Paradigmata programování 1

Program, jeho syntax a sémantika

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 1

Paradigmata programování

Přednášející:

doc. RNDr. Vilém Vychodil, Ph.D.
e-mail: vilem.vychodil@upol.cz
www: http://vychodil.inf.upol.cz/

Konzultační hodiny (viz webové stránky)

Zdroje:

- učební texty, slidy, poznámky, videozáznamy přednášek
- http://vychodil.inf.upol.cz/kmi/pp1/

Udělení zápočtu – *v kompetenci cvičících*

Přehled kursu

1. Program, jeho syntax a sémantika
2. Vytváření abstrakcí pomocí procedur
3. Lokální vazby a definice
4. Tečkové páry, symbolická data a kvotování
5. Seznamy
6. Explicitní aplikace a vyhodnocování
7. Akumulace
8. Rekurze a indukce
9. Hloubková rekurze na seznamech
10. Kombinatorika na seznamech, reprezentace stromů a množin
11. Kvazikvotování a manipulace se symbolickými výrazy
12. Čistě funkcionální interpret Scheme

Literatura

Závazná literatura:


Konečný J., Vychodil V.: *Paradigmata programování 1A, 1B*
http://vychodil.inf.upol.cz/download/text-books/pp1a.pdf
http://vychodil.inf.upol.cz/download/text-books/pp1b.pdf

Doporučená literatura:


Sperber M., Dybvig R., Flatt M., Van Straaten A., Findler R., Matthews J.: Revised[6] Report on the Algorithmic Language Scheme. *Journal of Functional Programming* **19**(S1)2009, 1–301.


Abelson H., Sussman G. J.: *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 2nd edition, 1986.


Felleisen M., Findler R. B., Flatt M., Krishnamurthi S.: *How to Design Programs: An Introduction to Computing and Programming.* The MIT Press, Cambridge, Massachusetts, 2001.

Přednáška 1: Přehled

1. Úvodní pojmy:
   - programovací jazyk, program, výpočetní proces,
   - syntaxe a sémantika programu,
   - přehled základních paradigmat programování.
2. Jazyk Scheme:
   - symbolické výrazy a syntaxe jazyka Scheme,
   - abstraktní interpret jazyka Scheme,
   - primitivní procedury a jejich aplikace,
   - rozšíření jazyka o speciální formy,
   - vytváření abstrakcí pojmenováním hodnot.

Výpočetní proces a program

Výpočetní proces

- aktivní (dynamická) entita
- počátek, konec, prováděn v elementárních krocích
- vstup $\Longmapsto$ výstup + vedlejší efekty

Program

- pasivní (statická) entita; soubor na disku (program = data)
- výpočetní proces = vykonávání programu

Programování

- tvůrčí činnost vytváření programu; programátor
- programovací jazyk = soubor pravidel v souladu s kterými je vytvořen program

Jak psát program tak, abychom vytvořili zamýšlený výpočetní proces?

Paradigmata programování

paradigma = styl

Účel kursů PP:

- seznámení se základními programovací styly
- vytváření programů s využitím různých stylů programování
- zkoumání různých typů výpočetních procesů a jejich efektivity
- zkoumání efektivity programování (chybovost)
- problematika interpretace programů

Možné postupy:

1. pro každé hlavní paradigma zvolíme typický jazyk
2. jeden (multiparadigmový) jazyk

Program a algoritmus

Program vs. algoritmus:

- dva pojmy algoritmus:
  1. naivní pojem algoritmus (místy „ošidné“)
  2. formální pojem algoritmus (zatím „příliš složité“)
- pro nás: algoritmus = takový *program*, že příslušný výpočetní proces pro libovolný vstup ukončí svoji činnost po konečně mnoha krocích

Více v kursech:

- algoritmická matematika
- formální jazyky a automaty
- vyčíslitelnost a složitost

Rovnocennost programovacích jazyků

Problém „volby programovacího jazyka“: Je jazyk A lepší než jazyk B?

