompilátor jazyka. Kompilátor překládá program do strojového kódu

procesoru, který je pak vykonán počítačem. Interpret je napsán v Lispu a je realizován funkcí `eval`.

Zásady pro náš interpret:

1. Při interpretaci programu budeme využívat hodnoty Lispu (páry, symboly, čísla ...).
2. Dále budeme využívat primitiva Lispu (např. funkci na sčítání čísel).
3. Program k interpretaci budeme zadávat jako datovou strukturu jazyka Lisp (seznam), nikoliv v textové podobě, jak je obvyklé. Tím odpadne potřeba analyzovat textový zápis programu.

Vstupním bodem našeho interpretu bude funkce `ss-eval` („`ss`“ jako *Simple Scheme*), jejímž úkolem bude vyhodnotit zadaný výraz a vrátit jeho hodnotu:

```
CL-USER 12 > (ss-eval '(+ 1 2))
3

CL-USER 13 > (ss-eval '((lambda (x y) x) 1 2))
1

CL-USER 14 > (ss-eval '(define proc (lambda (x y) (+ x y 1))))
PROC

CL-USER 15 > (ss-eval '(proc 1 2))
4
```

Jak vidíme, funkce bude akceptovat jako argument výraz (obvykle neprázdný seznam). Ten musí podle pravidel vyhodnocovacího procesu našeho Scheme postupně zpracovat a vrátit jako výsledek jeho hodnotu.

Takto navržená funkce `ss-eval` by ovšem nebyla schopna provést vyhodnocení výrazu podle všech pravidel, která jsme si říkali. Obecný vyhodnocovací proces totiž vyhodnocuje daný výraz v daném **prostředí**. Součástí našeho úkolu bude tedy reprezentovat prostředí vhodnou datovou strukturou a naprogramovat vyhodnocování výrazů v zadaném prostředí.

Na vyhodnocení výrazu v daném prostředí napíšeme funkci `ss-eval-env`, která bude mít druhý parametr, jímž budeme zadávat (vhodně reprezentované) prostředí. Pokud by například v prostředí `env` byla vazba symbolu `x` na hodnotu `1` a symbolu `y` na hodnotu `2`, pak by mělo jít udělat toto:

```
CL-USER 20 > (ss-eval-env '(+ x y) env)
3

CL-USER 21 > (ss-eval-env '((lambda (x) (* x y)) 5) env)
10
```

Funkce bude napsána přesně podle zásad vyhodnocovacího procesu našeho jazyka Scheme, tedy tak, jak jsme se učili během semestru se zjednodušením uvedeným na začátku této přednášky. Uvidíme tedy, že vyhodnocovací proces je možné naprogramovat.

Interpret Scheme bude zatím nejdelší program, který jsme v tomto předmětu psali. Proto se budeme přísně držet základních zásad psaní programu, abychom se v tom neztratili. (To bude poučné i pro vaši budoucí práci.)

1. Každý složitější problém, který budeme řešit, rozdělíme na jednodušší podproblémy. Ty pak opět rozdělíme a budeme v tom postupovat tak dlouho, až dostaneme triviální problémy, které půjde snadno naprogramovat pomocí krátkých funkcí. Žádná funkce, kterou budeme psát, tedy pokud možno nebude přímo řešit více než jeden problém. Pokud by měla, raději v jejím těle jen zavoláme jiné funkce.

2. Program budeme udržovat stále funkční. Každou úpravu a každé doplnění pečlivě naplánujeme a pak je uděláme tak, aby program opět (byť omezeně) fungoval. To vždy důkladně otestujeme.

V těchto poznámkách (a přidružené prezentaci) můžete číst, jak program postupně vznikal. Vzhledem k tomu, že se to dělo postupnou úpravou různých jeho částí, je pořadí definic ve výsledném zdrojovém kódu jiné, než jaké vidíte tady.

