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 2 ◇ poznámky k přednášce

10. Zásobníkové výpočty

verze z 23. dubna 2024

Na dalších přednáškách si vyzkoušíme zvláštní paradigma programování, tzv. **zásobníkové programování (zásobníkové výpočty)**.

Zásobníkové programování je široce používané jak v teorii, tak při nízkoúrovňovém programování. Na přednášce vytvoříme jednoduchý zásobníkový programovací jazyk, který vychází ze základního zásobníkového jazyka *Forth*.

Hlavním přínosem přednášky (kromě seznámení s principem zásobníkových strojů, programovacích jazyků a zásobníkových výpočtů obecně) by mělo být, že si uvědomíme, že každý běžící program (proces) používá svůj zásobník a že na rozdíl od programátorů v obvyklých programovacích jazycích (včetně Lispu) přímo uvidíme, jak takový zásobník pracuje.

Následuje jedna přípravná kapitola. Po ní se pustíme do zásobníkových výpočtů.

1 Viditelnost a životnost, dynamické proměnné

Proměnné a vazby, které v Lispu používáme, mají tu vlastnost, že jejich přístupnost je určena nějakou oblastí ve zdrojovém kódu. Říká se také, že je určena *lexikálně*.

Například vazba symbolu vytvořená operátorem `let` je platná těle operátoru, vazba parametru funkce na argument je platná v těle funkce. Jen výrazy uvedené v těle (operátoru `let` nebo funkce) mohou vazbu použít.

U proměnných v Lispu platí, že kromě omezení daného lexikálně, o kterém jsme mluvili teď, už nijak jinak omezeny nejsou. Konkrétněji, nejsou omezeny *časově*. Jednou vytvořená vazba proměnné na hodnotu z pohledu programátora nikdy nezaniká (proto je v Lispu možné používat lexikální uzávěry).

Přístupnost vazeb symbolů je tedy dána dvěma podmínkami: za jakých okolností jsou vidět a po jaký čas od svého vzniku existují.

Těmto dvěma rysům vazeb symbolů (a dalších věcí) říkáme **viditelnost** (*scope*) a **životnost** (*extent*). Základní typy viditelnosti jsou dva:

Lexikální viditelnost: vazba symbolu je vidět pouze v určité oblasti zdrojového kódu. V případě parametrů funkcí je to všude v těle funkce, v případě vazeb vytvořených operátorem `let` všude v jeho těle. V obou případech pokud není vazba překryta novou vazbou téhož symbolu.

Neomezenou viditelnost mají vazby, které jsou použitelné kdekoli (na libovolném místě) zdrojového kódu (opět pokud nejsou překryty jinou vazbou).

Základní typy životnosti jsou také dva.

Omezenou životnost má vazba, jestliže existuje moment v čase, kdy vazba zanikne.

Neomezená životnost vazby způsobuje, že vazba po svém vzniku (z pohledu programátora) nikdy nezanikne.

Typy proměnných a vazeb jsou dány tím, jakou mají viditelnost a životnost.

Lexikální vazby. Mají lexikální viditelnost a neomezenou životnost. Jsou to vazby, které známe z Lispu.

Dynamické vazby. Mají neomezenou viditelnost a omezenou životnost. Je to starší typ, ukážeme je za chvíli.

Dále například **lokální proměnné v C** mají lexikální viditelnost a omezenou životnost: lokální proměnnou funkce nelze použít mimo tělo funkce a proměnná po návratu z funkce přestává existovat, což lze ověřit tím, že si zapamatujeme adresu proměnné a později se podíváme na hodnotu na adrese uloženou.

Globální proměnné mívají neomezenou viditelnost a neomezenou životnost, případně lexikální viditelnost v rámci zdrojového souboru a neomezenou životnost.

V Lispu mohou mít symboly jak lexikální, tak dynamické vazby. Za normálních okolností jsou všechny vazby lexikální a to je také typ vazby, který zatím známe. S dynamickými vazbami se můžeme setkat v těchto situacích:

1. Symbol lze na globální úrovni definovat jako **dynamickou** (někdy též **speciální**) proměnnou. Pak jsou všechny jeho nové vazby dynamické.
2. Novou vazbu symbolu můžeme učinit dynamickou pomocí vhodné **deklarace**.

Vysvětlíme obě tyto možnosti.