- z pohledu možnosti řešení problémů jsou všechny „rozumné“ programovací jazyky rovnocenné (stejně silné)
- pojem *Turingovsky úplný jazyk* (Alan Turing)

Pozor ale:

- *různé jazyky poskytují různý komfort při programování*
- nezanedbatelný aspekt (**!!**)
- vyšší míra abstrakce $\Longmapsto$ vyšší komfort pro uživatele (programátora)
- extrémní příklad: jazyk `brainfuck` (pouze 8 elementárních instrukcí)

Více v kursech:

- formální jazyky a automaty
- vyčíslitelnost a složitost

Základní klasifikace programovacích jazyků

Programovatelné vs. neprogramovatelné počítače

- neprogramovatelné (–1945)
- programovatelné (cca 1945–);
  zajímavý přehled na `http://damol.info/12/10/04/holmes/`

Nižší programovací jazyky

- kódy stroje (vykonávané přímo procesorem)
- assemblery (mnemotechnické zkratky instrukcí, návěstí)
- autokódy, bajtkódy, ...

Vyšší programovací jazyky

- vyšší míra abstrakce (aritmetické operace, podprogramy, funkce, cykly)
- velké množství jazyků podporující různá paradigmata programování
- celá řada výhod: čitelnost programu, knihovny, přenositelnost, ...

Vytváření výpočetních procesů

kód stroje = výpočetní proces vytváří procesor

v ostatních případech:

- **interpretace** – prováděna **interpretem** (speciální program)
  - výrazy v programu postupně načítány
  - po načtení výrazu interpret vygeneruje příslušný výpočetní proces
- **překlad** (**kompilace**) – prováděna **překladačem** (**kompilátorem**, spec. prog.)
  program je načten celý a je vyprodukován ekvivalentní kód v jiném jazyku:

  Rozlišujeme:
  - překlad do kódu stroje
  - překlad do assembleru
  - překlad do (jiného) nižšího jazyka (bajtkód)
  - překlad do (jiného) vyššího jazyka (jazyk C)
- hybridní přístupy (např.: *just in time compilers*)

Syntax a sémantika programu

Dva (zcela jiné) pojmy:

- **syntax** – říká, jak program vypadá (jak se zapisuje)
- **sémantika** – říká, jaký má program význam (co dělá příslušný výp. proces)

SYNTAX $\neq$ SÉMANTIKA

- výraz: `03/05/2010` (možný zápis data, různé interpretace)
- výraz: `X=Y+3` (různý význam v jazycích C, Pascal, Metapost, Prolog)

Chyby v programech:

- **syntaktické chyby** – chyby v zápisu programu (snadno odstranitelné)
- **sémantické chyby**:
  - zjištěné během překladu / před interpretací (např. kolize typů)
  - za běhu programu (noční můra všech programátorů, Mars Climate Orbiter)
- chyby dělají i zkušení programátoři (**!!**)

Příklad (Notace zápisu aritmetických výrazů)

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

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

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

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

  Příklad: `2 3 5 + *`

Základní paradigmata programování

- **procedurální:** výpočet = sekvenční provádění procedur
  významný rys: přiřazovací příkaz
  teoretický model: RAM stroj (John von Neumann)
  jazyky: Fortran, Algol, Pascal, C
- **funkcionální:** výpočet = postupná aplikace funkcí
  významný rys: funkce vyšších řádů
  teoretický model: λ-kalkul (Alonzo Church)
  jazyky: Scheme, Common LISP, Haskel, ML
- **logické:** výpočet = automatická dedukce
  významný rys: deklarativní charakter
  teoretický model: matematická logika, princip rezoluce (Robinson)
  jazyky: Prolog, Datalog

Základní paradigmata programování

Procedurální jazyky dělíme na:

- **naivní:**
  významný rys: nekoncepční
  jazyky: Basic
- **nestrukturované:**
  významný rys: explicitní příkaz skoku „přejdi na řádek“
  jazyky: Fortran
- **strukturované:**
  významný rys: skok nahrazen podmíněnými cykly
  jazyky: Algol, Pascal, C

Další významná paradigmata:

- paralelní (více souběžných výpočetních procesů)
- objektové (interakce entit, které mají vnitřní stav)

Jazyk Scheme

Co je Scheme?

- multiparadigmový jazyk
- preferované paradigma je *funkcionální*
- specifikován v revidovaném reportu R^6RS
- programy obvykle interpretovány (existují výjimky)
- volně dostupné interprety: GUILE, MIT Scheme, Bigloo, ...