4 Interpret

4.1 Funkce `ss-eval` a `ss-eval-env`

Jak jsme řekli, funkce `ss-eval` bude vstupním bodem programu. Víme, že Scheme vyhodnocuje výrazy v daném prostředí. I když zatím nevíme, jak budeme prostředí reprezentovat, je jasné, že musíme napsat funkci na vyhodnocení výrazu v daném prostředí. Bude to funkce `ss-eval-env`. Ji bude funkce `ss-eval` volat s prostředím rovným globálnímu prostředí. Funkce se tedy budou aplikovat takto:

```
(ss-eval expr)
(ss-eval-env expr env)
```

kde

expr je libovolný výraz a

env prostředí, ve kterém se má vyhodnotit.

Ještě jednou opakuji: zatím nevíme, jak budeme prostředí reprezentovat. To nám ale zatím nevadí. Určitě to bude nějakou datovou strukturou, ve které budou uloženy potřebné informace (o tom později).

Abychom se vyhnuli technickým problémům souvisejícím s mutací datových struktur, které si podrobně vysvětlíme příští semestr, zavedeme možnost označovat globální prostředí symbolem `t`.

Funkci `ss-eval` tedy můžeme napsat takto:

```
(defun ss-eval (expr)
  (ss-eval-env expr t))
```

Teď se pustíme do funkce `ss-eval-env`.

První krok, který musí vyhodnocovací proces při vyhodnocování výrazu udělat, je zjistit, jestli je výraz atom, nebo složený výraz (tedy pár; prázdný seznam je atom). Podle toho pak proces postupuje dál.

Na zjištění, zda je výraz atomický, nebo složený si napíšeme predikát. Je jasné, že pokud je výraz napsán správně, je složený právě když je párem:

```
(defun compound-expr-p (expr)
  (consp expr))
```

Věrni své zásadě psát funkce co nejjednodušeji si na problémy vyhodnotit atom a vyhodnotit (neprázdný) seznam napíšeme později samostatné funkce (tedy rozdělíme složitý problém na dva jednodušší). Funkce `ss-eval-env` tedy bude jednoduchá:

```
(defun ss-eval-env (expr env)
  (if (compound-expr-p expr)
      (eval-compound-expr expr env)
    (eval-atom expr env)))
```

Funkce popisuje přesně to, co se děje na začátku vyhodnocování výrazu v daném prostředí: zjistí se, zda jde o složený výraz, a pokud ano, vyhodnotí se výraz postupem pro složený výraz (v jiné funkci). Pokud ne, vyhodnotí se jako atom.

4.2 Vyhodnocování neproměnných atomů

Funkci `ss-eval` zatím nemůžeme otestovat, protože žádná větev funkce `ss-eval-env` není dosud implementována. Jednodušší bude zřejmě napsat funkci `eval-atom`, tak to uděláme v tomto kroku.

Víme, že zvláštním typem atomu je symbol a vyhodnocuje se jinak než ostatní atomy, které se vyhodnocují jednoduše samy na sebe. Tuto část vyhodnocovacího procesu tedy můžeme rovnou napsat:

```
(defun eval-atom (atom env)
  (if (symbolp atom)
      (sym-value atom env)
    atom))
```

(Využili jsme predikát `symbolp` Lispu, který zjišťuje, zda zadaná hodnota je symbol.)

Nyní už náš interpret něco umí: vyhodnocovat neproměnné atomy. Můžeme to vyzkoušet:

```
CL-USER 1 > (ss-eval 1)
1

CL-USER 2 > (ss-eval "Lisp")
"Lisp"
```

(Pokus o vyhodnocení čehokoliv jiného by samozřejmě vedl k chybě.)

4.3 Prostředí a vyhodnocování symbolů

Funkce `sym-value` použitá výše má vrátit hodnotu symbolu v daném prostředí. Abychom ji mohli napsat, musíme už pořádně definovat prostředí.

