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 3 ◇ poznámky k přednášce

10. Prototypy 2: všechno je objekt

verze z 14. prosince 2020

V popisu jazyka uvedeném v tomto souboru není uvedeno primitivum `me`*. V budoucnu ho chci z jazyka vymazat, protože se ukázalo, že není potřeba. Zatím jsem jeho definici ve zdrojovém kódu nechal, ale sám ho nikde nepoužívám.*

1 Objekty

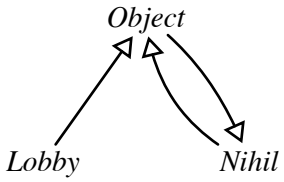
Objekty se skládají z pojmenovaných **slotů**. V každém slotu je opět uložen nějaký objekt. Některé sloty objektů hrají v rámci systému zvláštní roli. Další sloty můžeme objektům přidávat i z nich odstraňovat.

Sloty zapisujeme vždy jako lispový symbol bez počáteční dvojtečky (symboly s dvojtečkou, tzv. *klíče*, mají jiný význam).

Základním slotem je slot `super`, který jako jediný je obsažen v každém objektu. Určuje hierarchii dědičnosti objektů: každý objekt v něm má uloženého svého přímého předka.

Kromě slotů objekt obsahuje i **kód**. Objektům bez kódu říkáme **datové objekty**, objekty s kódem jsou **metody**.

Tři základní objekty a jejich propojení přes slot `super` jsou znázorněny na obrázku:



Objekt *Object* je předkem (prototypem) všech objektů. Je na vrcholu jejich hierarchie (kromě zvláštní role objektu *Nihil*, kterou vysvětlím za chvíli). Obsahuje funkčnost společnou všem objektům.

Objekt *Lobby* lze chápat jako globální prostředí. Ve svých slotech obsahuje hodnoty, které mají být přístupné odkudkoli (z jakéhokoli kódu).

Objekt *Nihil* je naší reprezentací chybějící hodnoty. Jeho paradoxní role v hierarchii objektů je způsobena tím, že náš prototypový jazyk je tak důsledně prototypový, že i „nic“ reprezentuje objektem. Objekt *Object* nemá mít žádného předka. Ve slotu `super` tedy má reprezentaci ničeho, tedy objekt *Nihil*. Ten je ovšem současně objektem a má mít funkčnost, kterou má každý objekt. Proto je přímým potomkem objektu *Object* (má ho ve slotu `super`).

Sloty v objektech se řídí **principem dědičnosti**, podle kterého se hodnoty neexistujících slotů objektu hledají v jeho předcích přes slot `super`. Chceme-li tedy zjistit hodnotu slotu ***x*** objektu *o* a slot v objektu nenajdeme, pokračujeme rekurzivně a pokusíme se slot najít v přímém předkovi objektu. Tak pokračujeme, dokud slot nenajdeme. Neúspěch nastane, když slot nenajdeme ani v objektu *Object*. Pak zjišťování hodnoty slotu skončí chybou.

Nastavování hodnoty slotu se vždy děje přímo v daném objektu. Pokud v něm slot není, nejprve se přidá. Tím dojde k **přepsání** slotu předka.

Dědičnost slotů vede k podobnému efektu jako dědičnost metod u objektově orientovaných jazycích založených na třídách: Při volání metody (zaslání zprávy) se použije metoda předka, pokud metoda třídy neexistuje. Pokud existuje, metoda předka je tím přepsána. Když metodu předka změníme, automaticky se změní pro všechny potomky, kteří nemají vlastní metodu (teď pro jednoduchost pomíjíme volání zděděné metody, ale později se k němu dostaneme).

Sloty v objektech v sobě sdružují několik různých rolí, které se v obvyklých programovacích jazycích řeší zvlášť:

- Sloty slouží k ukládání dat do objektů jako sloty v běžných jazycích.
- Obsahují také metody (které v jazycích založených na třídách patří třídám).
- Pomocí slotu `super` se implementuje dědičnost, a to dědičnost trojího typu:
  - dědičnost datových objektů,
  - metod,
  - lexikálních prostředí.
- Sloty metod hrají roli parametrů a lokálních proměnných.

Základní sloty objektu *Object*

`super`, `set-super`: slot pro přímého předka (obsahuje ho každý objekt) s hodnotou *Nihil*, metoda pro jeho nastavení.

`clone`: obsahuje metodu na klonování objektu (viz dále).

