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

8. Vedlejší efekt

verze z 20. listopadu 2019

1 Co je vedlejší efekt

Až na dvě výjimky, všechny funkce, makra a speciální operátory, o kterých jsme mluvili, mají jediný účel: pomocí argumentů (ať už vyhodnocených nebo ne, podle toho, o jaký operátor jde), na které byly aplikovány, získat výsledek a ten vrátit jako hodnotu. K žádné jiné změně nedojde: pokud bychom vrácenou hodnotu ignorovali, nemáme jak zjistit, že aplikace proběhla.

To je v souladu s tím, jak funguje vyhodnocovací proces v Lispu. Během práce programu se **vyhodnocují symbolické výrazy**. Získaná hodnota výrazu se buď použije jako argument pro další výraz (v případě složených symbolických výrazů), nebo je už konečným výsledkem výpočtu. Přitom se nic jiného s programem nestane.

Už jsme se ale setkali se dvěma případy, které tomuto čistému pojetí vyhodnocování výrazů nevyhovují. Při následujícím vyhodnocení se stane něco navíc:

```
CL-USER 2 > (setf a 10)
10
```

Kromě toho, že výraz vrátí hodnotu `10`, ještě nastaví hodnotu proměnné `a` v globálním prostředí na `10`. Zatímco před jeho vyhodnocením by pokus vyhodnotit výraz `a` mohl skončit například chybou, nyní chybou neskončí a vrátí hodnotu `10`. Takové změně říkáme vedlejší efekt.

K *vedlejšímu efektu* vyhodnocení výrazu dojde v případě, že kromě vrácení hodnoty výraz způsobí nějakou vnější změnu. Tou změnou může být (jako v uvedeném příkladě) změna výchozího prostředí, nebo třeba tiskový nebo jiný výstup.

Prvnímu typu vedlejšího efektu můžeme říkat *interní vedlejší efekt*. Takový efekt způsobil nahoře uvedený výraz, může jím také být mutace hodnoty v datové struktuře nebo třeba definice funkce:

```
(defun exp2 (x)
  (* x x))
```

Před vyhodnocením tohoto výrazu funkce `exp2` nemusela vůbec existovat, po jeho vyhodnocení existuje a dělá to, co je výrazem určeno. Vyhodnocením výrazu jsme tedy vykonali interní vedlejší efekt.

Až na použití makra `setf`, a to jen k testovacím účelům v Listeneru, a makra `defun` k definici nových funkcí nebudeme v tomto semestru interní vedlejší efekt používat (jednu zcela mimořádnou výjimku uděláme v poslední přednášce). Budeme se mu věnovat příští semestr a ukážeme, k jakým nepříjemnostem může vést.

Za *externí vedlejší efekt* považujeme takový, který není možné zpětně programem zjistit. Může to být například tisk nebo grafický výstup na obrazovce. Externí vedlejší efekt je bezpečnější než interní a budeme se mu věnovat na této přednášce.

2 Textové řetězce a tisk

Textový řetězec (*string*) je hodnota, která je jednorozměrným polem libovolných znaků. Zapisuje se, jak bývá zvykem, do uvozovek a z pohledu vyhodnocovacího procesu jde o atom, který se vyhodnocuje na sebe podobně jako například čísla nebo symboly `t` a `nil`:

```
CL-USER 17 > "Dobrý den"
"Dobrý den"
```

Řetězce budeme používat jen na komunikaci s uživatelem. Nebudeme se hlouběji zabývat jejich strukturou a funkcemi, které s nimi pracují.

Textový řetězec se samozřejmě rovná sám sobě:

```
CL-USER 18 > (setf string "Dobrou noc")
"Dobrou noc"

CL-USER 19 > (eql string string)
T
```

Když ale přímo do zdrojového kódu zadáváme řetězec (tedy jako tzv. *literál*), může Lisp pokaždé vytvořit řetězec nový, takže se může stát toto:

```
CL-USER 20 > (eql "Já" "Já")
NIL
```

Jde totiž o dva různé řezězce, které jen obsahují stejné znaky. S něčím podobným jsme se už setkali u seznamů.

Jak víme, u symbolů k tomu nedochází: když do zdrojového kódu zapíšeme vícekrát stejný symbol, jde opravdu o týž symbol:

```
CL-USER 21 > (eql 'abc 'abc)
T
```

Neexistují totiž dva různé symboly téhož jména. Jméno symbolu je přitom textový řetězec a dá se zjistit funkcí `symbol-name`:

```
CL-USER 22 > (symbol-name 'symbol)
"SYMBOL"
```

(Výsledkem této aplikace je opravdu řetězec; poznáme to podle uvozovek.) To jen pro úplnost, tuto funkci nebudeme potřebovat.