Prostředí reprezentujeme datovou strukturou.

Víme, že prostředí musí obsahovat následující informace:

1. Svého předka,
2. tabulku vazeb symbolů.

Předkem prostředí je opět prostředí. Pokud prostředí předka nemá, vyjádříme to hodnotou `nil`.

Tabulku vazeb symbolů budeme reprezentovat seznamem párů. Celkově budeme prostředí reprezentovat seznamem následujícího tvaru:

```
(env bindings parent)
```

Je to tedy tříprvkový seznam. První jeho prvek je symbol `env`. Sám o sobě nemá žádný význam, jen usnadňuje kontrolu toho, že zadaný seznam reprezentuje prostředí. Druhý prvek, *`bindings`*, je seznam vazeb. Ten, jak už bylo řečeno, je seznamem párů. Například seznam

```
((a . 0) (b . 1))
```

představuje vazbu symbolu `a` na hodnotu `0` a symbolu `b` na hodnotu `1`. Se seznamy tohoto tvaru se hezky pracuje.

Třetí prvek seznamu reprezentujícího prostředí, ***parent***, je předek prostředí, což je buď jiné prostředí, nebo `nil` (v případě, že prostředí nemá předka).

Když jsme si vyjasnili tvar struktury reprezentující prostředí, napíšeme její konstruktor a selektory.

```
(defun make-env (bindings parent)
  (list 'env bindings parent))
```

Konstruktor `make-env` vytvoří nové prostředí ze zadaného seznamu vazeb a předka. Například následující vyhodnocení vytvoří prostředí s vazbou symbolu `a` na 0 a symbolu `b` na 1, které nemá předka:

```
CL-USER 3 > (make-env '((a . 0) (b . 1)) nil)
(ENV ((A . 0) (B . 1)) NIL)
```

Jak jsem už řekl, symbolem `t` označujeme globální prostředí (které ale později taky vytvoříme).

Takto by tedy mohlo být vytvořeno prostředí, jehož předkem je jiné prostředí, jehož předkem je globální prostředí (prostředí si uložíme do proměnné pro další použití):

```
CL-USER 5 > (setf env
                  (make-env '((a . 0) (b . 1))
                            (make-env '((x . u) (z . w)) t)))
(ENV ((A . 0) (B . 1)) (ENV ((X . U) (Z . W)) T))
```

Selektory datové struktury prostředí budou vracet jednak seznam vazeb a jednak předka. Jak víme, globální prostředí může být reprezentováno symbolem `t`. Pokud se tak stane, je třeba, aby selektory nejprve našly globální prostředí a pak z něj požadované informace vrátily. Už také víme, že globální prostředí bude vracet funkce `initial-env`. Tu za chvíli také napíšeme. Takže selektory prostředí:

```
(defun get-env (spec)
  (if (eql spec t) (initial-env) spec))

(defun env-bindings (env)
  (cadr (get-env env)))
```

```
(defun env-parent (env)
  (caddr (get-env env)))
```

Funkce `sym-value`, kterou se snažíme napsat, má zjistit hodnotu vazby symbolu v daném prostředí. To obnáší několik věcí:

1. Zjistit, zda má symbol vazbu v daném prostředí.
2. Pokud nemá, hledat vazbu rekurzivně v předcích prostředí.
3. Vrátit hodnotu nalezené vazby.