Racket: `http://racket-lang.org/`

Symbolické výrazy a programy

program (v jazyku Scheme) = konečná posloupnost **symbolických výrazů**

Definice (symbolický výraz, S-výraz)

1. Každé *číslo* je symbolický výraz
   (zapisujeme `12`, `-10`, `2/3`, `12.45`, `4.32e-20`, `-5i`, `2+3i`, a pod.);
2. každý *symbol* je symbolický výraz
   (zapisujeme `sqrt`, `+`, `quotient`, `even?`, `muj-symbol`, `++?4tt`, a pod.);
3. jsou-li $e_1, e_2, \ldots, e_n$ symbolické výrazy (pro $n \geq 1$),
   pak výraz ve tvaru $(e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n)$ je symbolický výraz zvaný *seznam*.

- není definice kruhem (**!!**)
- složitější seznamy se definují pomocí jednodušších
- pro $(e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n)$ je n **délka seznamu**

Sémantika jazyka Scheme

S-výraz = syntaktický pojem

výpočetní proces = vzniká postupným vyhodnocováním S-výrazů

Vyhodnocení S-výrazu; **neformálně**, **!!**

- Každé *číslo* se vyhodnotí na *hodnotu*, kterou reprezentuje.
  (čísla se vyhodnocují na „sebe sama“)
- Každý *symbol* se vyhodnotí na svojí *aktuální vazbu*.
  (symboly se vyhodnocují na hodnoty, které jsou na ně navázané)
- Každý *seznam* $(e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n)$ se vyhodnotí:
  1. vyhodnotí se první prvek seznamu (hlava) $\Longrightarrow$ *procedura* (*operace*),
  2. vyhodnotí se zbylé prvky seznamu (tělo) $e_2, \ldots, e_n$,
  3. procedura je *aplikována* s výsledky vyhodnocení $e_2, \ldots, e_n$.

Zbývá upřesnit: *hodnota*, *vazba symbolu*, *procedura*, *aplikace*

Elementy jazyka, interní a externí reprezentace

Reader

- podprogram interpretu Scheme
- načítá S-výrazy a převádí je na jejich **interní reprezentace**

Elementy jazyka = hodnoty

- čísla (efektivní vnitřní reprezentace)
- symboly (tabulky symbolů – jména)
- seznamy (dynamický lineární spojový seznam)
- primitivní procedury (zabudované v interpretu, koncept „černé skříňky“)

Printer

- podprogram interpretu Scheme
- pro elementy vrací jejich textovou **externí reprezentace**
- externí reprezentace primitivních procedur není čitelná readerem (rys)

Prostředí: tabulka (počátečních) vazeb symbolů

Prostředí:

symbol	*element*
E_1	F_1
E_2	F_2
$\vdots$	$\vdots$
E_k	F_k
$\vdots$	$\vdots$

Možnosti:

- *aktuální vazba* symbolu E_i v prostředí je F_i
- *aktuální vazba* symbolu E *není definovaná*

Abstraktní interpret jazyka Scheme

Interpretace programů probíhá v cyklu REPL:

- **read**
  - prázdný vstup – činnost interpretu končí, nebo:
  - načten první vstupní S-výraz E a převeden do interní reprezentace F
  - možnost vzniku syntaktické chyby
- **eval**
  - F se vyhodnotí na G (netriviální krok)
- **print**
  - převod elementu G do externí reprezentace
  - vytištění externí reprezentace
- **loop**

Aplikace primitivních procedur

- jádro vyhodnocování elementů
- aplikací procedur vzniká kvalitativní výpočetní proces

Definice (aplikace primitivních procedur)

Nechť E je primitivní procedura a $E_1, \ldots, E_n$ jsou libovolné elementy jazyka.

Výsledek **aplikace** E na **argumenty** $E_1, \ldots, E_n$ v tomto pořadí:

$$\mathrm{Apply}[E, E_1, \ldots, E_n].$$

Pokud je výsledkem této aplikace element F, pak píšeme $\mathrm{Apply}[E, E_1, \ldots, E_n] = F$.

Příklad: $\mathrm{Apply}[\text{„primitivní procedura násobení“}, 2, 5, 10] = 100.$

Definice (vyhodnocení elementu E)

Výsledek vyhodnocení elementu E, značeno $\mathrm{Eval}[E]$, je definován:

(A) Pokud je E *číslo*, pak $\mathrm{Eval}[E] := E$.

