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 Scheme

Důležitá poznámka. Abyste mohli používat proceduru `set-car!`, je bohužel třeba opět změnit v DrRacket jazyk. Tentokrát potřebujeme jazyk `r5rs` s *vypnutou* volbou „Disallow redefinition of initial bindings“.

1 Úvod a zásady

Jako vyvrcholení našeho semestrálního kurzu napíšeme interpret programovacího jazyka Scheme. Je to přirozený krok: v prvních částech kurzu jsme se seznamovali s vyhodnocovacím procesem jazyka, který pracuje s **výrazy**, což jsou **atomy** (čísla, symboly, prázdný seznam, textové řetězce) a **seznamy**. V dalších částech jsme se učili s výrazy (zejména seznamy) pracovat. Teď tedy máme k dispozici všechny nástroje, abychom mohli vyhodnocovací proces naprogramovat.

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.

Zásady pro náš interpret:

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

Vstupním bodem našeho interpretu bude procedura `eval`, jejímž úkolem bude vyhodnotit zadaný výraz a vrátit jeho hodnotu:

```
> (eval '(+ 1 2))
3
> (eval '((lambda (x y) x) 1 2))
1
> (eval '(let ((x 1) (y 2))
           (+ x y)))
3
> (eval '(define proc (lambda (x y) (+ x y))))
> (eval '(proc 1 2))
3
```

Jak vidíme, procedura bude akceptovat jako argument výraz (obvykle neprázdný seznam). Ten musí podle pravidel vyhodnocovacího procesu postupně zpracovat a vrátit jako výsledek jeho hodnotu. (S výjimkou výrazu se speciálním operátorem `define`. Ten, jak víme, nevrací hodnotu, ale má vedlejší efekt.)

Takto navržená procedura `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 vhodně reprezentovat prostředí a umožnit proceduře `eval` v zadaném prostředí výrazy vyhodnocovat.

Proto bude procedura mít i druhý nepovinný parametr, kterým bude možné zadat (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:

```
> (eval '(+ x y) env)
3
> (eval '((lambda (x) (* x y)) 5) env)
10
```

Procedura bude napsána přesně podle (zjednodušených) zásad vyhodnocovacího procesu jazyka Scheme, jak jsme se o něm učili během semestru. Uvidíme tedy, že vyhodnocovací proces jazyka Scheme 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 velmi 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 procedur. Žádná procedura, kterou budeme psát, tedy pokud možno

nebude přímo řešit více než jeden problém. Pokud by měla, raději v jejím těle jen zavoláme jiné procedury.

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.

2 Procedura `eval`

Jak jsme řekli, bude vstupním bodem programu. Víme, že Scheme vyhodnocuje výrazy v daném prostředí. I když zatím nevíme, jak budeme prostředí reprezentovat, je jasné, že procedura `eval` je musí přijímat jako argument. Proto obecný tvar aplikace procedury je následující:

```
(eval expr env)
```

kde

expr je libovolný výraz a

env prostředí, ve kterém se má vyhodnotit.

Aby bylo možné proceduru používat jednoduše tak, jak je uvedené výše, bude druhý argument nepovinný. Procedura pak výraz vyhodnotí ve výchozím (globálním) prostředí. 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).

Výchozí prostředí uložíme do proměnné `initial-env`. (Zatím si do ní vložíme např. prázdný seznam.)

```
(define initial-env ())
```

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

Věrni naší zásadě psát procedury co nejjednodušeji si tedy na problémy vyhodnotit atom a vyhodnotit (neprázdný) seznam napíšeme později samostatné procedury (tedy dělíme složitý problém na dva jednodušší). První verze procedury `eval`:

```
(define (eval expr . env-lst)
  (let ((env (if (null? env-lst) initial-env (car env-lst))))
    (if (pair? expr)
        (eval-pair expr env)
        (eval-atom expr env))))
```

Výraz `(if (null? env-lst) initial-env (car env-lst))` zjišťuje prostředí, ve kterém se má výraz `expr` vyhodnotit. Pokud uživatel nezadal nepovinný argument, bude to prostředí `initial-env`, v opačném případě to bude zadaný nepovinný argument (který je prvvním prvkem seznamu `env-lst`).

Vidíme, že jsme ne zcela dodrželi předsevzetí, aby každá procedura řešila pouze jeden problém. V této verzi procedura `eval` řeší problémy dva: zjišťuje prostředí, ve kterém má vyhodnocení proběhnout a rozhoduje, zda se bude vyhodnocovat atom, nebo seznam. Je tedy prostor pro její vylepšení.

```
(define (optional-arg-val arg-lst default)
  (if (null? arg-lst) default (car arg-lst)))

(define (eval expr . env-lst)
  (let ((env (optional-arg-val env-lst initial-env)))
    (if (pair? expr)
        (eval-pair expr env)
        (eval-atom expr env))))
```

Tím jsme také napsali proceduru `optional-arg-val`, která nám pomůže pohodlněji pracovat s nepovinnými argumenty i v budoucnu.