`add`: obsahuje metodu na přidání nového slotu do objektu a případně i nového slotu s metodou na nastavení jeho hodnoty.

`remove`: obsahuje metodu na odstranění slotu.

`name`, `set-name`: název objektu (obvykle textový řetězec) s hodnotou `"OBJECT"`, metoda pro jeho nastavení. Slouží k lepší orientaci v objektech.

Základní sloty objektu *Lobby*

`super:` obsahuje objekt *Object*.

`object:` obsahuje objekt *Object*.

`lobby:` obsahuje objekt *Lobby*.

`nihil:` obsahuje objekt *Nihil*.

`owner`, `self`, `set-owner`, `set-self:` sloty pro použití v metodách (viz dále).

`name:` objekt `"LOBBY"`

Základní sloty objektu *Nihil*

`super:` obsahuje objekt *Object*.

`name:` objekt `"NIHIL"`

Nové objekty lze v programu vytvářet dynamicky klonováním a vkládat přímo do programu jako inline objekty.

Klonování je proces vytváření nového objektu ze zadaného existujícího objektu. Nový objekt obsahuje pouze slot `super`, jehož hodnotou je původní objekt, který je tedy jeho přímým předkem. Ten tak zdědí všechny sloty a jejich hodnoty objektu původního (a jeho předků). Kód má klon totožný s původním objektem. Proces klonování je podobný procesu vytváření nového potomka dané třídy v jazycích založených na třídách.

Inline objekty jsou objekty, jejichž popis je zapsán přímo ve zdrojovém kódu programu. Když interpret při průchodu zdrojovým kódem narazí na popis inline objektu, vytvoří podle něj nový objekt. Popis **standardního inline objektu** v našem jazyce vypadá takto:

```
{(slot1 ... slotm) code1 ... coden}

slot1 ... slotm: symboly (m ≥ 0)
code1 ... coden: výrazy pro zaslání zprávy (n ≥ 0), viz dále
```

Objekt vytvořený podle tohoto popisu bude mít sloty ***`slot1`***, ..., ***`slotm`*** a sloty na nastavení jejich hodnoty `set-`***`slot1`***, ..., `set-`***`slotm`***. Přímým předkem (a hodnotou slotu `super`) nového objektu je aktuální implicitní příjemce (o něm viz dále). Slot `super` nemusí být explicitně v seznamu `(`***`slot1 ... slotm`***`)` uveden A jak uvidíme dále, objektu mohou také automaticky přibýt další sloty.

Podle uvedené syntaxe bychom objekt s $m = n = 0$ (tedy bez nových slotů a bez kódu) zapsali takto:

```
{()}
```

Zápis je možné zkrátit na

```
{}
```

Zvláštním případem inline objektů jsou nestandardní inline objekty, které nazýváme *literály*. Jsou to objekty zapsané přímo ve zdrojovém kódu (tedy nikoli jako popis). Jde o primitivní objekty jako čísla a textové řetězce. Narazí-li interpret na literál, použije přímo jej a žádný nový objekt nevytváří. To má také za následek, že přímý předek literálu je dán konkrétním literálem a nikoli implicitním příjemcem (řetězce jsou například přímými potomky objektu *String*).

Kód programu v našem jazyce sestává pouze z popisů inline objektů a ze zasílání zpráv. O zprávách si povíme teď.

2 Zprávy

S objekty pracujeme tak, že jim posíláme **zprávy**. Zprávou objektu sdělujeme název jeho slotu, se kterým chceme pracovat. Zpráva může také obsahovat další argumenty. Výsledkem zaslání zprávy je nějaký objekt, může mít také vedlejší efekt (typicky nastavení hodnoty slotu).

Při posílání zprávy je třeba vždy znát jejího příjemce. V našem jazyce můžeme příjemce explicitně zadat ve **složeném zaslání zprávy**. Druhá možnost, tzv. **unární zaslání zprávy**, zadání příjemce nevyžaduje a vede k zaslání zprávy tzv. **implicitnímu příjemci**. Unární zaslání zprávy také neumožňuje zadat zprávě argumenty.

Unární zaslání zprávy se zapisuje čistě jako slot, tj. lispovým symbolem:

```
message
message: symbol
```

V unárním zaslání zprávy nejsou uvedeny žádné argumenty, ani objekt, kterému je zpráva zasílána. Tím je pak implicitní příjemce.