Dynamická proměnná je proměnná, jejíž vazby jsou vždy dynamické: každá nově vytvořená vazba dynamické proměnné (např. pomocí `let` nebo navázáním argumentu funkce na parametr) je dynamická.

Dynamická je například proměnná `*print-length*`, která určuje délku tištěného seznamu a kterou už známe. Pokud má proměnná hodnotu `nil`, tiskne se každý seznam celý. Pokud je její hodnotou číslo, určuje maximální počet vytištěných prvků seznamu (kvůli úspoře místa, ale za cenu ztráty informace):

```
CL-USER 1 > *print-length*
NIL

CL-USER 2 > (setf list (list 1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
```

```
CL-USER 3 > (setf *print-length* 8)
8

CL-USER 4 > list
(1 2 3 4 5 6 7 8 ...)

CL-USER 5 > (let ((*print-length* 5))
              (print list))

(1 2 3 4 5 ...)
(1 2 3 4 5 6 7 8 ...)
```

Poslední vyhodnocení používá techniku, která by nešla použít, kdyby proměnná `*print-length*` byla lexikální. Je to z toho důvodu, že její vazba na hodnotu `5` by pak byla vidět jen v těle `let`-výrazu, nikoliv v těle funkce `print` (které pochopitelně někde existuje, i když nevíme, jak vypadá).

Jen připomeňme, že druhý vypsaný seznam je `(1 2 3 4 5 6 7 8 ...)`, protože jde o vytisknutou návratovou hodnotu funkce `print`. Ta se tiskla v době, kdy vnitřní vazba symbolu `*print-length*` už neexistovala.

Dynamické proměnné lze chápat jako globální proměnné z jiných jazyků, s tím dodatkem, že jim lze tvořit nové vazby (které jsou vždy dynamické).

V Lispu jsou i další dynamické proměnné, které ovlivňují tisk. Jednou z nich je proměnná `*print-base*`, která určuje soustavu, ve které se tisknou čísla:

```
CL-USER 10 > (setf *print-base* 8)
10

CL-USER 11 > (let ((*print-base* 16))
               (print 140))

8C
214
```

Všimněte si, jak se v Listeneru jednotlivé hodnoty vytiskly. Na prvním řádku se vytisklo `10`, protože je to číslo 8 v osmičkové soustavě a v momentě tisku je už proměnná `*print-base*` nastavena na `8`. Na druhém řádku se vytisklo `8C` a `214`. První text vytiskla funkce `print` v našem kódu. V momentě, kdy tiskla, byla proměnná `*print-base*` nastavena na `16` (je to dáno **viditelností** vazby). `8C` je skutečně zápis čísla `140` v šestnáctkové soustavě. Funkce `print` pak vrátila hodnotu `140`. Ta se vytiskla v osmičkové soustavě, protože v ten moment už vazba proměnné `*print-base*` na číslo `16` neexistovala (má omezenou **životnost**).

V Lispu můžeme definovat vlastní dynamické proměnné na globální úrovni makrem `defvar`:

`(defvar` *var*`)`

var: symbol

Makro označí proměnnou *var* jako speciální. Všechny nové vazby proměnné nyní budou dynamické. Po použití tohoto makra obvykle novou dynamickou proměnnou nastavíme na nějakou hodnotu makrem `setf`. (Jde to i pomocí nepovinného argumentu přímo v makru `defvar`; tuto možnost ale nepoužíváme.)

Dynamické proměnné je třeba vždy jasně odlišit, aby je nebylo možné zaměnit za proměnné lexikální. V Lispu je proto zavedena následující konvence.

Název dynamické proměnné.

Všechny proměnné definované makrem `defvar` mají název ohraničený hvězdičkami.

Pomocí tzv. *deklarace* lze stanovit pro jednotlivou vazbu symbolu, že má jít o dynamickou vazbu. Tuto možnost nebudeme používat.

Jelikož dynamická vazba přestává v čase existovat, není ukládána do lexikálních uzávěrů. Kdybychom například chtěli vyzkoušet funkci `adder` s dynamickou vazbou parametru `x`, udělali bychom to takto:

```
(defvar *x*)

(defun adder (*x*)
  (lambda (y)
    (+ y *x*)))
```

Test:

```
CL-USER 1 > (funcall (adder 5) 2)

Error: The variable *X* is unbound.
...

CL-USER 2 : 1 > :a

CL-USER 3 > (setf *x* 98)
98

CL-USER 4 > (funcall (adder 5) 2)
100
```

```
CL-USER 5 > (adder 5)
#<Function 1 subfunction of ADDER 82300197C9>

CL-USER 6 > (funcall * 2)
100
```

Dynamické proměnné a dynamické vazby jsou potenciálně nebezpečné. Proto

Používání dynamických proměnných.