3 Vyhodnocování neproměnných atomů

Proceduru `eval` zatím nemůžeme otestovat, protože žádná její větev není dosud implementována. Jednodušší bude zřejmě napsat proceduru `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:

```
(define (eval-atom atom env)
  (if (symbol? atom)
      (symbol-value atom env)
      atom))
```

(Využili jsme predikát `symbol?` jazyka Scheme, který zjišťuje, zda zadaná hodnota je symbol.)

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

```
> (eval 1)
1
> (eval #t)
#t
> (eval ())
()
```

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

4 Prostředí, vyhodnocování symbolů

Procedura `symbol-value` použitá výše má vrátit hodnotu symbolu v daném prostředí. Abychom ji mohli napsat (což bude naším dalším úkolem), musíme už pořádně definovat prostředí. Víme, že prostředí obsahuje následující informace:

1. Svého předka,
2. tabulku vazeb symbolů.

Předkem prostředí je opět prostředí. Pokud prostředí předka nemá, vyjádříme to hodnotou `#f`. 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 . 1) (b . 2))
```

představuje vazbu symbolu `a` na hodnotu `1` a symbolu `b` na hodnotu `2`. Se seznamy tohoto tvaru se hezky pracuje a Scheme má pro ně už připraveny procedury.

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

Když jsme si vyjasnili tvar struktury reprezentující prostředí, napíšeme nejprve příslušný konstruktor a selektory. Konstruktor `make-env` je napsán s nepovinným druhým parametrem. Pokud ho uživatel nezadá, bude předek nového prostředí nastaven na `#f`.

```
(define (make-env bindings . parent-lst)
  (list 'env bindings (optional-arg-val parent-lst #f)))

(define (env-bindings env)
  (cadr env))

(define (env-parent env)
  (caddr env))
```

(Procedury `cadr` a `caddr` jsou náhradou za procedury `second` a `third`, které jsem použil na přednášce. Nyní je použít nemůžu, protože v jazyce `r5rs` bohužel nejsou. Detaily o nich si zjistěte v dokumentaci. To se týká i dalších procedur Scheme, které tady budu bez vysvětlení používat.)

Procedura `symbol-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 předkovi prostředí.
3. Vrátit hodnotu nalezené vazby.

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

```
(define (symbol-value env symbol)
  (cdr (symbol-binding env symbol)))
```

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

```
(define (symbol-binding env symbol)
  (let ((bnd (symbol-binding-1 env symbol)))
    (if (pair? bnd)
        bnd
        (symbol-binding (env-parent env) symbol))))
```

Použili jsme proceduru `symbol-binding-1`, která hledá vazbu symbolu v daném prostředí a nedívá se rekurzivně do předka. Procedura je napsána pomocí procedury `assoc` (najděte si ji v dokumentaci), ale šlo by to samozřejmě i jinak, bez ní:

```
(define (symbol-binding-1 env symbol)
  (assoc symbol (env-bindings env)))
```

Nyní bychom měli pečlivě otestovat jednak proceduru `symbol-value` a jednak celý vylepšený interpret.

```
> (define env-1 (make-env '((a . 1))))
> env-1
(env ((a . 1)) #f)
> (define env-2 (make-env '((b . 2)) env-1))
> env-2
(env ((b . 2)) (env ((a . 1)) #f))
> (symbol-binding-1 'a env-1)
(a . 1)
> (symbol-binding-1 'a env-2)
#f
> (symbol-binding 'a env-2)
(a . 1)
> (symbol-value 'a env-2)
1
> (symbol-value 'b env-2)
2
```

```
> (eval 'a env-2)
1
> (eval 'b env-2)
2
```

Nyní můžeme také definovat lépe výchozí prostředí. Pro zajímavost si do něj uložíme například číslo 0 a několik primitiv:

```
(define initial-env (make-env (list (cons 'zero 0)
                                    (cons '+ +)
                                    (cons '* *)
                                    (cons '> >)
                                    (cons 'cons cons))))
```

Test:

```
> (eval 'zero)
0
> (eval '+)
#<procedure:+>
> (eval 'cons)
#<procedure:cons>
```

5 Vyhodnocování seznamu a aplikace primitiv

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

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 proceduru `eval-pair` pro základní speciální operátory: `quote`, `if`, `define` a `lambda`.