Příklad. Základním implicitním příjemcem je *Lobby*. Pokud tomu tak je, tak například kód

```
nihil
```

vrátí objekt *Nihil*, protože slot `nihil` v *Lobby* obsahuje tento objekt. Kód

```
super
```

vrátí objekt *Object*.

Složené zaslání zprávy:

```
[rec-code message code1 :arg2-name code2 ... :argn-name coden]
```

rec-code: kód (tj. zaslání zprávy)
message: zpráva (tj. symbol)
code1 code2 ... coden: kód (tj. zaslání zprávy; $n \geq 0$)
:arg2-name ... :argn-name: lispové klíče, tj. symboly začínající dvojtečkou

Kód *code1* je nepovinný. Pokud je přítomen, je možno uvést i další dvojice *:argi-name codei*. Jejich smysl je zhruba týž jako u nepovinných pojmenovaných parametrů v Lispu: klíč určuje název argumentu, který po něm následuje. Proto nezáleží na jejich pořadí a ne všechny je nutné uvádět.

Když interpret narazí na složené zaslání zprávy, provede nejprve kód *rec-code*, čímž obdrží objekt *receiver*. Pak provede každý kód *code1*, *code2*, ..., *coden* a dostane objekty *arg1*, *arg2*, ..., *argn* zvané *argumenty*. Objekt *receiver* se použije jako příjemce zprávy *message*.

Po přijetí zprávy *receiver* zjistí hodnotu slotu *message* (s případným použitím dědičnosti). Je-li hodnota datovým objektem, vrátí ji jako výsledek (argumenty zaslání zprávy se ignorují). Je-li metodou, spustí metodu s objekty *arg1*, *arg2*, ..., *argn* jako argumenty. Pojmenované argumenty se při volání uloží do stejnojmenných slotů metody, s prvním argumentem se pracuje jinak (podrobnosti dále). Na závěr se vrátí objekt vrácený metodou.

Základní způsob spuštění kódu je zapsat ho přímo do Listeneru. Aby se kód spustil, musíme použít naši syntax složeného zaslání zprávy pomocí `[]`. Uvnitř ale můžeme na vhodných místech používat unární zaslání zprávy pomocí obyčejných lispových symbolů. V Listeneru je defaultním příjemcem zpráv objekt *Lobby*, takže symbol se vyhodnotí na objekt uložený v příslušném slotu objektu *Lobby* (pokud je tam uložen datový objekt, což je v *Lobby* obvyklý případ). Tak hrají unární zprávy v Listeneru roli globálních proměnných, jejichž „vazby“ jsou uloženy v *Lobby*.

Příklad. Tento kód spuštěný v Listeneru

```
[lobby nihil]
```

vrátí objekt *Nihil*. Dojde k tomu takto: první ze symbolů, symbol `lobby` je podle naší syntaxe zaslání unární zprávy. Jelikož v Listeneru je defaultním příjemcem objekt *Lobby*, zprávu `lobby` dostane on. Výsledkem bude objekt *Lobby*, protože tento objekt je v tomto slotu uložen. Druhý symbol, `nihil` je zpráva, bude zaslána

bez argumentů *Lobby*. Výsledkem bude objekt *Nihil* (jelikož *Lobby* ho má ve slotu `nihil` uložen).

Další příklad. Uvažme tento kód v Listeneru:

```
[clone set-name "LOBBY CLONE"]
```

Nejprve se provede kód `clone`. Znamená unární zaslání zprávy defaultnímu příjemci *Lobby*. V *Lobby* slot `clone` není, najde se v objektu *Object*. Hodnotou je metoda, která vytvoří klon příjemce, tedy *Lobby*. Klon dostane zprávu na nastavení jména (slot `set-name` se ovšem opět najde až v objektu *Object*). To povede k vytvoření slotu `name` klonu s hodnotou `"LOBBY CLONE"`.

Ještě jeden příklad. Opět kód v Listeneru:

```
[lobby add "TEST" :value [lobby super]]
```

Nejprve se provede kód `lobby`. Je to opět zaslání unární zprávy objektu *Lobby*. Výsledkem je *Lobby*. Tomu se přidá slot `test`. Hodnota slotu se získá posláním zprávy

```
[lobby super]
```

které vrátí *Object*. Hodnotou slotu se stane objekt *Object*. Je to způsob, kterým byl v *Lobby* vytvořen a nastaven slot `super`.