(B) Pokud je E *symbol*, pak:

(B.1) Pokud E *má aktuální vazbu* F, pak $\mathrm{Eval}[E] := F$.

(B.e) Pokud E *nemá vazbu*, pak „CHYBA: Symbol E nemá vazbu.“.

(C) Pokud je E *seznam* tvaru $(E_1\ \ E_2 \cdots E_n)$, pak pro $F_1 := \mathrm{Eval}[E_1]$ (nejprve vyhodnotíme první prvek seznamu, jeho hodnota je F_1) a rozlišujeme:

(C.1) Pokud je F_1 *procedura*, pak se v nespecifikovaném pořadí vyhodnotí zbylé prvky seznamu E, to jest uvažujeme $F_2 := \mathrm{Eval}[E_2], \ldots, F_n := \mathrm{Eval}[E_n]$. Pak $\mathrm{Eval}[E] := \mathrm{Apply}[F_1, F_2, \ldots, F_n]$ (výsledkem vyhodnocení E je element vzniklý *aplikací procedury* F_1 *na argumenty* $F_2, \ldots, F_n$).

(C.e) Pokud F_1 není procedura: „CHYBA: Nelze provést aplikaci: první prvek seznamu E se nevyhodnotil na proceduru.“.

(D) Ve všech ostatních případech klademe $\mathrm{Eval}[E] := E$.

Příklad (Vybrané primitivní funkce: aritmetika)

Sčítání:

```
(+ 1 2 3)        ⟹ 6
(+ (+ 1 2) 3)    ⟹ 6
(+ 1 (+ 2 3))    ⟹ 6
(+ 20)           ⟹ 20
(+)              ⟹ 0
```

Odčítání:

```
(- 1 2)          ⟹ -1
(- 1 2 3)        ⟹ -4
(- (- 1 2) 3)    ⟹ -4
(- 1 (- 2 3))    ⟹ 2
(- 1)            ⟹ -1
```

Násobení:

```
(* 4 5 6)        ⟹ 120
(* (* 4 5) 6)    ⟹ 120
(* 4 (* 5 6))    ⟹ 120
(* 4)            ⟹ 4
(*)              ⟹ 1
```

Dělení:

```
(/ 4 5)          ⟹ 4/5
(/ 4 5 6)        ⟹ 2/15
(/ (/ 4 5) 6)    ⟹ 2/15
(/ 4 (/ 5 6))    ⟹ 24/5
(/ 4)            ⟹ 1/4
```

Příklad (Zajímavé rysy aritmetiky ve Scheme)

Racionální čísla a čísla v semilogaritmickém tvaru:

```
(/ 2 3)            ⟹  2/3
(/ 2 3.0)          ⟹  0.6666666666666666
(* 1.0 (/ 2 3))   ⟹  0.6666666666666666
(sqrt 4)           ⟹  2
(* -2e-20 4e40)   ⟹  -8e+20
```

Libovolně přesná racionální čísla: `600479950316066 1/9007199254740992`

Komplexní čísla: `3+2i`, `-3-2i`, `+2i`, `-2i`, `-2/3+4/5i`, `0.4e10+2i`, `2/3-0.6i`

Implicitní přetypování (koerce) a explicitní přetypování:

```
(* 2/3 1.0)                          ⟹  0.6666666666666666
(exact->inexact 2/3)                 ⟹  0.6666666666666666
(inexact->exact 0.6666666666666666)  ⟹  600479···/900719···
(rationalize 600479/900719 1/1000)   ⟹  2/3
```

Vytváření abstrakcí pojmenováním hodnot

Motivační příklad:

- účetní program počítající se sazbou DPH
- jak vyřešit problém „změny sazby DPH“
- vede na pojmenovanou hodnotu: symbol `vat-value` s vazbou `10` nebo `20`

Chceme možnost definovat vazbu symbolu:

```
(define vat-value 20)
```

Problém: Na `define` nemůže být navázána procedura. (**!!**)

Nutné rozšířit abstraktní interpret o *speciální formy* (nový element jazyka)

Definice (doplnění vyhodnocovacího procesu o speciální formy)

Výsledek vyhodnocení elementu E, značeno $\mathrm{Eval}[E]$, je definován:

(A) $\cdots$

(B) $\cdots$

(C) Pokud je E *seznam* tvaru $(E_1\ E_2 \cdots E_n)$, pak pro $F_1 := \mathrm{Eval}[E_1]$ (nejprve vyhodnotíme první prvek seznamu, jeho hodnota je F_1) a rozlišujeme:

- (C.1) Pokud je F_1 *procedura*, pak $\cdots$
- (C.2) Pokud je F_1 *speciální forma*, pak $\mathrm{Eval}[E] := \mathrm{Apply}[F_1, E_2, \ldots, E_n]$.
- (C.e) Pokud F_1 není procedura ani speciální forma: „CHYBA: Nelze provést aplikaci: první prvek seznamu E se nevyhodnotil na proceduru ani na speciální formu.“.