Když zatím odhlédneme od toho, jak se vazba symbolu v prostředí hledá, můžeme funkci `sym-value` napsat takto:

```
(defun sym-value (symbol env)
  (cdr (symbol-binding symbol env)))
```

Jak vidíme, hledání vazby symbolu jsme svěřili funkci `symbol-binding`. Podle uvedeného návodu ji nyní můžeme napsat takto:

```
(defun symbol-binding (symbol env)
  (let ((bnd (symbol-binding-1 symbol env)))
    (if (null bnd)
        (symbol-binding symbol (env-parent env))
      bnd)))
```

Použili jsme funkci `symbol-binding-1`, která hledá vazbu symbolu v daném prostředí a nedívá se rekurzivně do předka. Funkce je napsána pomocí funkce `findf`, kterou známe z předchozích přednášek.

```
(defun symbol-binding-1 (symbol env)
  (findf (lambda (cons)
           (eql (car cons) symbol))
         (env-bindings env)))
```

Nyní bychom měli pečlivě otestovat jednak funkci `sym-value` a jednak celý vylepšený interpret. Použijeme přitom prostředí, které máme z předchozího testu uloženo v proměnné `env`.

```
CL-USER 6 > (symbol-binding-1 'a env)
(A . 0)

CL-USER 7 > (symbol-binding-1 'z env)
NIL

CL-USER 8 > (symbol-binding-1 'z (env-parent env))
(Z . W)

CL-USER 9 > (symbol-binding 'z env)
(Z . W)

CL-USER 10 > (sym-value 'a env)
0
```

```
CL-USER 11 > (sym-value 'z env)
W

CL-USER 12 > (ss-eval-env 'a env)
0

CL-USER 13 > (ss-eval-env 'z env)
W
```

Nyní můžeme také definovat globální prostředí. Uložíme do něj několik primitiv, která budeme v budoucnu potřebovat a pro zajímavost také proměnnou `zero` obsahující nulu.

Než to uděláme, vyjasníme, kam globální prostředí uložíme. Globální proměnné máme zatím zakázány, tak budeme postupovat tak, že vytvoříme operátorem `let` prostředí, ve kterém bude naše schemovské globální prostředí uloženo v proměnné `env`, a dále funkci `initial-env`, která bude hodnotu proměnné vracet:

```
(let ((env (make-env (list (cons 'zero 0) (cons '+ #'+)
                           (cons '* #'*) (cons '= #'=)
                           (cons 'cons #'cons))
                     nil)))
  (defun initial-env ()
    env))
```

Funkce `initial-env` funguje díky tomu, že to je **lexikální uzávěr** ve smyslu předchozí přednášky. Pamatuje si prostředí svého vzniku, tj. prostředí s vazbou

symbolu `env`, takže při aplikaci tento symbol ve svém těle vyhodnotí na požadovanou hodnotu.

Na našem testovacím prostředí `env`, které jsme definovali před chvílí, teď můžeme vyzkoušet i hodnoty proměnných uložených v globálním prostředí:

```
CL-USER 14 > (symbol-binding 'zero env)
(ZERO . 0)

CL-USER 15 > (symbol-binding 'cons env)
(CONS . #<Function CONS 40F008970C>)
```

Také můžeme konečně použít funkci `ss-eval` a zkusit vyhodnotit výrazy v globálním prostředí:

```
CL-USER 16 > (ss-eval 'zero)
0

CL-USER 17 > (ss-eval '>)
#<Function > 40F006E014>
```

4.4 Vyhodnocování složeného výrazu a aplikace primitiv

Už jsme naučili náš interpret vyhodnocovat atomy. Nyní se pustíme do vyhodnocování neprázdných seznamů. Musíme tedy naprogramovat funkci `eval-compound-expr`.

Víme, že při vyhodnocování seznamu se má systém nejprve podívat na první prvek (tedy operátor) a pokud půjde o speciální operátor, má zvolit příslušný speciální způsob vyhodnocení. Pokud o speciální operátor nepůjde, provede se standardní vyhodnocení, které povede na aplikaci procedury.

Případ speciálního operátoru lze řešit různými chytrými způsoby. My se budeme držet při zemi a v jednom velkém větvení postupně otestujeme, zda první prvek seznamu není jedním ze stanovených speciálních operátorů. Pokud nebude, budeme vědět, že jde o aplikaci procedury.