Pokud to není zcela nutné, dynamické proměnné nepoužíváme. Pokud musíme, tak vždy s dodržením uvedené dohody o jejich názvech.

Uvedené pravidlo je variantou pravidla známého ze všech programovacích jazyků a týkajícího se nevhodnosti používání globálních proměnných.

V Common Lispu lze dynamické proměnné používat zřejmě hlavně z historických důvodů. U moderních jazyků s kratší historií se s nimi nesetkáváme.

Ještě stručně poznamenejme, že **funkční vazby symbolů** u lokálních funkcí **jsou vždy lexikální**.

2 Zásobníkový stroj

Jak víte, zásobník je základní datová struktura, se kterou pracujeme pomocí operací přidávání prvku na vrchol a odebírání prvku z něj (struktura typu LIFO). Jiné operace se zásobníkem nejsou k dispozici, dále můžeme pouze procházet a číst jeho obsah. Výpočty založené na práci se zásobníky lze chápat jako programovací techniku, která k řešení daného problému používá datovou strukturu zásobníku.

Příkladem práce se zásobníkem může být každý program (proces), který běží v počítači. Zásobník v něm slouží k ukládání lokálních proměnných funkcí, díky němu se také mohou funkce vzájemně i rekurzivně volat.

Zásobník je nejjednodušší reprezentovat seznamem uloženým v proměnné nebo na jiném místě. Pro zopakování:

```
CL-USER 53 > (setf stack '(1 2))
(1 2)

CL-USER 54 > (push 0 stack)
(0 1 2)

CL-USER 55 > (pop stack)
0
```

```
CL-USER 56 > (pop stack)
1

CL-USER 57 > (pop stack)
2
```

(S poslední variantou jsme se myslím nesetkali: u prázdného zásobníku makro `pop` nevyvolá chybu, ale vrátí `nil`.)

Důsledným používáním zásobníků dostaneme programovací paradigma, které řeší všechno pomocí jednoho nebo více zásobníků. Takový **zásobníkový model výpočtu** je založen na představě virtuálního stroje, jehož výpočetní proces spočívá především v manipulaci se zásobníky. Na této přednášce napíšeme simulátor takového stroje.

Náš **zásobníkový stroj** v nejjednodušší podobě bude obsahovat dva zásobníky:

rslt (neboli *Result Stack*, *datový zásobník*) obsahuje **data**, a to jak data vstupní, tak mezivýsledky výpočtů, se kterými se dále pracuje, a konečný výsledek (výsledky) celého programu. Je obdobou hlavního zásobníku, který používá každý běžící program (proces) v počítači.

exec (*Execution Stack*, *programový zásobník*) obsahuje zdrojový kód programu, který má stroj vykonávat. Tento zásobník se v praxi často nevyskytuje, zdrojový kód může být prostě uložen místo na zásobníku někde v paměti, odkud jej stroj čte. My máme k použití zásobníku na zdrojový kód své důvody, které postupně pochopíme.

Prvky na vrcholu zásobníku ***exec*** budou dvojího typu:

Slovo — lispový klíč (tj. symbol, jehož zápis začíná dvojtečkou). Hraje roli „klíčového slova“.

Hodnota — cokoli jiného (na této přednášce jakékoli číslo nebo textový řetězec, později i další hodnoty).

Na začátku vykonávání programu je datový zásobník prázdný a na programovém je uložen celý program. Stroj pracuje iterativně v jednotlivých krocích. V každém kroku odebere prvek z programového zásobníku a podle toho, jaký je, vykonává akci se zásobníky.

Je-li odebraný prvek hodnota, uloží ho prostě na vrchol datového zásobníku. Je-li slovo, znamená instrukci pro stroj, co má dál dělat. Stroj provede akce s oběma zásobníky podle toho, o jaké slovo jde.

Takto stroj postupuje, dokud je programový zásobník neprázdný. V momentě, kdy se zásobník vyprázdní, program skončil. Na datovém zásobníku by měla zůstat jedna hodnota, která je výsledkem výpočtu.