```
(define (eval-pair pair env)
  (let ((op (car pair))
        (args (cdr pair)))
    (cond ((eq? op 'quote) (eval-quote args env))
          ((eq? op 'if) (eval-if args env))
          ((eq? op 'define) (eval-define args env))
          ((eq? op 'lambda) (eval-lambda args env))
          (#t (eval-application op args env)))))
```

Procedury `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 v proceduře `eval-pair` a napsat příslušnou proceduru pro vyhodnocení. Jak už bylo řečeno, tento trochu těžkopádný způsob, který vyžaduje přepisování procedury `eval-pair`, lze vylepšit.

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

```
(define (eval-application op args env)
  (apply-proc (eval op env)
              (map (lambda (arg) (eval arg env))
                   args)))
```

V proceduře jsme použili proceduru `map`. Ta prochází seznam argumentů `args`, každý argument vyhodnotí procedurou `eval` v prostředí `env` a výsledky vrátí v seznamu.

S takto implementovanou procedurou `eval-application` bychom se ovšem neměli spokojit. Je vidět, že dělá příliš mnoho věcí. Přitom vyhodnocení všech prvků daného seznamu si zaslouží vlastní proceduru:

```
(define (eval-list-elements lst env)
  (map (lambda (el) (eval el env))
       lst))

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

Procedura `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 ve Scheme (v tom, v němž programujeme náš interpret). Že jde o takovou proceduru, poznáme pomocí predikátu `procedure?`.

```
(define (primitive? proc)
  (procedure? proc))
```

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

```
(define (apply-proc proc args)
  (if (primitive? proc)
      (apply-primitive proc args)
      (apply-user-proc proc args)))
```

Proceduru `apply-primitive` napíšeme snadno:

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

Tím jsme náš interpret naučili aplikovat primitivní procedury:

```
> (eval '(+ 1 2))
3
> (eval '(cons (+ 1 2) (* 3 4)))
(3 . 12)
```

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

K zavedení speciálních operátorů je třeba napsat příslušné procedury, které volá procedura `eval-pair`. U speciálních operátorů `quote` a `if` to bude jednoduché:

```
(define (eval-quote args env)
  (car args))
```

Test:

```
> (eval '(quote a))
a
> (eval '(quote (a b c)))
(a b c)
> (eval ''x)
x
```

```
(define (eval-if args env)
  (if (eval (car args) env)
      (eval (cadr args) env)
      (eval (caddr args) env)))
```

Test:

```
> (eval '(if #t 1 2))
1
```

```
> (eval '(if #f 1 2))
2
> (eval '(if (> 1 2) (+ 1 2) (* 3 4)))
12
```

Speciální operátor `define` hraje zvláštní roli. Jako jediný z našeho malého Scheme má vedlejší efekt. Proto ho musí mít i náš interpret. Speciální operátor `define` má upravit seznam vazeb výchozího prostředí. K tomu použije proceduru `set-car!` (kvůli které jsme museli změnit jazyk na `r5rs`):

```
> (define c (cons 1 2))
> c
(1 . 2)
> (set-car! c 3)
> c
(3 . 2)
```

S její pomocí napíšeme proceduru `eval-define`.

```
(define (add-binding! env symbol value)
  (set-car! (cdr env)
            (cons (cons symbol value)
                  (env-bindings env))))

(define (eval-define args env)
  (add-binding! initial-env (car args) (eval (cadr args))))
```

Procedura vytváří prostředí, jehož seznam vazeb vznikl přidáním jedné nové vazby k seznamu vazeb prostředí `initial-env`. Nové prostředí uloží pomocí speciálního operátoru `set!` do proměnné `initial-env`.

7 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 tedy reprezentovat seznamem, který bude obsahovat tyto tři hodnoty a kromě nich symbol `proc` kvůli snadnější orientaci:

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

Konstruktor a selektory uživatelských procedur:

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

(define (proc-params proc)
  (cadr proc))

(define (proc-body proc)
  (caddr proc))

(define (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 v proceduře `eval-pair`, o vyhodnocení λ-výrazů se má starat procedura `eval-lambda`. Té tedy stačí zavolat konstruktor `make-proc`:

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

Poslední, co je v našem interpretu třeba udělat, je napsat proceduru `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 proceduru budeme 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.)

```
(define (make-env-sv symbols values . parent-lst)
  (make-env (map cons symbols values)
            (optional-arg-val parent-lst #f)))
```

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

Jeden test (více na cvičení):

```
> (eval '(define fact
                 (lambda (n)
                   (if (> n 0) (* n (fact (+ n -1))) 1))))
> (eval '(fact 10))
3628800
```

Otázky a úkoly na cvičení

1. Otestujte důkladně interpret na příkladech, které jste v tomto předmětu během semestru programovali. V případě potřeby si rozšiřte výchozí prostředí v proměnné `initial-env` o nové vazby na primitiva.

2. Rozšiřte interpret o speciální operátory `begin` a `let`. Udělejte to co nejjednodušeji a pokud možno s použitím procedur, které jsou v interpretu už napsané.

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

4. Rozmyslete, jak byste programovali speciální operátor `letrec`. Bylo by možné udělat to bez vedlejšího efektu?

5. Vylepšete proceduru `eval-pair`, 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.

6. Vylepšete speciální operátor `define`, aby umožňoval definici procedur způsobem, který obvykle používáme (tj. aby jeho první argument mohl být seznam).