Textové řetězce stejně jako ostatní hodnoty můžeme *tisknout* do nějakého výstupu (Listeneru, souboru apod.). To je příklad externího vedlejšího efektu, jak jsme o něm mluvili před chvílí. Nejjednodušší způsob je použít funkci `print`:

```
CL-USER 24 > (print 1)

1
1

CL-USER 25 > (print nil)

NIL
NIL

CL-USER 26 > (print "Dobrý den")

"Dobrý den"
"Dobrý den"
```

Funkce (v nejjednodušším případě) přijímá jeden argument. Funkce přejde v Listeneru na nový řádek a pak do něj vytiskne svůj argument. Nakonec argument vrátí jako svou hodnotu.

Proto jsme v Listeneru viděli hodnotu dvakrát. Poprvé vytištěnou (všimněte si prázdného řádku před tiskem) a podruhé jako výsledek aplikace.

Podobně funguje funkce `princ`. Hlavní rozdíly mezi ní a funkcí `print` jsou, že před tiskem nepřechází na nový řádek a že u řetězců netiskne uvozovky:

```
CL-USER 27 > (princ 2)
2
2

CL-USER 28 > (princ t)
T
T
```

```
CL-USER 29 > (princ "Dobrý den")
Dobrý den
"Dobrý den"
```

3 Bloky

Použitím externího vedlejšího efektu se částečně dostáváme k *imperativnímu stylu programování*. V něm se program skládá nikoli z **výrazů**, které se vyhodnocují, ale z **příkazů**, které se vykonávají. U nás zůstaneme stále u pojmu výrazu, z nichž některé budou ovšem roli příkazů hrát. Vzhledem k tomu, že budeme používat pouze externí vedlejší efekt, stále ale budeme zůstávat hlavně ve funkcionálním programovacím stylu.

Při používání vedlejšího efektu je často třeba vykonat několik výrazů za sebou (tzv. sekvenčně). K tomu slouží tzv. *bloky* kódu. Tělo funkcí i operátorů `let` a `let*` mohou být takovým blokem.

```
(defun printing-exp2 (n)
  (princ "Druhá mocnina čísla ")
  (princ n)
  (princ " je ")
  (princ (* n n)))
```

Test:

```
CL-USER 30 > (printing-exp2 5)
Druhá mocnina čísla 5 je 25
25
```

Funkce vytiskla, co jsme chtěli, a navíc vrátila hodnotu `25`, protože to je návratová hodnota naposledy vyhodnocovaného výrazu v bloku, tj. výrazu `(princ (* n n))`.

Tak totiž bloky kódu fungují: vyhodnotí postupně všechny své výrazy a vrátí hodnotu posledního. Hodnoty ostatních výrazů se ignorují; vyhodnotit je tedy mělo smysl jen kvůli vedlejšímu efektu. (Pokud nebereme v úvahu případný výskyt chyby.)

Řekněme, že stále chceme, aby funkce `printing-exp2` vrátila druhou mocninu argumentu jako výsledek, ale navíc chceme, aby vytisknutá věta končila jak se patří tečkou. Využijeme toho, že speciální operátor `let` může také v těle obsahovat blok:

```
(defun printing-exp2 (n)
  (let ((result (* n n)))
    (princ "Druhá mocnina čísla ")
```

```
    (princ n)
    (princ " je ")
    (princ result)
    (princ ".")
    result))
```

Test:

```
CL-USER 31 > (printing-exp2 12)
Druhá mocnina čísla 12 je 144.
144
```

Poznamenejme, že v Lispu je možné tisknout delší informace mnohem jednodušším způsobem, než jak jsme to udělali my. Nám to takto ale bude stačit.

Spíše se teď zaměřme na návratovou hodnotu funkce. Vidíme, že v posledním testu funkce vytiskla text `"Druhá mocnina čísla 12 je 144."` a vrátila jako výsledek hodnotu `144`. V Listeneru nejde vrácenou hodnotu od vytištěného textu rozeznat, ale musíme chápat, jaký je mezi nimi rozdíl.