Abyste práci stroje, která je tady jen teoreticky popsána, pochopili, dívejte se na animace na slajdech k přednášce. Práci stroje si také sami vyzkoušíte na emulátoru, který máte k dispozici a který popíšu na této přednášce nakonec.

Příklad vykonání jednoduchého programu.
Program `5 3 :- 4 :*` se umístí na programový zásobník tak, že začátek (číslo `5`) je na vrcholu. Kroky programu budou (viz také animaci na slajdech):

1. číslo `5` se přesune na datový zásobník
2. číslo `3` se přesune na datový zásobník
3. symbol `:-` je slovo; toto slovo vypustí z datového zásobníku obě čísla a na zásobník uloží jejich rozdíl, tedy číslo `2`
4. číslo `4` se přesune na datový zásobník
5. symbol `:*` je slovo; toto slovo vypustí z datového zásobníku obě čísla a na zásobník uloží jejich součin, tedy číslo `8`

Výsledkem programu je tedy `8`.

Vidíme, že zápis programu používá postfixovou notaci: nejprve je vždy třeba uvést operandy a pak až operaci. Není to ale žádný vrtoch, tento druh zápisu vyplynul z používání datového zásobníku. Je možné si také všimnout, že postfixová notace je důsledně imperativní a uvádí akce přesně v tom pořadí, ve kterém výpočet opravdu probíhá: například součin $(5-3)*4$ počítáme tak, že nejprve vezmeme čísla 5 a 3, pak je odečteme, pak vezmeme číslo 4 a s předchozím výsledkem je vynásobíme (předchozí výsledek si musíme někde pamatovat ... na zásobníku). Postfixovému zápisu se někdy říká **obrácená polská notace** (logika).

Základní slova

V popisu slov používáme zápis, který nejjednodušeji vysvětlím na příkladě už použitého slova `:-`. Slovo vezme první dvě hodnoty z datového zásobníku, řekněme n_1 a n_2. Hodnoty bereme v pořadí, v jakém jsou na zásobník ukládány; n_2 je tedy na vrcholu. Pak napíšeme pomlčku a za ni nové prvky, které se na zásobník uloží, a to opět ve stejném pořadí. Celý popis uzavřeme závorkami. Popis slova `:-` může tedy být tento: (n_1 n_2 – rozdíl $n_1 - n_2$).

Následuje tedy popis základních slov.

Manipulace se zásobníkem

`:swap`	$(a\ b - b\ a)$
`:rot`	$(a\ b\ c - b\ c\ a)$
`:drop`	$(a\ -\ )$
`:dup`	$(a - a\ a)$
`:over`	$(a\ b - a\ b\ a)$

Aritmetické operace

`:+`	$(n_1\ n_2$ – součet $n_1 + n_2)$
`:-`	$(n_1\ n_2$ – rozdíl $n_1 - n_2)$
`:*`	$(n_1\ n_2$ – součin $n_1 \cdot n_2)$
`:/`	$(n_1\ n_2$ – podíl $n_1/n_2)$

Slova mohou být **vestavěná** nebo **uživatelská**. Vestavěná slova jsou už dána jazykem, uživatelská jsou slova, která si uživatel definuje sám. Vestavěná slova jsou v našem simulátoru reprezentována lispovou funkcí, uživatelská kódem jazyka.

Slova a jejich význam ukládáme na zvláštní zásobník jménem ***word***. V našem simulátoru jsou jeho prvky páry `(`*`slovo`*` . `*`význam`*`)`. Takto utvořeným zásobníkům říkáme *slovníky*. *Významem* slova je jeho reprezentace (funkce nebo kód jazyka).

Další slova: větvení

Při popisu složitějších slov s uvedeným schematem nevystačíme. Pak je musíme popsat slovně:

Větvení

`:if ... :else ... :then`

1. Odstraní vrchol zásobníku ***rslt***.
2. Je-li odstraněná hodnota *Pravda*, vypustí z programového zásobníku vše po následující `:else` s tím, že případné vnořené bloky `:if ... :else ... :then` správně vypouští celé.
3. Je-li *Nepravda*, pokračuje ve vykonávání programu.
4. Samostatné slovo `:else` vypustí z programového zásobníku vše po následující `:then` s tím, že případné vnořené bloky `:if ... :else ... :then` správně vypouští celé.
5. Samostatné slovo `:then` se ignoruje.