3 Metody

Metody se dělí na **standardní metody** a **primitiva**. *Primitivum* je funkce napsaná v Lispu a v rámci našeho jazyka není specifikováno, jak funguje. *Standardní metoda* je standardní objekt zapsaný `{}`-syntaxí, obsahuje sloty a kód. Je-li metoda spuštěná s argumenty, dostanou se do příslušných slotů. Ty pak v kódu metody fungují jako parametry.

Jak bylo uvedeno dříve, metoda se volá s nepovinnými argumenty *`arg1`*, *`arg2`*, ..., *`argn`*. První je dán pozičně, ostatní podle názvu daného lispovým klíčem. Nyní popíšeme, co se děje při spouštění metody. Připomeňme jen, že metodu lze spustit jen jedním způsobem: posláním zprávy objektu, který metodu obsahuje ve slotu.

Spouštění primitiva

Jak bylo řečeno, primitivum je lispová funkce. Při jejím spuštění se funkce prostě aplikuje na seznam `(`*`receiver arg1 :arg2-name arg2 ... :argn-name argn`*`)`. Jako výsledek musí vrátit objekt.

Takto je například v našem systému definováno primitivum pro tisk (pro jednoduchost tady neuvádím deklaraci na potlačení warningu):

```
[object add "PRINT" :value (lambda (self arg1 &key)
                             (print self))]
```

Spouštění standardní metody

1. Metoda se nejprve **naklonuje**. Tím se zajistí, že nastavování hodnot slotů klonu neovlivní původní metodu. Klon bude mít také stejné předky jako původní metoda, což je důležité pro další proces.
2. Pokud byl příjemcem zprávy datový objekt, pak se klonu přidá slot `self` a nastaví se mu správná hodnota (podrobněji dále).
3. Klonu se dále nastaví sloty podle argumentů zaslání zprávy: najde se nebo vytvoří slot pro první argument a nastaví se mu hodnota prvního argumentu (podrobnosti dále). Pak se najdou sloty k pojmenovaným argumentům volání a nastaví jejich hodnoty na argumenty.
4. Klon se nastaví jako implicitní příjemce.
5. Provede se kód metody.

Popis speciálních slotů v metodě:

Slot pro první argument

Pokud je metoda volána s alespoň jedním argumentem, uloží se argument do zvláštního slotu pro první argument. Tento slot se pozná podle toho, že jeho název začíná na `"ARG1"`. Obvykle se používá přímo tento název, ale někdy potřebujeme rozlišit mezi sloty pro první argument více metod, pak používáme různé názvy. Pokud slot v metodě neexistuje, přidá se jí automaticky s názvem `"ARG1"`. U dalších slotů metody tento problém není, protože další argumenty volání mají název explicitně uveden.

Slot `super`

Jak bylo řečeno, v každý moment vykonávání kódu je určen **implicitní příjemce zpráv**. Při vytváření nové metody zadané `{}`-syntaxí interpret nastaví její slot `super` na aktuálního příjemce. Hodnota slotu `super` klonu je samozřejmě nastavena na původní metodu. Tak se pomocí slotů `super` realizuje **dědičnost lexikálních prostředí**.

Slot `self`

Je-li spuštění metody výsledkem poslání zprávy datovému objektu, je tento objekt uložen ve slotu `self` metody. Pozor, hodnota `self` se může lišit od objektu, který metodu obsahuje. Když datový objekt dostane zprávu, nemusí být slot téhož

názvu jako zpráva nalezen přímo v něm, ale může se uplatnit dědičnost a slot s metodou být nalezen až v některém předkovi. Hodnota slotu `self` metodu bude ovšem nastavena na příjemce zprávy, nikoli majitele metody.

Slot `owner`

Obsahuje majitele metody, je-li jím datový objekt. Hodnota slotu `owner` se nastaví (předtím se slot případně vytvoří) v momentě, kdy je metoda ukládána do slotu datového objektu.

Příklad. V tomto příkladu si podrobně vysvětlíme, jak funguje posílání zpráv a spouštění metod.

Máme datový objekt uložený ve slotu `obj` objektu *Lobby*. *Lobby* je také aktuálním implicitním příjemcem. Dále předpokládejme, že máme ve slotu `print` objektu *Object* primitivum na tisk a ve slotu `name` našeho objektu řetězec `"OBJ"`. Spustíme tento kód:

```
[obj add "X" :value {(arg1 b) [[self name] print]
                               [arg1 print]
                               [b print]
                               [name print]}]
[obj x "TEST1" :b "TEST2"]
```

Na prvním řádku se nejprve najde náš objekt ve slotu `obj` *Lobby* (jelikož *Lobby* je implicitním příjemcem). Pak se mu pošle zpráva `add` s argumenty `"X"` a novou metodou. (Náš objekt zprávě `add` rozumí, protože je potomkem objektu *Object*.) To povede k vytvoření slotu `x` v našem objektu s novou metodou jako hodnotou.