Jelikož funkce `printing-exp2` vrací hodnotu, můžeme ji použít například při výpočtu součtu druhých mocnin dvou čísel, například když chceme zjistit délku přepony pravoúhlého trojúhelníka z délek odvěsen:

```
(defun hypotenuse (a b)
  (sqrt (+ (printing-exp2 a)
           (printing-exp2 b))))
```

Test odhalí, že tisk můžeme vylepšit:

```
CL-USER 32 > (hypotenuse 3 4)
Druhá mocnina čísla 3 je 9.Druhá mocnina čísla 4 je 16.
5.0
```

Uděláme to podobně jako funkce `print`: ve funkci `printing-exp2` na začátku tisku přejdeme na nový řádek, a to pomocí funkce `terpri`. Pro jistotu na konci tisku ještě přidáme mezeru (to ve skutečnosti funkce `print` taky dělá).

```
(defun printing-exp2 (n)
  (let ((result (* n n)))
    (terpri)
    (princ "Druhá mocnina čísla ")
    (princ n)
    (princ " je ")
```

```
    (princ result)
    (princ ". ")
    result))
```

A test:

```
CL-USER 33 > (hypotenuse 5 12)

Druhá mocnina čísla 5 je 25.
Druhá mocnina čísla 12 je 144.
13.0
```

Při práci s vedlejším efektem se někdy hodí vytvářet blok explicitně. K tomu slouží speciální operátor `progn`. Ten akceptuje libovolný nenulový počet argumentů. Při vyhodnocení výrazu

```
(progn expr1 expr2 ... exprn)
```

se postupně (zleva doprava) vyhodnotí všechny výrazy ***expr1***, ***expr2***, ..., ***exprn***, hodnota posledního (tedy ***exprn***) se vrátí jako výsledek. Návratové hodnoty předchozích výrazů se ignorují.

Na závěr ukázka funkce, která vytiskne všechna nezáporná celá čísla menší než zadané číslo a nakonec vrátí `nil`. Funkce používá speciální operátor `progn`.

```
(defun print-numbers (n)
  (if (<= n 0)
      nil
    (progn
      (print (- n 1))
      (print-numbers (- n 1)))))
```

Test:

```
CL-USER 35 > (print-numbers 10)

9
8
7
6
5
4
3
```

```
2
1
0
NIL
```

Kdybychom chtěli čísla vytisknout v opačném pořadí (tj. od nuly nahoru), museli bychom použít pomocnou funkci:

```
(defun print-numbers-from-0 (n)
  (print-numbers-from 0 n))

(defun print-numbers-from (m n)
  (if (>= m n)
      nil
    (progn
      (print m)
      (print-numbers-from (+ m 1) n))))
```

Test:

```
CL-USER 36 > (print-numbers-from-0 10)

0
1
2
3
4
5
6
7
8
9
NIL
```

4 Želví grafika

Jak bylo řečeno, externí vedlejší efekt může představovat i kreslení do okna. Ukážeme si jednoduchý způsob kreslení pomocí tzv. **želví grafiky**. Bude pro nás představovat i užitečné cvičení na rekurzi.

Na webu předmětu máte k dispozici knihovnu `turtles` pro LispWorks, která základní funkce želví grafiky implementuje.

Instalace knihovny

Knihovna se instaluje načtením jejího souboru `load.lisp`. Načíst soubor můžeme, jak víme, dvojím způsobem. Můžeme v menu zvolit položku "Load..."(nesmí být nahoře okno editoru, to by se načetl aktuálně otevřený soubor) nebo můžeme soubor `load.lisp` otevřít a pak ho celý vyhodnotit.

Funkce a makro knihovny

Funkce `turtle:start`

```
(turtle:start)
```

Otevře okno s želví grafikou.

Funkce `turtle:draw`

```
(turtle:draw pixels)
```

Želva se posune ve svém směru o délku ***pixels***. Přitom nakreslí čáru.

Funkce `turtle:move`

```
(turtle:move pixels)
```

Želva se posune ve svém směru o délku ***pixels*** aniž by kreslila čáru.

Funkce `turtle:turn`

```
(turtle:turn angle)
```

Želva se na místě otočí proti směru hodinových ručiček o úhel ***angle***. Přitom nic nekreslí. Úhel se udává v násobcích π, může být záporný. K pohodlnějšímu používání jsou definovány proměnné `pi/2`, `-pi/2`, `pi/4`, `-pi/4`.

Funkce `turtle:clear`

```
(turtle:clear)
```

Okno se smaže a želva se umístí do počáteční pozice: do středu okna otočená vpravo.

Makro `turtle:excursion`

```
(turtle:excursion
  expr1
  expr2
    ⋮
  exprn)
```

Výrazy *expr1*, *expr2*, ..., *exprn* se postupně (sekvenčně) vyhodnotí. Potom se želva automaticky vrátí do pozice před vyhodnocením výrazu *expr1*.

Na přednášce jsme ukázali několik příkladů použití knihovny. Příklady jsou v kódu k přednášce, který máte k dispozici.