Slovo `:if` používá pomocná slova `:seek`, `:skip` a slova `:else` a `:then`, která se společně starají o správné vypouštění částí programového zásobníku, které se nebudou vykonávat. Správně pracují i s případnými vloženými bloky `:if ... :else ... :then`. K tomu slouží pomocný zásobník nazvaný *seek*. Celý postup je vidět ze zdrojového kódu (možná by to šlo zjednodušit, zkuste něco navrhnout).

Zásobníkové programovací jazyky: poznámky

Základním a zřejmě prvním zásobníkovým jazykem je jazyk FORTH (počátky: 1968), jehož základy jsou také volnou inspirací pro náš jazyk. Dalším hodně používaným zásobníkovým jazykem je jazyk PostScript používaný ke komunikacemi s tiskárnami (hlavně laserovými) i v grafických souborech. Jeho následníkem je formát PDF.

Zásobníkové programování bylo nebo je pro svou jednoduchost, efektivitu, paměťovou úspornost (díky postfixové notaci, která umožňuje jednoduchou práci se zásobníkem) a nízkoúrovňovost používáno k aplikacím blízkým hardwaru (vestavěné systémy, standard Open Firmware na počítačích, řízení družic ...). Jeho nespornou výhodou proti podobně orientovanému jazyku C je, že zásobníkové jazyky jsou dynamické (s programy lze snadno interagovat a uživatelsky je měnit).

V současné době jsou také na zásobníkových strojích založeny bytecody některých velmi rozšířených jazyků (Java, Python, jazyky platformy .NET).

Pro nás je zásobníkové programování dobrým příkladem také proto, že díky explicitní manipulaci se zásobníky (hlavně s datovým) pomáhá pochopit funkci zásobníku při běhu programu.

3 Implementace první verze simulátoru

V této části si ukážeme a vysvětlíme zdrojový kód první verze simulátoru zásobníkového stroje. Nejprve popíšeme implementaci zásobníků. Datový zásobník budeme ukládat do proměnné `*rslt*`, programový do `*exec*`. Pomocný zásobník *seek* používaný při větvení programu bude v proměnné `*seek*`.

Proměnné budou dynamické — tím zajistíme, že jejich vazby v čase, kdy budou existovat, budou viditelné odkudkoli ze zdrojového kódu. Proměnné v hlavní funkci `execute` zastíníme. Hlavní důvod je čistě technický: lokální vazby nám umožní snadné prohlížení obsahu zásobníků při krokování v LispWorks. Mohli bychom ovšem také chtít (někdy v budoucnu) spustit více programů současně; každý by pak měl své zásobníky.

```
(defvar *rslt*)
(defvar *exec*)
(defvar *seek*)
```

Zásobník slov také definujeme jako globální proměnnou:

```
(defvar *word*)
(setf *word* '())
```

Hlavní funkcí simulátoru je funkce `execute`, která na vstupu přijímá kód k vykonání a jako výsledek vrací hodnotu, která zbyla na vrcholu hodnotového zásobníku:

```
(defun execute (&rest code)
  (let ((*ret* '())
        (*seek* '())
        (*rslt* '())
        (*exec* code))
    (loop (when (null *exec*) (return))
          (exec-elem (pop *exec*)))
    (pop *rslt*)))
```

Jak vidíme, funkce nejprve vytvoří dynamické vazby na proměnné `*seek*`, `*rslt*` a `*exec*` (a také na proměnnou `*ret*`, kterou budeme používat až příště). Datový zásobník a zásobník *seek* jsou na začátku prázdné, programový obsahuje zadaný kód.

Dále funkce pracuje čistě iterativně. V cyklu vytvořeném makrem `loop` volá funkci `exec-step`. Ta bude realizovat vždy jeden krok výpočtu, dokud nebude programový zásobník prázdný. Zde používáme zatím neznámé makro `return`, které zařizuje výskok z cyklu.

O vyskakování z cyklu

Makro `loop` a makro `return` zde používám v podstatě z nouze, protože jsem vám neřekl o jiných možnostech psaní cyklů v Lispu a protože rekurzivní funkce by se v tomto případě nedala tak snadno ladit. Obecně ale platí, že vyskakování z prostřed cyklu je lepší se vyhnout.