Nová metoda bude mít kromě slotů `arg1` a `b` ještě sloty `super` (každý objekt ho má) a `owner`. Hodnota `super` je nastavena na implicitního příjemce, který byl aktuální v momentě vytváření metody, tedy na objekt *Lobby*. Slot `owner` se metodě nastaví v momentě, kdy je ukládána do slotu našeho objektu — jelikož náš objekt je datový.

Na druhém řádku se našemu objektu pošle zpráva `x` s prvním argumentem `"TEST1"` a argumentem s názvem `:b` rovným `"TEST2"`. Slot `x` objektu obsahuje metodu, takže se provedou následující kroky:

1. Metoda se naklonuje. Vznikne tedy metoda s jediným slotem `super` obsahujícím původní metodu a s totožným kódem. Předkem klonu je původní metoda, jejímž předkem je *Lobby*.
2. Zavolání metody je důsledkem poslání zprávy `x` našemu objektu, který je **datový**. Proto se klonu přidá slot `self` a nastaví se na náš objekt.
3. Náš klon je potomkem metody, která obsahuje sloty `arg1`, `b`, `set-arg1` a `set-b`. Podle názvu prvního slotu se zjistí, že jde o slot pro první argument. Zasláním zpráv `set-arg1` a `set-b` se nastaví hodnoty slotů `arg1` a `b` na `"TEST1"` a `TEST2"`.

4. Klon se nastaví na implicitního příjemce.

5. Vykoná se kód:

   5.1 Pošle se zpráva `self`. Implicitním příjemcem je teď náš klon, jehož slot `self` obsahuje náš objekt. Následovné poslání zpráv `name` a `print` tedy vytiskne řetězec `"OBJ"` (slot `name` se najde přímo v našem objektu, slot `print` v jeho předku *Object*).

   5.2 Další dva výrazy posílají zprávu `arg1` a `b`. Obě dojdou implicitnímu příjemce (tedy našemu klonu), ve kterém se příslušné sloty najdou. Jejich hodnotám se pošle zpráva `print`, takže se vytisknou řetězce `"TEST1"` a `TEST2"`.

   5.3 Pošle se zpráva `name`. Náš klon ani jeho předek slot `name` neobsahuje, slot se najde teprve v *Lobby*, takže se vytiskne `"LOBBY"` (*Lobby* je potomkem objektu *Object*, takže zprávě `print` rozumí).

4 Použití základních zpráv

Podrobněji k základním zprávám:

`super` obsahuje přímého předka objektu, v objektu *Object* má i setter (slot `set-super` s metodou pro jeho nastavení).

`add` Používá se k přidání nového slotu k objektu, který je příjemcem zprávy. Jako první argument se zadává název slotu jako řetězec, pojmenovaný argument `value` udá počáteční hodnotu slotu. Syntax poslání zprávy tedy je

```
[obj add slot-name :value val]
```

Pokud je jeho počáteční hodnota datový objekt, přidá se i slot se setterem, což je slot `set-`*`slot-name`* obsahující primitivum na nastavení hodnoty nového slotu. Primitivum akceptuje jen jeden argument, a to novou hodnotu. (Primitivum)

`remove` Odstraní slot, jehož název je jako řetězec uveden v prvním argumentu. (Primitivum)

`clone` Posílá se bez argumentu, vrátí nový klon příjemce zprávy. Metoda je napsána v našem jazyce takto:

```
[object add "CLONE" :value {() [{} set-super self]}]
```

`name` Jméno objektu (většinou textový řetězec), má i setter. Je naprogramován v našem jazyce:

```
    [object add "NAME" :value "OBJECT"]
    [nihil set-name "NIHIL"]
    [lobby set-name "LOBBY"]
```

5 Nestandardní objekty

Textové řetězce a čísla jsou implementovány jinak než standardní objekty: jsou reprezentovány přímo lispovými hodnotami a ve zdrojovém kódu fungují jako literály. Jak řetězce, tak čísla jsou ovšem objekty ve smyslu našeho jazyka.