Pro začátek napíšeme funkci `eval-compound-expr` pro základní speciální operátory: `quote`, `if`, `define` a `lambda`.

```
(defun eval-compound-expr (expr env)
  (let ((op (car expr))
        (args (cdr expr)))
    (cond ((eql op 'quote) (eval-quote args))
          ((eql op 'if) (eval-if args env))
          ((eql op 'define) (eval-define args))
```

```
          ((eql op 'lambda) (eval-lambda args env))
          (t (eval-application op args env)))))
```

Funkce `eval-quote`, `eval-if`, `eval-define` a `eval-lambda` slouží k vyhodnocení speciálních výrazů s operátory `quote`, `if`, `define` a `lambda`. Víme, že pro každý speciální operátor musí interpret znát jeho způsob vyhodnocování. Při rozšiřování našeho interpretu o další speciální operátory bychom tedy museli přidat větev do `cond`-výrazu ve funkci `eval-compound-expr` a napsat příslušnou funkci pro vyhodnocení. Jak už bylo řečeno, tento trochu těžkopádný způsob, který vyžaduje přepisování funkce `eval-compound-expr`, lze vylepšit.

My se teď budeme věnovat základnímu případu vyhodnocení seznamu, kdy operátor není speciálním operátorem. Jinými slovy, teď se pustíme do funkce `eval-application`.

Jak víme, při vyhodnocování seznamu, jehož první prvek není speciální operátor, je nejprve třeba vyhodnotit první a potom všechny ostatní prvky seznamu. Hodnota prvního prvku musí být procedura, ta se aplikuje na hodnoty zbylých prvků. Máme tedy návod, jak napsat funkci `eval-application`:

```
(defun eval-application (op args env)
  (apply-proc (ss-eval-env op env)
              (mapcar (lambda (el) (ss-eval-env el env))
                      args)))
```

S takto implementovanou funkcí bychom se ovšem neměli spokojit. Je vidět, že dělá příliš mnoho věcí. Přitom vyhodnocení všech prvků daného seznamu si zaslouží vlastní funkci:

```
(defun eval-list-elements (list env)
  (mapcar (lambda (el) (ss-eval-env el env))
          list))

(defun eval-application (op args env)
  (apply-proc (ss-eval-env op env)
              (eval-list-elements args env)))
```

Funkce `apply-proc`, která je teď na řadě, má aplikovat proceduru na (již vyhodnocené) argumenty. Jak se aplikace má provést, záleží na tom, zda je procedura uživatelsky definovaná, nebo jde o primitivum. Druhá varianta je jednodušší, proto se nejprve pustíme do ní.

Jako primitiva budeme chápat procedury, které jsou definovány jako funkce v Lispu (v tom, v němž programujeme náš interpret). Že je hodnota funkcí v Lispu, poznáme pomocí predikátu `functionp`.

```
(defun primitivep (proc)
  (functionp proc))
```

Funkce `apply-proc` se podívá, zda má aplikovat primitivum, nebo uživatelskou proceduru, a podle toho zavolá příslušnou pomocnou funkci:

```
(defun apply-proc (proc args)
  (if (primitivep proc)
      (apply-primitive proc args)
    (apply-user-proc proc args)))
```

Funkci `apply-primitive` napíšeme snadno:

```
(defun apply-primitive (proc args)
  (apply proc args))
```

Tím jsme náš interpret naučili aplikovat primitivní procedury:

```
CL-USER 18 > (ss-eval '(+ 1 2))
3

CL-USER 19 > (ss-eval '(cons (+ 1 2) (* 3 4)))
(3 . 12)
```

4.5 Speciální operátory `quote`, `if` a `define`

K zavedení speciálních operátorů je třeba napsat příslušné funkce, které volá funkce `eval-compound-expr`. U speciálních operátorů `quote` a `if` to bude jednoduché:

```
(defun eval-quote (args)
  (car args))
```