Funkce `execute` na závěr pomocí makra `pop` vrací hodnotu uloženou na vrcholu datového zásobníku a ze zásobníku ji odstraní. (Pokud byl program správně napsán, měl by pak zásobník zůstat prázdný.)

Funkce `exec-elem` vykonává jeden krok výpočtu. Jako argument obdrží prvek z vrcholu programového zásobníku (už na něm není). Pokud je tento prvek slovo, vykoná je. Jinak prvek uloží na datový zásobník.

```
(defun exec-elem (elem)
  (if (wordp elem)
      (exec-word elem)
    (exec-val elem)))
```

Funkce na prvek zavolá pomocnou funkci podle jeho typu. Typ testuje funkcí `wordp`:

```
(defun wordp (el)
  (keywordp el))
```

Vykonání hodnoty je jednoduché, stačí ji uložit na vrchol datového zásobníku:

```
(defun exec-val (elem)
  (push elem *rslt*))
```

Vykonání slova znamená spuštění jeho kódu zjištěného funkcí `word-code`, kterou napíšeme později. V případě vestavěného slova (primitiva) je kódem funkce bez parametru, kterou je třeba spustit, v případě uživatelsky definovaného slova jeho kód v jazyce, který je nutno uložit na programový zásobník, kde pak bude v dalších krocích vykonáván:

```
(defun exec-word (word)
  (let ((code (word-code word)))
    (if (functionp code)
        (built-in-exec code)
      (user-exec code))))

(defun built-in-exec (fun)
  (funcall fun))

(defun user-exec (uc)
  (setf *exec* (append uc *exec*)))
```

Dále potřebujeme naprogramovat práci se slovy. Zásobník *word* má být **slovník**, má tedy obsahovat páry, které mají v *Car* klíč (symbol) a v *Cdr* jeho kód. Věřím, že následující operace na práci s obecnými slovníky nevyžadují komentář:

```
(defun dict-find (key dict)
  (let ((pair (find key dict :key #'car)))
    (unless pair (error "Key not found in dictionary: ~s" key))
    (cdr pair)))

(defmacro dict-push (key val dict)
  `(push (cons ,key ,val) ,dict))
```

V případě slovníku ***word*** bude klíčem vždy slovo a hodnotou kód, který má slovo vykonat. U vestavěných slov bude kódem vždy lispová funkce bez parametru, u uživatelsky definovaných seznam s kódem našeho jazyka.

Zjištění kódu daného slova obnáší jeho vyhledání ve slovníku ***word***:

```
(defun word-code (w)
  (dict-find w *word*))
```

K definici slova slouží funkce `define-word`, která na základě jména slova (kterým je, jak víme, klíč) a jeho kódu popis slova uloží na zásobník *word*:

```
(defun define-word (name code)
  (dict-push name code *word*))
```

K definici vestavěných slov slouží makro `defprim`, které je na této funkci postavené. Jako argumenty přijímá název nového slova a výrazy Lispu, jeichž postupným vyhodnocením se slovo vykoná (s vedlejším efektem na zásobnících):

```
(defmacro defprim (name &body body)
  `(define-word ',name (lambda () ,@body)))
```

S jeho pomocí nejprve definujeme slova na manipulaci s datovým zásobníkem:

```
;; (a b - b a)
(defprim :swap
  (let ((b (pop *rslt*))
        (a (pop *rslt*)))
    (setf *rslt* (append `(,a ,b) *rslt*))))

;; (a b c - b c a)
(defprim :rot
  (let ((c (pop *rslt*))
        (b (pop *rslt*))
        (a (pop *rslt*)))
    (setf *rslt* (append `(,a ,c ,b) *rslt*))))

;; (a - )
(defprim :drop
  (pop *rslt*))

;; (a - a a)
(defprim :dup
  (push (car *rslt*) *rslt*))

;; (a b - a b a)
(defprim :over
  (push (cadr *rslt*) *rslt*))
```

Slova realizující aritmetické operace všechna odebírají dva prvky ze zásobníku hodnot a zpět vracejí jeden, který je výsledkem aplikace nějaké funkce. Jejich definici tedy lze zjednodušit použitím pomocné funkce `word-bin-op`, která jako argument přijímá funkci, která se má na dva prvky použít:

```
(defun word-bin-op (fun)
  (let ((b (pop *rslt*))
        (a (pop *rslt*)))
    (push (funcall fun a b) *rslt*)))
```