Textové řetězce mají jako přímého předka nastavený jistý standardní objekt *String*, který je uložený v *Lobby* ve slotu `string`. Obecným řetězcům je tedy možné programovat metody. V systému je jako primitivum definována zpráva `+` s jedním argumentem na spojování řetězců:

```
> ["Dobrý " + "den"]
"Dobrý den"
```

Kdybychom chtěli, aby řetězce správně reagovaly na zprávu `name`, můžeme napsat toto:

```
[string set-name {() self}]
```

(všimněte si, že jsme přepsali datový objekt metodou)

Pak bude fungovat

```
> ["Ahoj" name]
"Ahoj"
```

Bohužel toto řešení má nevýhodu, že samotný objekt *String* nebude mít hezky nastavené jméno. S tím, co zatím víme, to nejde napravit, ale později to půjde.

Konkrétním textovým řetězcům nelze přidávat žádné sloty. Rozumějí pouze zprávám, které jsou implementovány v jejich společných předcích tedy v objektu *String* a jeho předcích.

Čísla v zásadě fungují stejným způsobem jako textové řetězce. Jsou reprezentována lispovými čísly a mají společného předka *Number*, což je standardní objekt uložený ve slotu `number` *Lobby*. V něm jsou uloženy metody na práci s obecnými čísly, například primitivum na sčítání:

```
[number add "+" :value (lambda (self arg1 &key)
                         (+ self arg1))]
```

a podobně na další aritmetické operace (za chviličku využijeme zprávy na výpočet rozdílu a součinu).

Na rozdíl od řetězců ale lze konkrétním číslům přidávat i nové sloty (je to uděláno tak, že plisty se sloty a jejich hodnotami jsou uloženy v globální proměnné). Proto můžeme elegantně naprogramovat například zprávu na výpočet faktoriálu čísla, aniž bychom používali větvení programu (této techniky si všimněte, je poměrně obvyklá):

```
[number add "!" :value {() [self * [[self - 1] !]]}]
[0 add "!" :value 1]
```

(Při nastavování hodnoty slotu v nule jsme museli použít zprávu `add`, protože setter `set-!` neexistuje, jelikož jsme v objektu `number` nastavili slot `!` na metodu. To je jeden z drobných nedostatků našeho jazyka, které by nebylo špatné odstranit.)

6 Příklady a užitečné techniky

Příklad (logika).

Jak už jsme viděli v příkladu na faktoriál, větvení programu lze někdy provést deklarativně, aniž bychom explicitně použili test nějaké podmínky. Je to technika, kterou je **vhodné v našem jazyce používat**.

Přesto samozřejmě pravdivostní hodnoty a také explicitní větvení programu někdy potřebujeme. Zajímavé je, že v našem jazyce je možné definovat větvení jako zprávu. Nemusí to být něco jako *speciální operátor*, jak to známe z jiných jazyků včetně Lispu (naposledy jsme se s touto zajímavostí setkali u líného vyhodnocování).

V našem systému definujeme objekty *True* a *False*, které zastupují pravdivostní hodnoty. Každý obsahuje slot `if-true` se speciálně definovanou metodou:

```
[true add "IF-TRUE" :value {(arg1 else) arg1}]
[false add "IF-TRUE" :value {(arg1 else) else}]
```

Jako test je uveden tento program:

```
[true if-true {() [1 print] 10} :else {() [2 print] 11}]
```

Test funguje podle očekávání: vytiskne `1` a vrátí `10` a nic jiného neudělá (zejména tedy nevytiskne `2`).

Je to z těchto důvodů: samotné odeslání zprávy nic nevytiskne, protože argumenty jsou metody. Ty se jen vytvoří, ale nespustí. V kódu metody objektu `true` jsou metody uloženy ve slotech `arg1` a `else`. Kód metody posílá zprávu `arg1`, takže dojde ke spuštění metody uložené ve slotu `arg1`. S metodou ve slotu `else` se nic nestane.

Příklad (volání zděděné metody).

Chceme definovat zkušební objekt, jehož jméno bude vždy obsahovat jméno předka. Pokud ho dáme do lobby do slotu `test`, měl by se objekt i jeho potomci chovat takto:

```
> [test set-super object]
...

> [test name]
"CHILD OF OBJECT"

> [test set-super lobby]
...

> [test name]
"CHILD OF LOBBY"

> [test set-super [lobby clone]]
...

> [test name]
"CHILD OF LOBBY"

> [[test clone] name]
"CHILD OF LOBBY"
```

Jedna možnost řešení: testovacímu objektu definovat novou metodu ve slotu `name` (protože jméno se má zjišťovat dynamicky). Metoda by měla **zavolat zděděnou metodu** a k výsledku zepředu připojit řetězec `"CHILD OF "`.