Otázky a úkoly na cvičení

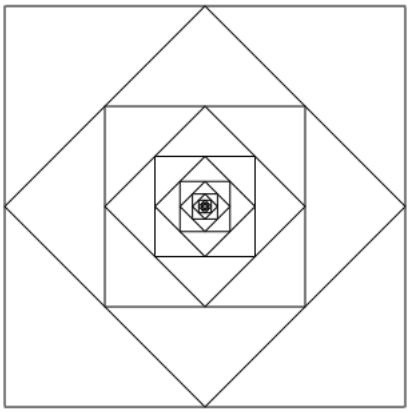
1. Vysvětlete, proč operátor `turtle:excursion` musí být makro.
2. Při popisu vyhodnocovacího procesu jsme neříkali, v jakém pořadí se před aplikací funkce vyhodnocují argumenty. V případě, že používáme vedlejší efekt, může mít pořadí význam. Navrhněte a vyzkoušejte způsob, jak je zjistit. (Mějte ovšem na paměti, že takové experimentální zjištění nám nedává jistotu.)
3. Napište funkci `printing-fact` na výpočet faktoriálu, která tiskne informaci o svých argumentech a vráceném výsledku. Výpočet faktoriálu 5 by tak mohl tisknout toto:

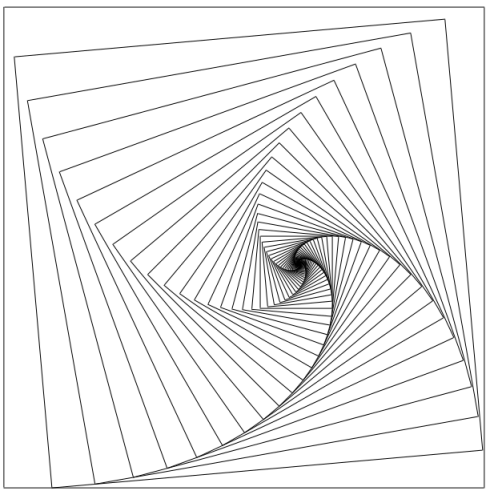
```
printing-fact aplikován na 5
printing-fact aplikován na 4
printing-fact aplikován na 3
printing-fact aplikován na 2
printing-fact aplikován na 1
printing-fact aplikován na 0
printing-fact vrátil 1
printing-fact vrátil 1
printing-fact vrátil 2
printing-fact vrátil 6
printing-fact vrátil 24
printing-fact vrátil 120
```

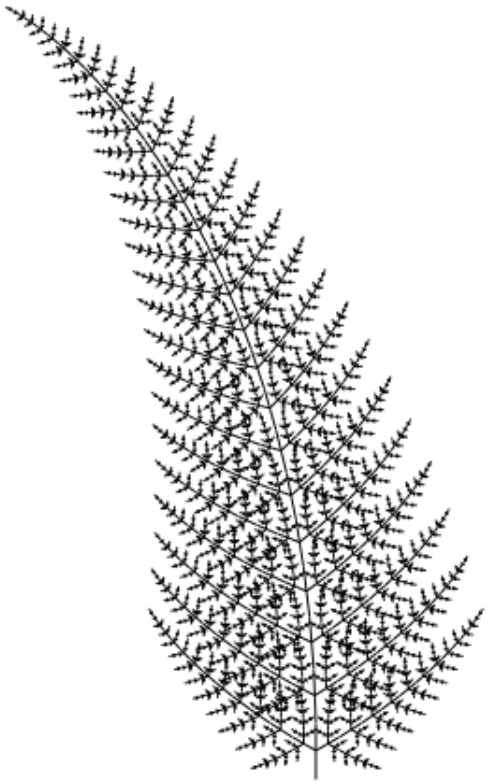
4. Vylepšete funkci tak, aby tiskla toto:

```
printing-fact aplikován na 5
 printing-fact aplikován na 4
  printing-fact aplikován na 3
   printing-fact aplikován na 2
    printing-fact aplikován na 1
     printing-fact aplikován na 0
     printing-fact vrátil 1
    printing-fact vrátil 1
   printing-fact vrátil 2
  printing-fact vrátil 6
 printing-fact vrátil 24
printing-fact vrátil 120
```

5. Napište funkce pro želví grafiku, které nakreslí obrázky podobné následujícím obrázkům:







6. Napište funkci, která pomocí želví grafiky nakreslí strukturu daného binárního stromu. Tím, že má nakreslit strukturu, se myslí, že pomocí čar nakreslí hierarchii uzlů a nebude vykreslovat jejich hodnoty.