Test:

```
CL-USER 20 > (ss-eval '(quote a))
A

CL-USER 21 > (ss-eval '(quote (a b c)))
(A B C)

CL-USER 22 > (ss-eval ''x)
X
```

```
(defun eval-if (args env)
  (if (ss-eval-env (car args) env)
      (ss-eval-env (cadr args) env)
    (ss-eval-env (caddr args) env)))
```

Test:

```
CL-USER 1 > (ss-eval '(if 't 1 2))
1

CL-USER 2 > (ss-eval '(if 'nil 1 2))
2

CL-USER 3 > (ss-eval '(if (= 1 2) (+ 1 2) (* 3 4)))
12
```

Připomeňme, že speciální operátor `define` má hrát (zhruba) úlohu našeho `setf`: vytváří novou vazbu symbolu v globálním prostředí a nastavuje její hodnotu. Operátor hraje zvláštní roli. Jako jediný z našeho malého Scheme má vedlejší efekt. Proto ho musí mít i náš interpret. Bude to v tomto semestru jediná výjimka z pravidla nepoužívat makro `setf`.

Speciální operátor `define` naprogramujeme tak, aby vyměnil globální prostředí za jiné, které bude navíc obsahovat novou vazbu. Na nastavení nového globálního prostředí napíšeme funkci `set-initial-env`:

```
(let ((env (make-env (list (cons 'zero 0) (cons '+ #'+)
                           (cons '* #'*) (cons '= #'=)
                           (cons 'cons #'cons))
                     nil)))

  (defun initial-env ()
    env)

  (defun set-initial-env (value)
    (setf env value)))
```

Vidíme, že funkce vzniká ve stejném prostředí jako funkce `initial-env`, takže má přístup ke stejné vazbě symbolu `env`. Udělá něco, s čím jsme se ještě nesetkali: **hodnotu této vazby změní**. Funkce `initial-env` pak bude vracet tuto novou hodnotu.

Abychom si efekt funkce `set-initial-env` mohly rozumně vyzkoušet, napíšeme si pomocnou funkci `add-binding`, která k danému prostředí vrátí nové prostředí s přidanou vazbou.

```
(defun add-binding (env symbol value)
  (make-env (cons (cons symbol value) (env-bindings env))
            (env-parent env)))
```

Test:

```
CL-USER 5 > (initial-env)
(ENV ((ZERO . 0)
      (+ . #<Function + 40F0044AD4>) (* . #<Function * 40F006D2E4>)
      (= . #<Function = 40F006E964>)
      (CONS . #<Function CONS 40F008970C>))
     NIL)

CL-USER 6 > (set-initial-env (add-binding t '> #'>))
(ENV ((> . #<Function > 40F006E014>) (ZERO . 0)
      (+ . #<Function + 40F0044AD4>) (* . #<Function * 40F006D2E4>)
      (= . #<Function = 40F006E964>)
      (CONS . #<Function CONS 40F008970C>))
     NIL)
```

A konečně funkce `eval-define`:

```
(defun eval-define (args)
  (set-initial-env
   (add-binding t (car args) (ss-eval (cadr args))))
  (car args))
```

4.6 Uživatelské procedury

Nyní se pustíme do speciálního operátoru `lambda` a uživatelských procedur. Jak víme, uživatelské procedury obsahují tři údaje:

1. seznam parametrů,
2. tělo,
3. prostředí vzniku.

Budeme je reprezentovat datovou strukturou, která tyto údaje bude obsahovat. Struktura bude seznamem, který bude obsahovat tyto tři hodnoty a kromě nich symbol `proc` kvůli snadnější orientaci:

```
(proc params body env)
```

Konstruktor a selektory uživatelských procedur:

```
(defun make-proc (params body env)
  (list 'proc params body env))

(defun proc-params (proc)
  (cadr proc))

(defun proc-body (proc)
  (caddr proc))

(defun proc-env (proc)
  (cadddr proc))
```

Uživatel našeho malého Scheme bude procedury vytvářet speciálním operátorem `lambda`. V něm zadává seznam parametrů vytvářené procedury a její tělo. Prostředí, ve kterém je λ-výraz vyhodnocován, je také známo.