(D) $\cdots$

Každá speciální forma si sama určuje:

- jaké argumenty a jakém pořadí budou vyhodnoceny; rozdíl proti procedurám (**!!**)

Definice (speciální forma `define`)

Speciální forma `define` se používá se dvěma argumenty ve tvaru:

`(define` $\langle jméno \rangle$ $\langle výraz \rangle$`)`

Postup aplikace speciální formy:

1. ověří se, jestli je $\langle jméno \rangle$ symbol (jinak „CHYBA: První výraz musí být symbol.“)
2. $F := \mathrm{Eval}[\langle výraz \rangle]$
3. v prostředí je vytvořena nová vazba symbolu $\langle jméno \rangle$ na element F
4. pokud již $\langle jméno \rangle$ měl vazbu, původní vazba je nahrazena elementem F
5. výsledkem aplikace je *nedefinované hodnota* (nový typ elementu)

U každé zavedené speciální formy musíme definovat:

- **syntax** – v jakém tvaru se forma používá,
- **sémantiku** – jak probíhá její interpretace, co je výsledkem a vedlejším efektem.

Příklad (příklady použití `define`)

```
(define a 20)
(* 2 a)          ⟼  40
(sqrt (+ a 5))   ⟼  5

(define a (+ a 1))
a                ⟼  21
(define b (+ 4 a))
a                ⟼  21
b                ⟼  25

(define a 666)
a                ⟼  666
b                ⟼  25

(define + -)
(+ 20)           ⟼  -20
```

Pravdivostní hodnoty

- `#t` – pravda (angl. *true*)
- `#f` – nepravda (angl. *false*)
- **predikáty** – procedury vracející pravdivostní hodnoty jako výsledky

Příklad

```
#t          ⟼  #t
#f          ⟼  #f
(<= 2 3)    ⟼  #t
(< 2 3)     ⟼  #t
(= 2 3)     ⟼  #f
(= 2 2.0)   ⟼  #t
(= 0.5 1/2) ⟼  #t
(>= 3 3)    ⟼  #t
```

Definice (speciální forma `if`)

Speciální forma `if` se používá se dvěma argumenty ve tvaru:

`(if` $\langle test \rangle$ $\langle důsledek \rangle$ $\langle náhradník \rangle$`)`,

přitom $\langle náhradník \rangle$ je *nepovinný argument* a nemusí být uveden. Aplikace:

1. nejprve vyhodnocen argument $\langle test \rangle$
2. pokud je výsledná hodnota různá od `#f`, pak je výsledkem aplikace hodnota vzniklá vyhodnocením argumentu $\langle důsledek \rangle$
3. pokud je výsledná hodnota `#f`, pak:
   - pokud je $\langle náhradník \rangle$ přítomen, pak je výsledkem aplikace výsledek vyhodnocení argumentu $\langle náhradník \rangle$
   - pokud není $\langle náhradník \rangle$ přítomen, pak je výsledek aplikace nedefinovaný.

Podmíněné výrazy

Zobecněné pravdivostní hodnoty:

- pravda = vše kromě `#f`
- nepravda = `#f`

Důsledek: jakýkoliv element může být použitý jako „pravdivostní hodnota“

Pozor: `if` není procedura (!!)

- `if` vyhodnocuje druhý a třetí argument v závislosti na prvním
- teoreticky jej lze chápat jako proceduru (do budoucna neudržitelné)

Příklad (Podmíněné výrazy)

```
(define a 10)
(define b 13)

(if (> a b) a b)  ⟼  13

(if (<= a b)
    (if (= a b)
        a
        (- a))
    #f)  ⟼  -10

(if 1 2 3)  ⟼  2

((if #f + -) 10 20)  ⟼  -10
```