Definice aritmetických operací a porovnávání čísel:

```
;; (a b - součet a + b)
(defprim :+
  (word-bin-op #'+))

;; (a b - rozdíl a - b)
(defprim :-
  (word-bin-op #'-))

;; (a b - součin a * b)
(defprim :*
  (word-bin-op #'*))

;; (a b - podíl a / b)
(defprim :/
  (word-bin-op #'/))

;; (a b - log. hodnota a = b)
(defprim :=
  (word-bin-op #'=))
```

Abychom mohli uživatelská slova definovat přímo v jazyce, napíšeme si na to slovo `:def`. Slovo použije vrchol programového zásobníku jako název nového slova a celý zbytek jako kód. Nové slovo definuje a pak programový zásobník vymaže (aby se jej stroj nepokoušel vykonat) a na datový zásobník dá jako výsledek definované slovo.

```
(defprim :def
  (let ((word (pop *exec*)))
    (define-word word *exec*)
    (setf *exec* '())
    (push word *rslt*)))
```

Slovo `:def` používáme například k definici uživatelského slova `:rot-`:

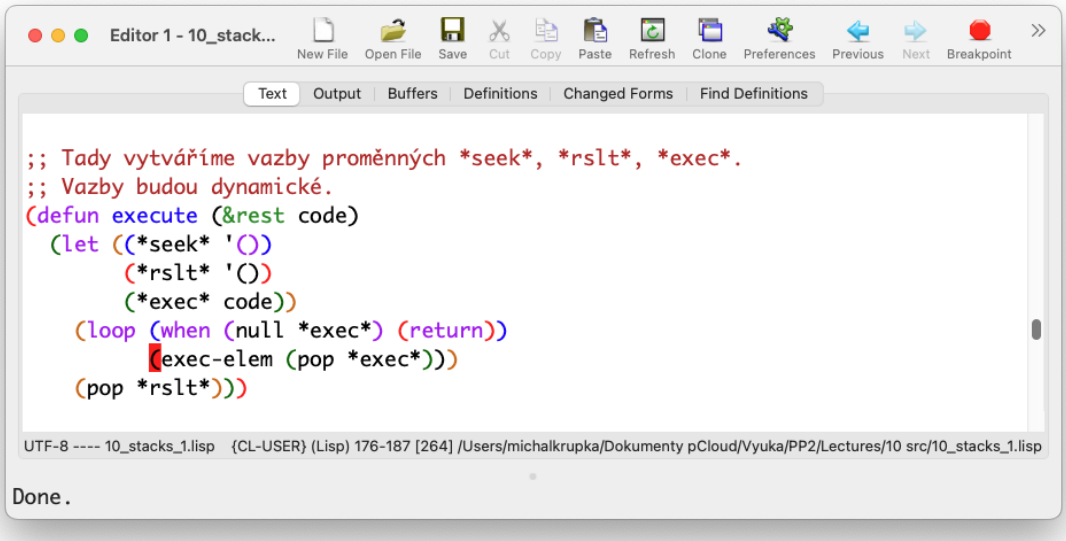
```
:def :rot- :rot :rot
```

Slovo jako jediné používáme v prefixové notaci, protože potřebujeme zabránit vykonání kódu, který je za ním.

4 Práce se simulátorem

Simulátor pracuje čistě iterativně. Nepoužívá skrytě žádnou paměť kromě zásobníků a lokálních proměnných slov definovaných lispovými funkcemi (tj. vestavěných slov). Po vykonání každého slova je veškerá informace o stavu programu uložena na zásobnících. To přináší možnost snadno a jednoduše sledovat vykonávání programu.

Když umístíme breakpoint do funkce `execute` sem:



a definujeme slovo `:fact`, jak je ukázáno v příkladech v kódu:

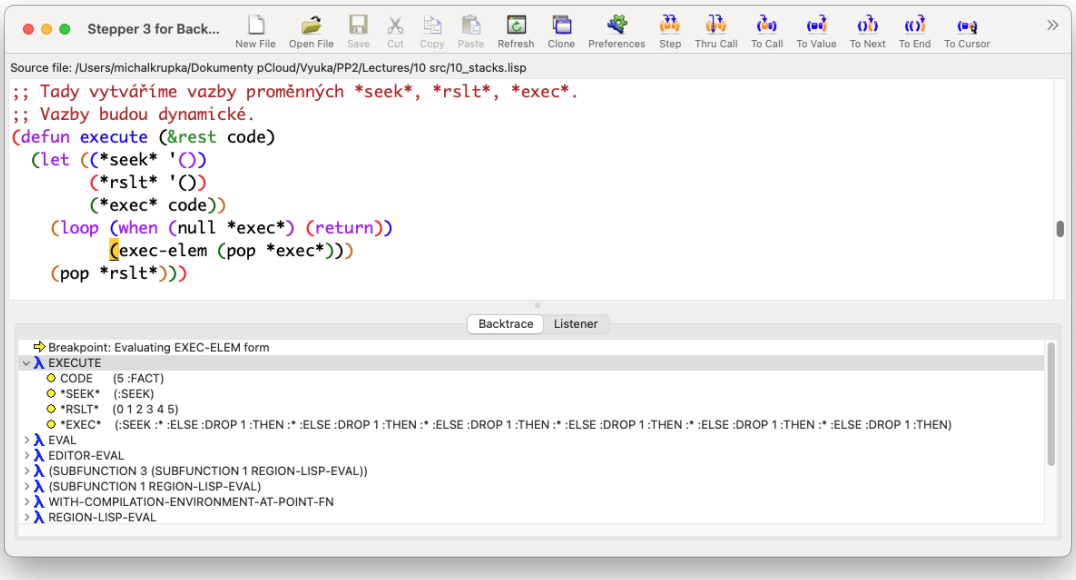
```
(execute :def :fact
             :dup 0 := :if :dup 1 :- :fact :* :else
                           :drop 1 :then)
```

můžeme krok po kroku sledovat práci stroje např. na programu `5 :fact`. Po zadání

```
(execute 5 :fact)
```

se výpočet zastaví před každým krokem a ve výpisu zásobníku, který nám LispWorks ukáže, vidíme obsah zásobníků. Další krok spustíme kliknutím na tlačítko  (*Continue*).

Například po 50 krocích budou zásobníky vypadat takto (dole v lokálních proměnných funkce `execute`):



Otázky a úlohy na cvičení

U všech úloh na zásobníkový stroj se nová slova pokuste definovat jako uživatelská. Jen když to nepůjde, napište je jako vestavěná.

1. Funkce `f` a `g` jsou definovány takto:

```
(defun f (x)
  (funcall (g 'b)))

(defun g (x)
  (lambda () x))
```

   Co bude hodnotou výrazu `(f 'a)`, pokud budou všechny vazby lexikální, a co, pokud dynamické?

2. Definujte slovo `:hyp` na výpočet délky přepony pravoúhlého trojúhelníka z délek odvěsen:

```
(a b -- c)
```

3. Definujte slovo `:discr` na výpočet diskriminantu kvadratické rovnice:

```
(a b c -- diskriminant)
```

4. Definujte jako slova logické operace: `:not`, `:and`, `:or`.

5. Napište slovo na umocňování čísla daným celočíselným exponentem pomocí násobení. Udělejte to jak klasickou rekurzí, tak iterativně.

6. Definujte slovo `:+n` na součet n čísel. n je na vrcholu zásobníku, čísla pod ním:

```
(a1 ... an n -- a1 + ... + an)
```

   V první verzi můžete předpokládat, že n není nula. Pak tuto možnost přidejte.

7. Definujte následující slova na práci s páry:

   `:cons` $(a\ b$ – `(`a `.` b`)`$)$
   `:split` $($`(`a `.` b`)` – $a\ b)$

8. Pomocí slov `:cons` a `:split` napište slova pracující rekurzivně se seznamy: délku, spojení, převrácení, hledání apod.

9. Na reálném zásobníkovém stroji by se uživatelská slova nenahrazovala svým významem za běhu, ale fungovala by spíše jako makra, která se expandují během překladu. Upravte simulátor tak, aby funkce `execute` nejprve v programu nahradila všechna uživatelská slova jejich významem (v programu by zůstala jen vestavěná slova) a pak by teprve program vykonala. Takto upravený simulátor bude vylučovat rekurzivní volání. Navrhněte proto nějaké vhodné slovo na podmíněný cyklus (pozor, zřejmě pak bude potřeba upravit i definici slova `:if`). Nakonec pro upravený simulátor implementujte několik vybraných rekurzivních programů (faktoriál, mocninu, součet n čísel ...).