Zděděnou metodu zavoláme tak, že zjistíme majitele aktuální metody a jeho předkovi přepošleme tutéž zprávu:

```
[test set-name {() ["CHILD OF " + [[owner super] name]]}]
```

Můžete vyzkoušet, že s touto definicí se objekt chová podle zadání.

Volání zděděné metody je samozřejmě základní nástroj objektově orientovaného programování. Tady jsme si ukázali, jak je lze realizovat prostředky našeho jazyka bez použití nového klíčového slova (jak tomu bývá v jiných jazycích).

Náš jazyk nám ovšem umožňuje dělat i jiné věci, které už nejsou tak běžné. Změňme definici takto:

```
[test set-name {() ["CHILD OF " + [[self super] name]]}]
```

Zkuste si zopakovat uvedené testy. Jaké budete dostávat výsledky?

Příklad (zpráva jako argument).

Chceme napsat metodu *Lobby*, která pošle danou unární zprávu více objektům současně. Objekty by byly třeba tři, každý zadaný v jednom argumentu (obecné seznamy máte jako úkol). Chceme, aby zpráva mohla být jakákoli, tedy aby se také zadávala jako argument.

Přesně takto to samozřejmě nejde, ale zprávu můžeme zapouzdřit do metody. Například požadavek vytisknout tři objekty by pak mohl vypadat takto:

```
[lobby send-to-3 {(arg1) [arg1 print]} :obj1 1 :obj2 2 :obj3 3]
```

kde objekty k vytištění jsou čísla `1`, `2`, `3`.

Než se podíváme, jak metodu vytvořit, uvědomte si ještě jednu zajímavost. Místo předchozího programu jde také napsat

```
[{} send-to-3 {(arg1) [arg1 print]} :obj1 1 :obj2 2 :obj3 3]
```

Složené závorky `{}` jsou inline objekt, jehož předkem je aktuální příjemce. Pokud ten nebude obsahovat slot `send-to-3` (a to bychom viděli ve zdrojovém kódu, protože jde o lexikální prostředí) a nebudou ho obsahovat ani lexikálně nadřazené metody, skončí hledání v *Lobby*.

Metodu napíšeme takto:

```
[lobby add "SEND-TO-3" :value {(arg1 obj1 obj2 obj3)
                               [{} arg1 obj1]
                               [{} arg1 obj2]
                               [{} arg1 obj3]}]
```

Opět jsme použili trik s prázdným objektem `{}`, kterému jsme poslali zprávu `arg1`. Prázdný objekt slot `arg1` nemá, takže hledání pokračuje v předkovi, což je lexikálně nadřazená metoda. Ta je totiž v daný okamžik implicitním příjemcem, takže předkem prázdného objektu.

V metodě se slot `arg1` nachází. Jeho hodnotou je ovšem metoda, takže ta se spustí; poprvé s argumentem `obj1`, pak s `obj2` a nakonec s `obj3`. (To jsou ovšem opět zprávy, tentokrát unární, takže se pošlou implicitnímu příjemci, který příslušné sloty obsahuje.)

Příklad (spuštění inline metody).

Podobný problém řeší následující zpráva `force`, která se později ukáže užitečnou. Umožňuje spustit metodu, která je dána v jejím argumentu jako inline objekt. Například výraz

```
[{} force {() ["Test" print]}]
```

vytiskne řetězec `"Test"`. Metoda pro zprávu `force` je velmi prostá. využívá toho, že metodu spustíme, když se ji pokusíme „vytáhnout ze slotu“:

```
[lobby add "FORCE" :value {(arg1) arg1}]
```

Příklad (vylepšení jména řetězců).

Vylepšíme metodu `name` objektu *String* tak, aby fungovala dobře nejen na konkrétní řetězce, ale i na samotný *String*. Pokud jste nad problémem přemýšleli, zjistili jste, že je potřeba podívat se, jestli je příjemce zprávy `name` přímo roven objektu *String*. Příklad také demonstruje použití zprávy `if-true`. K otestování rovnosti použijeme primitivum `eql`, které máme v systému k dispozici (viz soubor se základními definicemi).