Jak jsme si stanovili ve funkci `eval-compound-expr`, o vyhodnocení λ-výrazů se má starat funkce `eval-lambda`. Té tedy stačí zavolat konstruktor `make-proc`:

```
(defun eval-lambda (args env)
  (make-proc (car args) (cadr args) env))
```

Poslední, co je v našem interpretu třeba udělat, je napsat funkci `apply-user-proc` na aplikaci uživatelské procedury. Zopakujme si, co se má během této aplikace stát:

1. Vytvořit nové prostředí, ve kterém budou parametry procedury navázány na argumenty a jehož předkem bude prostředí vzniku procedury.
2. V tomto prostředí vyhodnotit tělo procedury.

Když se zamyslíme, jak budeme funkci psát, narazíme na jeden izolovaný problém: jak vytvořit nové prostředí, máme-li zadán zvlášť seznam symbolů a zvlášť seznam hodnot, na které mají být navázány?

Je vhodné napsat si na to nový konstruktor prostředí. (Že ho píšeme až teď, je dáno tím, že při návrhu reprezentace prostředí jsme ještě nevěděli, že budeme potřebovat prostředí tímto způsobem vytvářet.)

```
(defun make-env-sv (symbols values parent)
  (make-env (mapcar #'cons symbols values)
            parent))
```

```
(defun apply-user-proc (proc args)
  (ss-eval-env (proc-body proc)
               (make-env-sv (proc-params proc)
                            args
                            (proc-env proc))))
```

5 Test

Jeden test interpretu (více na cvičení):

```
CL-USER 7 > (ss-eval '(define fact
                                (lambda (n)
                                  (if (> n 0)
                                      (* n (fact (+ n -1)))
                                    1))))
FACT

CL-USER 8 > (ss-eval '(fact 10))
3628800
```

Otázky a úkoly na cvičení

1. Přidejte do interpretu primitivum `print` na tisk, případně další primitiva. Udělejte to tak, že zavoláte funkci `set-initial-env`; nikoli editací zdrojového kódu.

2. Rozšiřte interpret o speciální operátory `progn` a `let`. Udělejte to co nejjednodušeji a pokud možno s použitím funkcí, které jsou v interpretu už napsané. Operátor `progn` můžete otestovat s primitivem na tisk z předchozího příkladu.

3. Doplňte do interpretu speciální operátor `let*`. Udělejte to pomocí už napsaného operátoru `let`.

4. Otestujte důkladně interpret na vybraných příkladech, které jste v tomto předmětu během semestru programovali. V případě potřeby si rozšiřte globální prostředí o nové vazby na primitiva.

5. Proč v testech funkce `eval-if` musely být symboly `t` a `nil` zakvotované?

6. Pokud dáme interpretu vyhodnotit symbol bez vazby, interpret se zacyklí. Kde a jak ho vylepšit, aby místo toho nahlásil chybu?

7. Vylepšete proceduru `eval-compound-expr`, aby pracovala elegantněji se speciálními operátory. Je například možné vytvořit si seznam speciálních operátorů a k nim příslušných vyhodnocovacích procedur. Procedura `eval-pair` by se do seznamu podívala, zda je symbol speciálním operátorem a pokud ano, aplikovala by příslušnou proceduru.

8. Tento příklad ukazuje, jaké problémy mohou vzniknout s mutacemi. Změňte definici funkce `ss-eval` takto:

```
(defun ss-eval (expr)
  (ss-eval-env expr (initial-env)))
```

   dále se ujistěte, že ve Schemovském globálním prostředí nemáte proceduru `fact`, pak ji definujte stejně, jako v příkladě na přednášce, a vysvětlete, proč nefunguje.