Definice:

```
[string set-name
        {() [[self eql string] if-true "STRING" :else self]}]
```

Poznámka. Test rovnosti dvou objektů patří k základní funkčnosti. Zajímavé je, že zvláštním trikem je možné jej naprogramovat přímo v našem jazyce, není nutné jej dělat jako primitivum. Řešení je ukázáno v příkladech (zpráva `equals`). Je sice poněkud krkolomné, ale ukazuje, co všechno se s malým množstvím nástrojů dá dělat (tohle nejde ani v líném vyhodnocování, jak jsme si ho ukazovali v minulém semestru).

Příklad (esoterická čísla).

Ve zdrojovém kódu máte příklad na reprezentaci přirozených čísel pomocí standardních objektů. Příklad má za cíl ukázat modelovací možnosti jazyka (jinak je samozřejmě vhodnější používat jako čísla externí objekty) a neobvyklé techniky dědičnosti (např. číslo 1 je potomkem čísla 0). Doporučuji se na příklad podívat.

Příklad (lokální proměnné).

Pokud bychom chtěli změnit hodnotu parametru metody, dojdeme k závěru, že náš programovací jazyk to neumožňuje. Představme si například, že chceme napsat metodu, která po řadě provede dvě metody a vrátí výsledek první.

Metodu napíšeme tak, že jí dáme navíc slot `var`, ve kterém si chceme zapamatovat první vypočítanou hodnotu:

```
[lobby add "RETURN-FIRST"
           :value {(arg1 arg2 var) [?? set-var arg1]
                                   arg2
                                   var}]
```

Narazíme ovšem na problém, že neumíme poslat implicitnímu příjemci zprávu s argumentem. Pokud bychom místo otazníků napsali obvyklé `{}`, nastaví se hodnota slotu `var` prázdnému objektu zadaného jako `{}`, nikoli celé metodě (protože hodnota slotu se nastavuje vždy příjemci zprávy, nepoužívá se dědičnost).

Řešení používá zprávu `force`:

```
[lobby add "RETURN-FIRST"
           :value {(arg1 arg2)
                   [{} force [{(var) arg2 var} set-var arg1]]}]
```

Vycházíme z toho, že inline objekt `{(var) arg2 var}` má slot `var`, který lze nastavit zprávou `set-var`. Výsledkem je objekt s novou hodnotou slotu, který se pak spustí (jakožto metoda) zprávou `force`.

Otázky a úkoly na cvičení

Pokud není uvedeno jinak, řešte úkoly čistě prostředky našeho prototypového jazyka. Neupravujte interpret ani nedefinujte nová primitiva nebo externí objekty.

1. Doplňte do jazyka logickou disjunkci (zpráva `or`) a negaci (zpráva `not`).
2. Doplňte do jazyka zprávu `is-nihil`, která vrátí `true` pro `nihil`, jinak `false`. Nepoužijte ani zprávu `equals`, ani `is`, ani `if-true`. Udělejte to jednodušeji.
3. Doplňte do jazyka seznamy tak, že vytvoříte prototyp tečkového páru a prázdný seznam. Tečkový pár by měl rozumět zprávám `car` a `cdr`. Je potřeba vytvořit prototyp pro obecný seznam? (Tj. jak prázdný, tak neprázdný?) Zdůvodněte.
4. Napište zprávu `is-list`, která vrátí `true`, pokud je příjemce seznam, a jinak vrátí `false`.
5. Definujte zprávu `length` pro zjišťování délky seznamu vyjádřenou esoterickým číslem. Např. výraz `[list length]` vrátí délku seznamu `list`.
6. Definujte další operace pro seznamy, např. spojování, prohledávání, mazání prvku, převrácení.
7. Jak implementovat zprávu `broadcast` na poslání dané unární zprávy všem prvkům seznamu? Například toto volání

```
[list broadcast {(arg1) [arg1 print]}]
```

by mělo vytisknout všechny prvky seznamu `list`. Jak implementovat mapování přes seznam?

8. Seznamy vytvořené v předchozích příkladech jsou automaticky líné (neboli jsou to *proudy*, jak je známe z minulého semestru). Přesvědčte se o tom tak, že vytvoříte například seznam všech sudých čísel.

9. Definujte násobení a podle výběru i další operace s esoterickými čísly.

10. Přidejte primitivum `error` na vyvolání chyby. Umístili byste je do objektu *Object*, nebo do *Lobby*?

11. Napište prototypovou verzi základů knihovny `omg` z předchozích přednášek.